DANIELLE GEORGIA MARSHALL

GRADED MODAL TYPES FOR MEMORY AND COMMUNICATION SAFETY

GRADED MODAL TYPES FOR MEMORY AND COMMUNICATION SAFETY

DANIELLE GEORGIA MARSHALL

a thesis submitted to
the University of Kent
in the subject of Computer Science
for the degree
of Doctor of Philosophy (PhD).

March 2025

Danielle Marshall: *Graded Modal Types for Memory and Communication Safety,* PhD thesis, © March 2025

ABSTRACT

Using types to delineate various kinds of values is a ubiquitous technique in modern programming. However, one common pattern is for a piece of data to represent access to a resource (such as a reference or channel) which needs to be used in a certain way. This presents challenges for using types to ensure software correctness, as we need to reason about *how* data is used rather than *what* data is being used.

Linear types are one approach to this problem, separating data that can be used any number of times from data that will be used exactly once. Graded type systems such as the one underlying the Granule research language generalise this to allow for more precise restrictions on exactly how data must be used. However, existing graded systems generally focus on describing local properties pertaining to single pieces of data, whereas sometimes we are interested in global properties that hold over an entire program.

In this thesis, we demonstrate that the framework of graded types can be extended to allow for the verification of global program behaviour, encapsulating key properties from the paradigms of both *memory* and *communication* safety.

First, we discuss memory safety. We describe in detail the relationship between the two similar but contrasting ideas of linear types and uniqueness types, which ensure that only a single mutable reference to a value can exist, and then embed uniqueness into an existing graded calculus by introducing a new flavour of modality. We then extend this system to describe more general notions of ownership and borrowing as popularised by the Rust programming language, using a new form of fractional grading which allows for multiple immutable references to a single value. The overall system is made formal through a heap-based operational model.

Second, we discuss communication safety. We first present a formalisation of session types alongside grading, using an operational model extended with processes and channels to show that concurrency issues are ruled out and resolving problems with Granule's prior instantiation of session-typed channels. We then introduce additional primitives for non-linear communication, allowing for essential communication patterns that cannot be described with purely linear session types to be precisely represented. All of the systems we discuss are fully implemented in Granule.

ACKNOWLEDGMENTS

I'm afraid this isn't going to be brief, but now that I finally have some space to thank all of the people I'd like to thank, I'm going to grab the opportunity with both hands. It's been a long five years, and so many people have played a part in finally getting me to the end. I'm thankful for you all!

RHONA First of all, I need to thank the wonderful Rhona for the unfailing love and support she has given me over the past couple of years. We both went through quite a lot of stress getting our PhDs over the line, but we both got our theses written and passed our respective vivas, just like we kept telling each other that we could! I would say I couldn't possibly have done it without her, but I have a feeling she'd reply that I was being silly, and of course I could still have managed it either way. Instead, I'll say that I have no idea whether I could have done it without her, but there is one thing that I do know for sure: I'm infinitely grateful that I didn't have to. ♡

UNDERGRADUATE MENTORS & SUPERVISORS I would never have been in a position to write this thesis if it wasn't for the advice I received and lessons I learned from those who I worked with when I was an undergraduate, back home in Sheffield.

Thanks to Georg Struth who supervised my undergraduate dissertation on program verification with Isabelle, and introduced me to some of the things that functional programming makes possible while also finding the time in his busy schedule to give me advice on applying for PhDs. Thanks also to Dirk Sudholt who supervised our Darwin (master's) project on graph colouring algorithms; although the onset of the pandemic scuppered our overly optimistic plans to get that work published, I still learned a great deal about the process of research and working as part of a larger group.

Credit is also due to those who I worked on summer projects with. I developed an agent-based model of tuberculosis transmission under the mentorship of Pete Dodd, who told me my work was of the quality he might expect from a typical postdoc. Well, now I am a postdoc, so hopefully my work continues to live up to that standard! I also learned the basics of category theory from Jordan Williamson, who was himself a PhD student at the time. I appreciate his patience, and though I remember absolutely nothing about Morita theory, at least I've managed to hold on to the knowledge of what a functor is.

LECTURERS & TEACHERS I first learned of the existence of Haskell from the Advanced Programming Topics module I took in my second year at Sheffield, and as any of my friends at the time could tell you, I became enamoured with the language and with the notion of functional programming in general very quickly indeed. I would be remiss to not thank the lecturers, Phil Green and Mike Stannett, since the things I learned in that module formed a foundation for the things I have been thinking about over the last four years. I'm also very grateful to many of my other lecturers at Sheffield; there are far too many to thank individually, but the broad education in mathematics and computer science that I received there is something that I will value for the rest of my life.

Even before I started at university, though, there were many teachers who helped me onto the road that led me here. Thanks to Jon, Sharon and Ruth from the maths department at Longley, who fostered (and sometimes put up with) my enthusiasm for the subject, and particularly to Jon who supervised my extended project on the mathematics of artificial intelligence, which led to me switching tracks from pure maths to a dual degree involving computer science. Thanks also to Andrew for not being too upset that I didn't continue to pursue chemistry after getting the A-level! Going back even further, thanks to Mr Gardner who was the first maths teacher to make me feel true passion in his lessons, but thanks also to Mr Storey and Miss Tomlinson, the history and English teachers who imparted many of the ideas that would later grow into my research skills.

INDUSTRY MENTORS & COLLEAGUES I've been fortunate enough to have a few opportunites to apply my knowledge in the 'real world', both before and during my PhD. Thanks to Chris Smith who mentored my Google Summer of Code project on writing a requirements checker plugin in Haskell for the CodeWorld platform. This gave me a very valuable experience of working on a larger scale Haskell project, without which getting up to speed in my first year at Kent would have been significantly more difficult.

A couple of years into my PhD, Dan Ghica offered me a research internship on his team at Huawei's research centre in Edinburgh, working on some new ideas for their programming language Cangjie. Though the things we were looking at never came to fruition, I'm grateful for the opportunity to think about how my research fits into a more practical setting and also for the chance to meet and get to know many of the programming languages researchers in Scotland. It became clear to me before too long that this was a tight-knit community that I'd like to be a part of, and now that I live and work here I'm glad to have realised that at the right time.

Thanks also to those who work with me on the Stardust project here in Glasgow, where I am currently sitting as I write these acknowledge-

ments. You had to put up with me spending time on my thesis for a little bit longer than I probably should have been, so I'm sure we're all very relieved that it's finished!

DOMINIC Some of the biggest thanks of all have to go to Dominic Orchard, who has supervised me for the last four years as I've worked on my PhD. I couldn't have asked for a better supervisor than Dominic, and I have no doubt that any of his future students would have a similar experience, whether their work is on graded types or climate science or whatever other field he might go on to do research in!

Dominic has always given me the freedom to work on the things I find most interesting; none of the chapters in this thesis bear any resemblance to the initial proposal I submitted when I first came to do research at Kent, and while I'm glad some of that work is still ongoing (spearheaded by Vilem Liepelt - thanks for picking it up!) I am also glad that I was able to prioritise following my passions. Dominic was an enormous asset when it came to getting papers submitted, often staying up until frankly ridiculous times along with me as I made an ill-considered push to meet a deadline. Research aside, though, I always knew that I could rely on Dominic for anything I might need, and I'm endlessly grateful for his wisdom, patience, and support. I'm sure he's very glad that I got my thesis written before the real deadline, even if I didn't make it in time for many of the other deadlines we imagined for ourselves along the way.

GRANULE A lot of interesting research has happened and continues to be done as part of the overarching Granule project by many talented people, and I am grateful for the help and support they have given me both while I was getting to grips with the language at the start of my PhD and when we were writing together in later stages.

Thanks to Harley Eades who has done a huge amount of great work on the project all the way from Augusta, and has given me a lot of useful advice. Thanks to Mike and Tori Vollmer, both of whom I've had the pleasure of working with on very different papers and both of whom I've learned from. And thanks to the other PhD students who have been working on Granule alongside me: Jack Hughes, Paulo Torrens, and Vilem who I've already mentioned. Granule is a very cool project and I think there's a lot of fascinating work still to be done using it as a vehicle, and it's been a pleasure spending a few years working on it with all of these people.

KENT & PLAS Kent has been a wonderful place to do research (even if being in Canterbury for the summers has sometimes felt a bit like trying to work inside an oven), and it would have been impossible to finish a PhD that I started during the lockdowns at the apex of a

global pandemic if it wasn't for the fantastic research environment that is the the Programming Languages and Systems research group.

Thanks to all of my officemates in both Cornwallis and Kennedy: it wouldn't have been the same without you, especially without the fun conversations I had with Octave, Sophie, Caio and Jonah. Not that we weren't all very focused on getting our papers written and code compiling, of course! Thanks to my supervisory panel Olaf, Rogerio, and David for keeping me on track, even if it meant far too many meetings spent asking me when I was finally going to be done with this thesis. And thanks to Vineet Rajani for spending a few months working with me and teaching me about how to write grant proposals; hopefully that knowledge will come in handy some day soon.

FRIENDS Outside of PLAS and the broader School of Computing, I've been lucky to meet many other amazing friends while living in Canterbury. Special thanks go to the Sunflower Seeds (don't ask): I've mentioned Rhona already, of course, but thanks also to Zöe for running many online events in my first year that led to me getting to know so many lovely people in the first place, to Holly for spending two years living with me and being the best housemate I could have asked for, and to Naveed for always being a source of good humour in dull times. Thanks also to Kaz, who was my neighbour when I lived in Woolf and a great friend too.

Going back to my time as an undergraduate, I can't possibly forget to thank my closest friends from my time at Sheffield, Ana and Katie. I know we don't talk as much these days, but as far as I'm concerned, we are still besties! They made those years spent studying a thousand times better, and I still think of them all the time. I also need to thank all those who I spent two years working alongside on the LGBT+ representative committee; particular thanks go to the chairs, Ciara, Marlowe, and Fran, who made that committee an excellent place to find myself while also working to make things a little better for other students like me.

There are so many other friends I've had over the course of my life, many of whom I've lost touch with while this thesis has taken over my waking moments, so I'd like to use this space to say that if you still think of me fondly, I probably still think of you fondly as well. If any of you are reading this, I'd be more than happy to hear from you.

COMMUNITY The broader international programming languages research community is a community that I feel very proud and lucky to be a part of. It was really only through coincidence and happenstance that I ended up going to and volunteering at ICFP 2019 in Berlin when I was still an undergraduate, and it was only thanks to the people I met on that excursion that I have found myself working in this field to begin with. In fact, it was Vilem's talk about Granule at that very

conference that caused me to apply to do a PhD with Dominic in the first place; and the rest is history! Since then I have had the privilege to attend ESOP 2022 in Munich, POPL 2024 in London and OOPSLA 2024 in Pasadena (along with countless virtual conferences and workshops during the pandemic), and each time has felt like a priceless opportunity to reconnect with the community that brought me into this career and that I deeply appreciate.

Thanks also to everyone who I've spoken to about research and otherwise on both the website formerly known as Twitter and more recently on the Mastodon server `types.pl` and in the broader Fediverse. I've always been the sort of person who socialises more easily online than in person, and through these platforms I've been able to connect with people who are thinking about similar things to me all around the world. The last few years haven't always been the easiest years of my life, and as we all know, oversharing on social media is how people these days get through this kind of thing; here's to you all for putting up with it!

GANDALF For the first year of my PhD it was difficult to meet up with any humans in person thanks to the lockdown restrictions, so as unusual as it might be to acknowledge a cat who doesn't belong to me: I'd like to thank Gandalf (the Black and White, Second of his Name, Protector of Postgraduates and so on) for being a constant companion during that early period, and making my time living in Woolf College more than a little brighter. Once upon a time it was tradition for him to offer a blessing upon any dissertation or thesis that was presented to him; though we have both moved away from campus, I hope I will still be lucky enough to receive his blessing from afar.

EXAMINERS Gratitude of course must go to my external examiner Neel Krishnaswami and my internal examiner Marco Paviotti, who made the viva I was predictably anxious for into a relaxed and engaging conversation about my research, and whose corrections helped to make this document what it is today. Thanks very much to you both!

NANNAN Unfortunately, my nannan Carole passed away during the time period in which I was working on this thesis. It's sad that she never got to hear about me finally being done with it, and even more sad that she never really got to meet the person I turned out to be, but in both instances I hope she would have been proud.

MUM & DAD Lastly, of course, I need to thank my mum and dad for their neverending belief in me, over the course of my PhD but also throughout my life beforehand and no doubt set to continue on into my career. I couldn't have gotten this far in life without such supportive and loving parents, and I'm lucky to have them.

CONTENTS

1 Introduction 1
- 1.1 Contributions 3
- 1.2 Outline 6
- 1.3 Publications 7

I Linear Types and Graded Modalities

2 A Piece of Cake: Background on Linearity 11
- 2.1 Terminology 11
- 2.2 Linearity and Substructurality 12
  - 2.2.1 The Linear λ-Calculus 13

3 Making the Grade: A Graded Modal Primer 17
- 3.1 A Brief History of Graded Modal Types 17
- 3.2 Granule and Graded Modalities 18
- 3.3 A Core Graded Modal Calculus 21
  - 3.3.1 Syntax 22
  - 3.3.2 Type System 22
- 3.4 Operational Semantics 27
  - 3.4.1 Metatheory 28

II Uniqueness Types and Graded Resources

4 One of a Kind: Uniqueness and Sharing 33
- 4.1 Introduction 33
- 4.2 Key Ideas 35
  - 4.2.1 Are Linearity and Uniqueness the Same? 35
  - 4.2.2 Are Linearity and Uniqueness Dual? 39
- 4.3 The Linear-Cartesian-Unique Calculus 41
  - 4.3.1 Typing 42
  - 4.3.2 Equational Theory 43
  - 4.3.3 Exploiting Uniqueness for Mutation 45
  - 4.3.4 Operational Heap Model 46
  - 4.3.5 Metatheory 49
- 4.4 Implementation 52
  - 4.4.1 Frontend 52
  - 4.4.2 Compilation and Evaluation 54
- 4.5 Conclusion 56

5 Divide and Conquer: Ownership and Borrowing 57
- 5.1 Introduction 57
- 5.2 Key Ownership and Borrowing Concepts 59
- 5.3 Owned is Unique 62
- 5.4 Immutably Borrowed is Fractionally Unique 66
  - 5.4.1 Equational Theory 75
  - 5.4.2 Partial Views via Distributive Laws 75

5.5 Semantics and Metatheory 79
5.5.1 Preliminary Definitions 80
5.5.2 Single-Step Reduction 81
5.5.3 Theorems 87
5.6 Related Work 91
5.7 Conclusion 94

III Session Types and Graded Channels
6 Message in a Bottle: Linear Session Types 97
6.1 Introduction 97
6.2 Linear Session-Typed Channels 99
6.3 Grading, Call-by-Value, and Effects 104
6.4 Semantics 106
6.4.1 Operational Model 106
6.4.2 Runtime Typing 111
7 Replicate, Reuse, Repeat: Non-Linear Communication 115
7.1 Introduction 115
7.2 Grading for Non-Linear Communication 115
7.2.1 Reusable Channels 115
7.2.2 Replicated Servers 120
7.2.3 Multicast Sending 124
7.3 Properties 126
7.4 Related Work 129
7.5 Discussion and Conclusion 132

8 Further Work 133
8.1 Guarantees 133
8.2 Linear Haskell 136
8.3 Adjoint Models 137
8.4 Ordered and Dependent Types 137
8.5 Performance Improvements 138
8.6 Counting Permissions 138
8.7 Non-Lexical Lifetimes 138
8.8 Recursion and Other Combinators 139
8.9 Applications of Graded Sessions 139
8.10 Other Resource Allocators 140
8.11 Uniqueness and Graded Sessions 140
8.12 Deadlock Freedom 140
9 Conclusions 143
9.1 Summary 143
9.2 Discussion 144

Bibliography 145

Appendices

A Appendix for Chapter 3 165
A.1 Collected rules 165
A.1.1 Typing 165
A.1.2 Reduction rules for heap semantics 166
A.2 Substitution proofs 166
A.3 Type safety 173
A.3.1 Value lemma 173
A.3.2 Progress proof 175
A.3.3 Preservation proof 177
B Appendix for Chapter 4 181
B.1 Collected rules 181
B.1.1 Typing 181
B.1.2 Reduction rules for heap semantics 182
B.2 Substitution and weakening proofs 186
B.3 Type safety 198
B.3.1 Value lemma 198
B.3.2 Progress proof 201
B.3.3 Conservation proof 207
B.4 Uniqueness proof 218
B.5 Soundness of heap model wrt. equational theory 233
C Appendix for Chapter 5 237
C.1 Collected rules 237
C.1.1 Typing 237
C.1.2 Reduction rules for heap semantics 241
C.1.3 Equational theory 244
C.1.4 Parallel sum example in Granule 244
C.2 Substitution proofs 246
C.3 Type safety 251
C.3.1 Value lemmas 251
C.3.2 Progress proof 257
C.3.3 Preservation proof 272
C.4 Uniqueness and borrow safety proofs 292
C.5 Soundness of heap model wrt. equational theory 315
D Appendix for Chapters 6/7 321
D.1 Collected rules 321
D.1.1 Typing 321
D.1.2 Global configuration reduction rules 323
D.1.3 Standard small-step reduction rules 324
D.2 Auxiliary results 324
D.3 Type safety 341
D.3.1 Progress proof 341
D.3.2 Preservation proof 353
D.4 Further comparison with GV 365
D.5 Involution is an isomorphism 366

1 INTRODUCTION

Software is involved in every aspect of life in our modern society, but designing high quality pieces of software that are also reliable is one of the biggest challenges in programming. The more complex a piece of software becomes, the more difficult it is to be aware of everything that could go wrong. A 2022 report by the *Consortium for Information & Software Quality* determined the total cost to the US economy from software bugs that year to be $2.41 trillion [74], so it is clear that finding ways to reduce their prevalence is essential. This becomes even more salient when considering systems where safety is critical and any failure of the software could be disastrous; one well-known example is the case of the Therac-25 radiation therapy machine, which gave some patients massive overdoses of radiation due to race conditions in its concurrent programming [76].

Formal verification aims to resolve these issues by proving the correctness of a program using mathematical techniques, often assisted by computational tools. Such methods have seen wide use in the realm of pure mathematics [53], and more recently have also been applied to all kinds of industrial software projects. One high profile example is the CompCert project, which sought to verify the correctness of a compiler for a significant subset of the C programming language using the Rocq proof assistant. Being able to verify the correctness of a compiler is particularly important, since the verification of the compiler helps to guarantee that safety properties proved on the source code hold for the executable compiled code also [75]. Another example of real world software verification is the seL4 microkernel [71], formally verified using Isabelle/HOL. The security and reliability of a program can only be as good as that of the underlying operating system, so verification is crucial here since any fault in the kernel's implementation has the potential to undermine the correct operation of the rest of the system.

However, as a piece of software increases in complexity any proof of its correctness must likewise become more complex, and any software that sees practical use in the 21st century will certainly present challenges when it comes to verifying every aspect of its behaviour. CompCert comprises over 35,000 lines of Rocq code, where 87% of this is made up of proofs [75]. Similarly, the seL4 microkernel specifications are around 480,000 lines in Isabelle/HOL [71]. There is significant value to be found in aiming for a happy medium between the lofty goal of formally verifying every property of a program and the default approach of simply writing tests which check individual cases.

One lightweight approach to verifying properties about programs is through static type systems; these provide a convenient way to track *what* data the programmer is working with, enabling the compiler to catch a subset of mistakes before a program is ever run. More recently, however, *substructural* type systems [148] in particular are growing in popularity because they allow for a *resourceful* interpretation of data, which can be used to rule out bugs relating to *how* data is used.

These systems were developed (via the *Curry-Howard correspondence* which links logic with type theory) from ideas originating in substructural logics, where the *weakening* rule (corresponding to the ability to discard a value) and the *contraction* rule (corresponding to the notion of duplication) from intuitionistic logic are restricted. This captures the well-known maxim that "not all things in life are free" [146]; many kinds of data behave *resourcefully*, and are subject to constraints on their usage. Sensitive data should not be infinitely duplicated and passed around freely, file handles should not be arbitrarily discarded without being properly closed, and communication channels should not be used without adherence to a fixed protocol, to name a few!

Indeed, the idea of substructurality is finally taking hold in modern programming; while several languages have implemented linear type systems over the years, including ATS [156], Alms [134] and Quill [96], Haskell now has linear types as of the ninth version of the Glasgow Haskell Compiler [19] which are roughly based on Girard's linear logic [50], bringing linearity closer to the mainstream.

This idea is developed by languages with even more expressive *graded* type systems such as Granule and Idris, which build on the idea of linearity by allowing more fine-grained restrictions on resource usage. Such systems allow for precise properties of programs to be expressed, such as *quantitative* restrictions on how many times a value can be used or more varied properties such as ensuring that values are accessed at an appropriate level of security [103]. Sometimes it is even possible to express behaviour so tightly that programs can be synthesised directly by the compiler [64].

Granule in particular is the language we will focus on and build upon in this thesis, as it provides a mature and feature-rich implementation of linear and graded types which acts as a convenient foundation for the extensions we will describe throughout later chapters [103].

Granule offers two kinds of *modality* which act as wrappers annotated with grades that describe how the data under the modality behaves: a *coeffect* modality, written $\Box_r$ or `[r]` (describing data that makes demands on its context, and so must *necessarily* be used in a particular way according to its grade r), and an *effect* modality, written $\Diamond_r$ or `<r>` (describing data that might change its context, in *possible* ways that are described by the grade r) [47].

To illustrate how this connects to the ideas above, we present some small examples to build some intuition. Consider the classic example

of a function which cannot be represented using linear types, assuming an interface where `eat : Cake → Happy` and `have : Cake → Cake`:

```
impossible : Cake → (Happy, Cake)
impossible cake = (eat cake, have cake)
```
Granule ✗

Note that Granule's function type → is the type of linear functions, more traditionally written ⊸. The above function is ill-typed and the Granule compiler will brand it with a linearity error; this is because the value of type `Cake` passed into the function is a linear resource, and the body of the function requires us to duplicate it (via contraction), which is forbidden. Thus, linear types remind us of the familiar aphorism: *you can't have your cake and eat it too*.

In order to fix this, we could resort to giving up on linearity altogether (and we will see how this can work in more detail later), but Granule's graded coeffect modality allows us to find a happy medium: we can precisely describe how the value is used by using a natural number grade. See the example below:

```
possible : Cake [2] → (Happy, Cake [1])
possible [cake] = (eat cake, [have cake])
```
Granule

Here, the box modality is annotated with the grade 2, meaning that the value of type `Cake` needs to be used exactly twice. Applying the `eat` function accounts for one usage, meaning that one usage is preserved for later. Another way to think of this is that if you start with two cakes and consume one, you are left with one remaining. We will build on this and describe other forms of grading in Chapter 3.

1.1 CONTRIBUTIONS

The combination of linear and graded types allows for precisely specifying behaviour of each piece data within a program, in such a way that the behaviour is enforced statically by the type checker. But how can we use linearity and grading to enforce *global* properties, which hold over an entire program rather than for an individual resource?

Memory safety is a guarantee that situations in which a program reads or writes from memory that may lead to undefined behaviour are not permitted to occur. Some common examples are reading outside the bounds of an array, reading from uninitialised memory, or indeed accessing memory through a pointer to an invalid location. A large proportion of bugs in modern industrial code arise due to improper management of memory safety, often due to the wide usage of systems languages like C that have significant efficiency benefits but provide very little guarantee of any safety properties [102, 117].

Using linear types pervasively throughout your program can provide some important memory safety properties; every value being used exactly once means that data races or multiple concurrent accesses to the same location in memory cannot occur, and linearity allows for the proper deallocation of references to be enforced. However, it is often necessary to make use of non-linear values for many practical programming patterns, and as soon as non-linearity is introduced these properties become much harder to enforce. *Uniqueness types*, which ensure that every value has precisely one reference [16], offer a more viable solution to these problems, but though they are certainly related to linear types the precise connection is not entirely clear in the existing literature.

One example of a modern programming language that integrates memory safety as a core principle of its design is Rust. The Rust language aims to achieve memory safety through an intricate system of ownership and lifetimes. The concept of borrowing allows for multiple references to exist that point to the same location in memory, but only temporarily and in situations that are understood to be safe. The original design of Rust's ownership system took inspiration from the literature on linear types [92]. However, the system used in modern Rust is complex, relying on static analysis featuring many ad-hoc extensions, and so the best way to model it in terms of linearity (or, indeed, uniqueness) is even less clear as things stand.

Communication safety, conversely, guarantees that situations where multiple separate parts of a program interacting by sending and receiving information between one another cannot lead to undefined behaviour. Some standard examples in this case are ensuring that messages must always be sent and received in the expected order, and that the data being communicated is of the expected type. Communicating behaviours are essential in concurrent code for transferring information between threads, and so as one might expect many of the bugs that occur in such programs are caused by failures of these aforementioned communication properties [86, 135], even in languages such as Go that are designed around concurrency.[1]

In the domain of type theory, one popular framework for providing communication safety guarantees is the notion of *session types*. These are generally applied to the setting of channel-based communication, where interaction occurs over a channel that a sender and receiver both have access to. (Another related but distinct framework is that of *mailbox types*, which are outside the scope of this thesis but allow for similar guarantees in languages like Erlang which feature *actor*-based communication [40, 43].) Assigning a session type to a channel allows for the type of data being communicated to be specified, but also restricts communication interactions to a specified protocol, with the channel typically being treated as linear to prevent this ordering

1 https://go.dev/blog/survey2025

from being circumvented [62]. However, many possible communication patterns that are used in real-world programs require non-linear behaviour, such as a single server that interacts with many clients or a channel that needs to be used multiple times. Thus, linear session types can often be somewhat restrictive in practice.

In this thesis, we aim to demonstrate that graded types are an expressive enough framework to encompass both of the above safety properties while allowing for programs to be written that cannot be represented using just linearity. We elucidate the relationship between linearity and uniqueness, before showing that a form of 'graded uniqueness' can capture notions of borrowing akin to those in Rust. These allow for more flexible management of references that still ensures memory safety. We also extend a framework of linear session types using grades to encapsulate various non-linear communication patterns while still preserving communication safety. In other words:

THESIS STATEMENT The framework of graded types can be extended to allow for the verification of global program behaviour, encapsulating key properties from the paradigms of both *memory* and *communication* safety.

The critical obstacle to this endeavour in both cases is that graded types excel at capturing *local* properties by tracking information about a piece of data at the type level, whereas both memory and communication safety are *global* properties. It is impossible to know that a memory access is safe without having information about other references that may exist elsewhere, and it is similarly impossible to know that a communication is safe without having information about the state of other communicating parties in the system. We will show that it is in fact possible to use the local information tracked through grading in order to prove such global properties, and address some challenges that arise in doing so; it turns out that some of the key solutions can be transferred across domains, applying to both memory and communication safety and perhaps beyond.

Specifically, this thesis makes the following contributions:

- We identify the key similarities and differences between *linear* types and *uniqueness* types (which ensure memory safety in a coarse-grained way, by requiring that each value has at most one reference at a given time), and extend Granule's core calculus with a new modality for uniqueness.
- We develop a new form of grading for the uniqueness modality, based on fractional permissions, which encapsulates various ideas relating to ownership and borrowing (as popularised by the Rust programming language) and allows for more flexible programming than pure uniqueness.

- We refine and formalise the existing embedding of linear session types into Granule, repairing several problems that were inherent to the previous approach and showing that our improved formulation respects communication safety properties.
- We extend this embedding to include several new primitives for non-linear communication patterns, each using precise notions of grading to continue enforcing communication safety while still allowing for forms of concurrency that are generally not possible to represent in standard linear session type systems.

1.2 OUTLINE

This dissertation is structured into nine chapters; the six chapters succeeding this one follow a general structure of being three parts split into two halves each, where the first half describes how a linearity restriction can allow us to ensure that some global property holds over a program, and the second half shows how more precise grading can allow for more fine-grained reasoning about the behaviour of the program while still preserving the same guarantees. Chapter 2 explains some of the necessary background on linear types and their origins in Girard's linear logic, taking inspiration from the presentation in the first section of Advanced Topics in Types and Programming Languages [148]. Chapter 3 then extends this to the realm of graded types, giving an introduction to the syntax and features of Granule and developing a core graded calculus which will be extended throughout the rest of the thesis, along with a basic operational semantics in order to prove various key properties. The thesis then proceeds as follows.

First, we consider memory safety. Chapter 4 formalises the relationship between the two well-studied but rarely contrasted ideas of linearity and uniqueness (which aims to ensure that all values have at most a single reference), building on two distinct bodies of literature and showing that it is possible and advantageous to have both linear and unique types in the same type system. We study the guarantees of the resulting system and provide a practical implementation in the graded modal setting of the Granule language, adding a third kind of modality alongside the existing coeffect and effect modalities. We then demonstrate via a benchmark that our implementation benefits from expected efficiency gains enabled by adding uniqueness to a language that already has a linear basis. Chapter 5 develops this connection further, showing that just as graded type systems like those of Granule and Idris generalise linearity, Rust's ownership model arises as a graded generalisation of uniqueness. We combine fractional permissions with grading to give the first account of ownership and borrowing that smoothly integrates into a standard type system alongside linearity and grading, and extend Granule with these ideas.

Second, we consider communication safety. Chapter 6 presents the first formalisation of linear session types alongside a graded core calculus, using a similar operational model to the one applied in previous chapters in order to show that concurrency failures are ruled out and resolving problems with the previous instantiation of session types in Granule along the way. Chapter 7 then extends this system to allow for *non-linear* communication, introducing several additional primitives (and corresponding constructs in the semantics) which allow for various essential communication patterns to be captured that could not previously be described in a purely linear setting.

Finally, we consider the broader implications of our work. Chapter 8 discusses numerous potential avenues for further research, both in the areas of graded types for memory and communication safety but also including the possibility for a general framework which can capture global properties of programs at the type level. Chapter 9 offers a summary of the work presented in this thesis and a discussion of its consequences for programming languages in academia and industry.

1.3 PUBLICATIONS

The content of much of this thesis is drawn from existing publications. A full list of these papers is below; the declaration at the end of this thesis lists the primary work contributing to the content of each chapter.

[1] Jack Hughes, Danielle Marshall, James Wood, and Dominic Orchard. "Linear Exponentials as Graded Modal Types." In: *5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021)*. Rome (virtual), Italy, June 2021. URL: https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271465.

[2] Danielle Marshall and Dominic Orchard. "How to Take the Inverse of a Type." In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Ed. by Karim Ali and Jan Vitek. Vol. 222. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 5:1–5:27. ISBN: 978-3-95977-225-9. DOI: 10.4230/LIPIcs.ECOOP.2022.5.

[3] Danielle Marshall and Dominic Orchard. "Replicate, Reuse, Repeat: Capturing Non-Linear Communication via Session Types and Graded Modal Types." In: *Proceedings of PLACES 2022, Electronic Proceedings in Theoretical Computer Science* 356 (Mar. 2022), pp. 1–11. DOI: 10.4204/eptcs.356.1.

[4] Danielle Marshall and Dominic Orchard. "Graded Modal Types for Integrity and Confidentiality." In: *17th Workshop on Pro-*

grammming Languages and Analysis for Security (PLAS). 2023. DOI: 10.48550/arXiv.2309.04324.

[5] Danielle Marshall and Dominic Orchard. "Functional Ownership through Fractional Uniqueness." In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024). DOI: 10.1145/3649848.

[6] Danielle Marshall and Dominic Orchard. "Non-Linear Communication via Graded Modal Session Types." In: *Information and Computation* 301 (2024), p. 105234. ISSN: 0890-5401. DOI: 10.1016/j.ic.2024.105234.

[7] Danielle Marshall, Michael Vollmer, and Dominic Orchard. "Linearity and Uniqueness: An Entente Cordiale." In: *Programming Languages and Systems*. Ed. by Ilya Sergey. Cham: Springer International Publishing, 2022, pp. 346–375. ISBN: 978-3-030-99336-8. DOI: 10.1007/978-3-030-99336-8_13.

Part I

LINEAR TYPES & GRADED MODALITIES

2 A PIECE OF CAKE
BACKGROUND ON LINEARITY

ROADMAP The body of this chapter discusses the foundations of linear and graded types in detail. We briefly discuss some of the terminology used to describe type systems throughout the thesis in Section 2.1. We then begin in earnest by defining the linear λ-calculus with an exponential modality in Section 2.2 in preparation for generalising this to our core graded calculus in Section 3.3. We include some examples of basic Granule programs without grading, in order to better explain the core ideas while also offering a gentle introduction to the syntax of Granule.

2.1 TERMINOLOGY

Using the Curry-Howard correspondence, we can understand logics and type systems as being closely related; through the lens of Curry-Howard propositions in a logic correspond to types, while proofs of a proposition correspond to values of the proposition's corresponding type. We focus on the perspective of type systems in this thesis, as we are aiming for the target of leveraging theoretical knowledge for practical programmers, though we will sometimes wish to consider the respective logics in question.

The core languages and their extensions we describe are presented as typed calculi given by sets of *types*, *terms* (or programs of a given type) and *typing rules* that describe how a term relates to its type. We assume familiarity with the simply typed λ-calculus, which via Curry-Howard corresponds to the implicative fragment of intuitionistic logic, and build upon it as we explain linear and graded type systems. A reader lacking the prerequisite knowledge may wish to consult a textbook; one good example is *Types and Programming Languages* [111]. However, we will briefly recap some of the crucial terminology.

In the simply typed λ-calculus, a *judgment* defines the relation between a type and a term under a given context. Judgments have the form $\Gamma \vdash t : A$, stating that the term t can be assigned the type A under some context of *assumptions* Γ; an assumption is a variable name with an associated type, written $x : B$ (where name x has type B), and corresponds to a variable that may be in scope in a program.

A term is related to a type in some type system if we can derive such a judgment only through the application of the typing rules present in the given system. The application of these rules forms a tree structure that is sometimes referred to as a *typing derivation*.

$$\frac{\Gamma \vdash t : B}{\Gamma, x : A \vdash t : B}\ \textsc{Weakening}$$

$$\frac{\Gamma, x : A, y : A \vdash t : B}{\Gamma, x : A \vdash [x/y]t : B}\ \textsc{Contraction}$$

$$\frac{\Gamma_1, y : B, x : A, \Gamma_2 \vdash t : C}{\Gamma_1, x : A, y : B, \Gamma_2 \vdash t : C}\ \textsc{Exchange}$$

Figure 2.1: Substructural rules for weakening, contraction, and exchange

2.2 LINEARITY AND SUBSTRUCTURALITY

In the simply typed λ-calculus, the explicit typing rules are directed by syntax, while the *structural* rules (those that determine how the context of assumptions can be manipulated) are left implicit. The key structural properties we consider are *weakening*, *contraction*, and *exchange*.

Weakening allows assumptions that are not relevant to the typing derivation in question to be discarded. Contraction is dual to weakening, allowing for an assumption to be present multiple times in the context. Exchange allows assumptions to be arbitrarily reordered. Explicit rules describing these three structural properties are displayed in Figure 2.1.

Linear logic [50] is an example of a *substructural* logic, which means that it lacks some subset of the above structural properties. In particular, linear logic forbids weakening and contraction, while permitting exchange. In practice, this means that in order to construct a valid typing derivation, every assumption must be used *exactly* once; no hypotheses can be copied or discarded, meaning that a vast number of programs which would be typeable in the simply typed λ-calculus are not valid in a linear type system.

Note that permitting and excluding other combinations of the structural rules presented above therefore leads to other examples of substructural logics, which we briefly explain as they are frequently confused with one another [118]. *Relevance* or *relevant* logic forbids *only* weakening, but permits contraction and exchange, so each assumption must be used *at least* once. *Affine* logic, conversely, forbids only contraction while permitting weakening and exchange, so each assumption must be used at *most* once. *Ordered* (or *planar*) logic is the most restrictive, excluding all three of the above structural properties. Hence, each assumption must not only be used exactly once as in linear logic, but also in the correct order.

2.2.1 The Linear λ-Calculus

We now present a *linear* type system based closely upon the simply typed λ-calculus, but with the weakening and contraction properties excluded as described above; through Curry-Howard the linear λ-calculus we will describe corresponds to the implicative fragment of Girard's linear logic, much as the simply typed λ-calculus corresponds to the implicative fragment of intuitionistic logic.

The types of the linear λ-calculus (along with those drawn from a set of base types, elided for simplicity) are given by the grammar:

$$A, B ::= A \multimap B \mid !A \qquad \text{(types)}$$

As in the simply typed λ-calculus, the function type is the primary type former. Following Girard, we use $\multimap$ rather than $\to$ as a reminder that these functions are *linear*, and so cannot duplicate or discard their input; they consume a value of type A to produce a value of type B.

Following Girard and most linear type systems based on his work, we also include a ! (pronounced "bang") modality. This modality, which is formally known as the *exponential* modality, allows for non-linear usage of data to be expressed, as a value wrapped inside the exponential modality behaves as one might expect from a value in standard intuitionistic logic.

Typing judgments still have the form $\Gamma \vdash t : A$, where Γ ranges over contexts of assumptions constructed as follows:

$$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A] \qquad \text{(contexts)}$$

A context may therefore either be empty $\emptyset$, extended with some linear assumption $x : A$, or extended with some non-linear assumption $x : [A]$. The comma symbol denotes both disjoint context concatenation and context extension as described here. Since exchange is permitted in linear logic, we treat contexts as unordered, allowing their assumptions to be freely rearranged. This marking of assumptions in a context as linear or non-linear (see Terui [130]) is one way to guarantee substitution is admissible (avoiding, for example, issues pointed out by Wadler where substitution is not well-typed if care is not taken [144, 145], an issue noted also by Prawitz in 1965 in the context of S4 modal logic [114]).

The syntax of terms is identical to that of the simply typed λ-calculus but with terms for introducing and eliminating the ! modality:

$$t ::= x \mid \lambda x.t \mid t_1 t_2 \mid !t \mid \text{let } !t = t_1 \text{ in } t_2 \qquad \text{(terms)}$$

A term is either a linear variable usage x, an *abstraction* which binds a linear variable x in a term t, an application of a term t_1 to its argument t_2, a non-linear term $!t$, or a let binding eliminating a non-linear term ($\text{let } !t = t_1 \text{ in } t_2$).

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

Figure 2.2: Typing rules for the linear λ-calculus core

The rules that relate terms to their types in the linear λ-calculus core are presented in Figure 2.2. Linear variables must be typed in a singleton context (Var). The rest of the context must be marked as non-linear, denoted by $[\Gamma]$ which marks each assumption as non-linear.

Definition 1 (All non-linear assumptions)**.** *A context Γ is denoted as containing only non-linear assumptions by writing $[\Gamma]$ in the typing rules, where $[\emptyset]$ and $[\Gamma] \implies [\Gamma], x : [A]$.*

The abstraction rule (Abs) binds a linear variable which is used to type the premise. The application rule (App) combines the assumptions involved in both premises through the operation of *context addition* (Definition 2). Context addition combines contexts that have come from typing multiple subterms in a given rule. This operation, written $\Gamma_1 + \Gamma_2$, is undefined for contexts Γ_1 and Γ_2 that share some linear assumption. This requires the contexts to be *disjoint* with the exception of non-linear assumptions, which gives us the required substructural behaviour of forbidding contraction.

Definition 2 (Linear context addition)**.** *The partial operation $+$ on contexts is the union of two contexts as long as they are disjoint in their linear assumptions and any variables occurring in both contexts are both non-linear assumptions, i.e.*

$$\begin{aligned}
\Gamma + \emptyset &= \Gamma \\
\emptyset + \Gamma &= \Gamma \\
(\Gamma, x : A) + \Gamma' &= (\Gamma + \Gamma'), x : A \textit{ iff } x \notin |\Gamma'| \\
\Gamma + (\Gamma', x : A) &= (\Gamma + \Gamma'), x : A \textit{ iff } x \notin |\Gamma| \\
(\Gamma, x : [A]) + (\Gamma', x : [A]) &= (\Gamma + \Gamma'), x : [A]
\end{aligned}$$

The non-linear modality ! has introduction and elimination rules and a related dereliction rule as described in Figure 2.3.

The top left rule captures the idea that a computation t of type A can be used non-linearly, by 'promoting' it to $!A$ as long as all its inputs are also non-linear, denoted by $[\Gamma]$ in the context. The bottom rule eliminates a non-linear modality (a capability to use an A value non-linearly) by composing it with a variable x which is non-linear

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash\ !t : !A}\ !_I \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A] \vdash t : B}\ \textsc{der}$$

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ !x = t_1\ \mathsf{in}\ t_2 : B}\ !_E$$

Figure 2.3: Typing rules for the non-linear ! modality

in t_2. These rules are accompanied by the top right rule that says non-linear variables can be treated as linear variables.

We have now defined a simple substructural calculus, which we will build on in Section 3.3 by introducing a graded modality.

Example 1. An example of a program typeable in this calculus is the basic combinator for composing two functions one after the other:

$$\textsc{comp} : (A \multimap B) \multimap (B \multimap C) \multimap A \multimap C$$
$$\textsc{comp} = \lambda x.\lambda y.\lambda z.y(xz)$$

Here, the first line represents a type signature and the second line represents a function body as in a standard functional program (though we follow Granule in using a single colon for type declarations rather than a double colon as in Haskell), in the following sense:

$$\emptyset \vdash \lambda x.\lambda y.\lambda z.y(xz) : (A \multimap B) \multimap (B \multimap C) \multimap A \multimap C$$

This program satisfies the restrictions of linearity because each of the three arguments x, y, and z are used exactly once in the body of the function. In Granule, the above program may be written as follows:

```
comp : ∀ {a b c : Type} . (a → b) → (b → c) → a → c
comp x y z = y (x z)
```

There are two key differences to note here. First, since Granule has a linear base functions behave linearly by default, so we use the standard $\rightarrow$ syntax for function arrows since within the context of Granule this unambiguously denotes a linear function. Second, Granule code features explicit quantification over kind-annotated type variables, as can be seen in the type scheme annotating this definition, since Granule makes use of ML-style polymorphism in type declarations.

In contrast, the familiar K combinator from SKI combinatory logic cannot be typed in our linear λ-calculus. Consider the following attempt at a definition:

$$\textsc{k} : A \multimap B \multimap A$$
$$\textsc{k} = \lambda x.\lambda y.x \qquad \text{(ill typed)}$$

In other words:

$$\emptyset \nvdash \lambda x.\lambda y.x : A \multimap B \multimap A$$

Here, the argument y is not used in the body of the function, which violates the linearity restriction. The corresponding Granule program would be written as follows:

```
k : ∀ {a b : Type} . a → b → a
k x y = x -- ill typed
```

The Granule type checker will reject this program due to the aforementioned linearity violation, as one might hope. The resulting output is a *linearity error* stating that `Linear variable 'y' is never used.`

Granule includes the exponential modality, so to illustrate the idea the K combinator described above can be correctly typed in Granule as follows.

```
k : ∀ {a b : Type} . a → !b → a
k x !y = x
```

There are various interesting properties that can be proven regarding the above linear λ-calculus, including standard type safety results such as progress and preservation. We defer these for now, since they are all standard and well-trodden in the literature [142], instead proving them for the graded generalisation presented as our core calculus in Section 3.3. We will prove properties regarding the non-linear ! modality specifically in Chapter 4 alongside the extensions presented in that chapter, since this is the only other chapter that builds directly on the linear and non-linear core presented here rather than building upon the graded calculus of Chapter 3.

3 MAKING THE GRADE
A GRADED MODAL PRIMER

3.1 A BRIEF HISTORY OF GRADED MODAL TYPES

Graded type systems build upon linear types, which as described earlier treat data as a finite resource which must be consumed exactly once, with all instances of duplication or discarding disallowed [50]. Since Girard's original formulation of linear logic, there have been many instances of linear type systems in the literature [4, 142].

We have seen that linear type systems embedding the ! modality offer a binary view of data usage; either data is linear and is used exactly once, or it is non-linear and can be used freely.

Bounded linear logic, developed by Girard and collaborators shortly after linear logic itself [51], already begins to refine this binary view by replacing the singular exponential modality with a family of indexed modalities where the index gives an upper bound on how many times a piece of data may be used. For example, in a type system corresponding to bounded linear logic, $!_{\leq 3}A$ would be a type containing values of type A which may be used up to three times.

More recently, various related yet contrasting lines of work [2, 29, 32, 49, 94, 108, 153] have aimed to further generalise bounded linear logic. These systems typically allow for indices that, rather than only giving an upper bound on the number of times a piece of data is used, are instead drawn from an arbitrary pre-ordered semiring (or a related structure, such as a *skew* semiring [152]); different choices of semiring allow for different behavioural program properties to be tracked and verified statically. This can include the number of times a variable is reused but can also encompass other properties, such as privacy and confidentiality. Graded types are making their way into practical programming systems, and form the basis of not only Granule but also Idris 2 [27] and indeed Haskell's linear types extension [19].

In the realm of categorical semantics, these exponential modalities indexed by semiring elements are modelled by *graded exponential comonads* [29]. The original paper on Granule [103] defines the notion of *graded modal types*, which encompasses both said graded exponential comonads but also the dual notion of *graded monads* [45, 47, 105, 127], which generalise monads and can be used to precisely capture effectful behaviour. This thesis does not consider graded monads in detail; we focus on the graded comonads which extend and build on the substructural ideas we have described thus far. In general, we refer to systems incorporating such graded modalities as *graded* type systems,

though this terminology also includes type systems with pervasive indices that do not include modalities.

Returning to the example of the K combinator above, we can wrap the arguments of the function in graded modalities and annotate them with natural numbers describing the precise number of times that they are used, which in Granule is written as follows:

```
k : ∀ {a b : Type} . a [1] → b [0] → a
k [x] [y] = x
```

Note that we made a particular choice of semiring here, and there are numerous other possibilities for specifications that would have been valid for programs with the same operational behaviour; we will explore some of them later. Here the input x is annotated with the grade 1, which in the particular semiring of natural numbers used in Granule (which does not allow for approximation, a notion that will be explained later in the chapter that allows for more flexible usage) indicates that it *must* be used once.

Granule's underlying type system is linear, which means that values without an accompanying graded modality (such as the output value x here) are treated as linear; they must always be used exactly once, regardless of what flavours of grade are being used in the rest of the program. Graded modal types must themselves be introduced and eliminated explicitly through the use of the indexed modal $\Box_r$ operators (written `[r]` in Granule code). For example, the type $\Box_2 A \multimap B$ is a type of linear functions, but the argument is *graded* by 2. *Unboxing* the graded modality gives a value of type A which must be used precisely twice in the body of the function to produce the output of type B.

3.2 GRANULE AND GRADED MODALITIES

Granule's type system is based on the linear λ-calculus augmented with *graded modal types* [103] as well as polymorphic and indexed types (in the form of user-definable generalised algebraic data types, or GADTs). With linear typing as the basis, we cannot write functions that discard or duplicate their inputs as in a standard functional programming language. However, we can introduce non-linearity via graded modalities and use these to represent such behaviour, exemplified by the following Granule code:

```
drop : ∀ {a : Type} . a [0] → ()
drop [x] = ()

copy : ∀ {a : Type} . a [2] → (a, a)
copy [x] = (x, x)
```

The function arrow can be read as the type of linear functions (which consume their input exactly once; recall that these are generally notated $\multimap$, but in Granule functions are linear by default, so we simply

use $\rightarrow$), but `a [r]` is a graded modal type capturing the capability to use the value 'inside' in a non-linear way as described by `r`, i.e., `drop` uses the value 0 times and `copy` uses it 2 times. The pattern match `[x]` on the left-hand side of each function eliminates the graded modality, binding `x` in a non-linear way. The Haskell-style tuple syntax `(a, a)` used here represents a linear tensor product type, which we later notate using the $\otimes$ operator in our core calculus.

The central idea of graded modal types is to capture program structure via an indexed family of modalities where the indices have some algebraic structure which gives an abstract view of program structure. We focus on *semiring graded necessity* in this chapter (written $\Box_r A$ in mathematical notation but `A [r]` in Granule) which generalises linear logic's ! [50] and Bounded Linear Logic [51]. The semiring structure is also equipped with a preorder. The above example uses the natural number graded modality with equality as its preorder, thus providing an analysis which counts exactly how many times a value can be used.

Another useful graded modality has grades drawn from a semiring of *intervals* which allows us to give bounds on how a value might be used. To demonstrate, we define the `fromMaybe` function which allows a value that may or may not exist to be retrieved from a `Maybe` type.

```
data Maybe a = Just a | Nothing

fromMaybe : ∀ {a : Type} . a [0..1] → Maybe a → a
fromMaybe [_] (Just x) = x;
fromMaybe [d] Nothing  = d
```

Note that without the graded modality, this function would be ill-typed in Granule, since values are linear by default and one of the cases requires discarding the default value (given by the first parameter). By giving this parameter a type of `a [0..1]`, we specify that it can be used *either* `0` times or `1` time (in other words, this value has *affine* behaviour rather than linear). There is only one total function in Granule which inhabits the type we give to `fromMaybe` here–linearity forbids defining an instance which always returns the default value.

In order to use `fromMaybe` we need to 'promote' its first input to be a graded modal value, which is written by wrapping a value in brackets, e.g., `fromMaybe [42]`. Promotion propagates any requirements implied by the graded modality to the free variables. For example, the following takes an input and shares the capabilities implied by its grade to two uses of `fromMaybe`:

```
fromMaybeIntPlus : Int [0..2] → Maybe Int → Maybe Int → Int
fromMaybeIntPlus [d] x y = fromMaybe [d] x + fromMaybe [d] y
```

The `0..1` usage implied by the first `fromMaybe [d]` and `0..1` usage by the second are added together to get the requirement that the incoming integer `d` is graded as `0..2`.

Lastly, Granule includes *indexed* types which offer a lightweight form of dependency, allowing for type-level access to information

about data. For example, length-indexed vectors can be defined and used as follows:

```
data Vec (n : Nat) (a : Type) where
  Nil  : Vec 0 a;
  Cons : a → Vec n a → Vec (n+1) a

append : ∀ {a : Type, n m : Nat} . Vec n a → Vec m a → Vec (n+m) a
append Nil ys         = ys;
append (Cons x xs) ys = Cons x (append xs ys)
```

Indexed types allow us to ensure at the type-level that when we append two vectors, the length of the output is equal to the sum of the lengths of the two inputs. Again, thanks to linearity, we gain further assurances from our type signatures: here we can guarantee that since we must use every element of the input vectors, these must also appear in the output, and so no information is being lost along the way.

Indexed natural numbers can be similarly defined to vectors, and are used later to provide lightweight type-dependence on natural number inputs:

```
data N (n : Nat) where
    Z : N 0;
    S : ∀ {n : Nat} . N n → N (n+1)
```

This definition gives a way to capture a runtime witness for a type level `Nat`.

Granule allows type indices to be used also as grades. We can define for example the functorial action on vectors which links the size of the vector with the usage of the function to be applied to each element:

```
vmap : ∀ {a b : Type, n : Nat} . (a → b) [n] → Vec n a → Vec n b
vmap [_] Nil         = Nil;
vmap [f] (Cons x xs) = Cons (f x) (vmap [f] xs)
```

In Granule, predicates on types and grades can be represented like Haskell's type class constraints with type signatures of the form:

```
functionName : ∀{typeVariables} . {constraints} ⇒ type
```

For example, we can define a much more general version of `copy` as:

```
copy : ∀ {a : Type, s : Semiring, r : s}
     . {((1 + 1) : s) ⩽ r} ⇒ a [r] → (a, a)
copy [x] = (x, x)
```

This is polymorphic on the semiring `s`, as well as including a constraint as a predicate that the grade `r` is approximated by $1 + 1$ in this particular semiring. (Note that $1 + 1$ cannot be simply replaced by 2 here, since the function is polymorphic and so we cannot assume we are working with the semiring of natural numbers: 1 represents the multiplicative identity of the semiring s, and so the result of $1 + 1$ is dependent upon the choice of semiring.) Types appearing in this constraint position have kind `Predicate`. We later introduce several type functions, returning types of kind `Predicate`, in order to restrict behaviour in various settings.

3.3 A CORE GRADED MODAL CALCULUS

Before we can define the graded type system which we will go on to extend throughout the main body of this thesis, we need to establish an understanding of the foundations of the particular approach to graded types that has been taken by Granule and its core calculus.

The origins of graded types lie in Girard's original work on linear logic [50], which has spawned a plethora of work on type systems which incorporate additional information on the behaviour of a program into the types. Some graded type systems involve every function arrow in a program being annotated with a grade describing the usage of the input value (sometimes referred to as "graded base" systems), while some start from a basis of linear types and then extend this with a *graded modality* (sometimes referred to as "linear base" systems), where values are linear by default but the usage of those wrapped inside a modality is described by the grade annotating said modality.

It has recently been shown that each of these two contrasting styles of graded type system can be translated into one another, preserving both types and grades, in such a way that the operational behaviour of the program remains the same [78]. Hence, we will not linger too much on the distinction here in this thesis; we focus on the *linear base* approach, since this is the one used in Granule's core.

We now recap Granule's core calculus which incorporates graded modal types into a linear type theory [103]; we later extend this calculus with our functional models of various kinds of resource in subsequent chapters. This section covers the syntax and static semantics (type system), whilst Section 3.4 gives an operational semantics that will later be extended to incorporate the various extensions discussed throughout this work.

The calculus extends the linear λ-calculus with multiplicative products and unit, and a *semiring-graded necessity modality* $\Box_r A$ where r is an element of a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ which includes a requirement that $*$ and $+$ must be monotonic with respect to the ordering $\sqsubseteq$. This calculus gives a simplified monomorphic subset of Granule [103], and closely resembles other graded systems from the literature [2, 10, 21, 29, 32, 47, 94, 95, 108]. Our extensions in later chapters could be made compatible with such systems, as the differences between these calculi do not interfere with the key ideas here.

3.3.1 Syntax

Our syntax consists of the linear λ-calculus with multiplicative products and unit (first line of syntax below), and graded modal terms (second line).

$$t ::= x \mid \lambda x.t \mid t_1\, t_2 \mid (t_1, t_2) \mid \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 \mid () \mid \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2$$
$$\mid [t] \mid \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \qquad \text{(terms)}$$

Following the syntax of variables, we group terms above into pairs of introduction and elimination forms, for functions, products, units, and the graded modality respectively. The meaning of these terms is explained in the next subsection with reference to their typing.

3.3.2 Type System

Typing judgments have the form $\Gamma \vdash t : A$, assigning type A to term t under context Γ. Types are:

$$A, B ::= A \multimap B \mid A \otimes B \mid \mathsf{unit} \mid \Box_r A \qquad \text{(types)}$$

Hence, our type syntax comprises linear function types $A \multimap B$, linear multiplicative products $A \otimes B$, a linear multiplicative unit (unsurprisingly written unit), and the graded modality $\Box_r A$ where $r \in \mathcal{R}$.

Contexts Γ contain both linear assumptions $x : A$ and graded assumptions $x : [A]_r$ which have originated from a graded modality:

$$\Gamma ::= \varnothing \mid \Gamma, x : A \mid \Gamma, x : [A]_r \qquad \text{(contexts)}$$

Typing of the λ-calculus fragment is given by the rules in Figure 3.1.

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

Figure 3.1: Typing rules for the linear λ-calculus

The VAR, ABS, and APP rules are the standard rules of the linear λ-calculus, augmented with a notion of *contraction* captured by the $+$ operation on contexts coming from multiple sub-terms. This operation is defined only when contexts are disjoint with respect to linear assumptions, and on overlapping graded assumptions we add their grades using the addition operation provided by the semiring,

e.g. $(\Gamma_1, x : [A]_r) + (\Gamma_2, x : [A]_s) = (\Gamma_1 + \Gamma_2), x : [A]_{r+s}$. More explicitly, context addition is declaratively specified as follows:

$$\begin{aligned}
\varnothing + \Gamma &= \Gamma \\
\Gamma + \varnothing &= \Gamma \\
(\Gamma, x : A) + \Gamma' &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma'| \\
\Gamma + (\Gamma', x : A) &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma| \\
(\Gamma, x : [A]_r) + (\Gamma', x : [A]_s) &= (\Gamma + \Gamma'), x : [A]_{(r+s)}
\end{aligned}$$

(context addition)

This is a declarative rather than algorithmic specification of $+$ as graded assumptions of the same variable may appear in different positions within the two contexts—for example, when type checking the program `((x, y), (y, x))`. The VAR rule also embeds the notion of *weakening*, allowing a context of variables graded by 0, using the partial operation of *scalar multiplication of a context*:

$$r \cdot \varnothing = \varnothing \qquad r \cdot (\Gamma, x : [A]_s) = (r \cdot \Gamma), x : [A]_{r*s}$$

(context multiplication)

This operation scales graded assumptions by r, but is undefined if Γ contains linear assumptions.
The rules involving graded modalities are presented in Figure 3.2.

$$\frac{\Gamma \vdash t : A}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

Figure 3.2: Typing rules for the graded modality

The PR rule (*promotion*) introduces a graded modality with grade r, implying that the result of t can be used in an 'r-like' way and thus all of the dependencies of t must be scaled by r to propagate usage to the dependencies, none of which are allowed to be linear due to the partiality of context multiplication. In later chapters, promotion will also require an explicit restriction that the value to be promoted must not be a *resource allocator*, designed to avoid problems that otherwise arise when promoting resources with certain behaviours in a call-by-value setting; we will discuss this in detail when we encounter the problem for the first time, in Chapter 5.

The ELIM rule eliminates a graded modality, capturing the idea that a requirement for x to be used in an 'r-like' way in t_2 can be matched with the capability of t_1 described by its graded modal type.

The DER rule (*dereliction*) connects linear typing to graded typing, stating that a requirement for a linear assumption is satisfied by an assumption graded by 1. The APPROX rule converts a grade r to s if s *approximates* r according to the semiring's pre-order $\sqsubseteq$. Some useful and/or interesting examples of semirings are described below.

Example 2. [Natural numbers] One possible choice of semiring, which we have seen already in the basic examples presented thus far, is over the natural numbers $(\mathbb{N}, *, +, 0, 1, \equiv)$ with *discrete ordering* $\equiv$ such that there is no approximation and we track the exact number of times a term has been used. We could instead use the standard $\leq$ ordering on numbers which would permit approximation, allowing for an *upper bound* on a term's usage, and various other works on graded types use this ordering for their natural number semiring. Another useful semiring is *intervals* of natural numbers where a grade `0..1` represents values which can be used *either* zero or one times, capturing the notion of an *affine* value. In fact, an interval semiring can be constructed by choosing two endpoints from any other semiring; we will only use intervals of natural numbers in this thesis, but details of this construction are given in the original Granule paper [103].

Example 3. [None-One-Tons [94]] The semiring over $\{0, 1, \omega\}$ with $0 \leq \omega$ and $1 \leq \omega$, where $+$ and $*$ are saturating at ω, can be used to distinguish between linear and non-linear uses: assigning $r \equiv 1$ to linear usage, $r \equiv 0$ to non-usage (when a resource is discarded), and $r = \omega$ to arbitrary usage.

Example 4. [Security levels] Information flow control properties can be tracked by instantiating the semiring with a lattice of security levels [47], e.g., with $(\{\mathsf{Lo} \leq \mathsf{Hi}\}, \mathsf{Lo}, \wedge, \mathsf{Hi}, \vee)$ where Hi-graded inputs are treated as irrelevant: we cannot depend on any high-security inputs when building a low-security graded output $\Box_{\mathsf{Lo}} A$. This thesis does not focus on security, so we will use this simple case as an illustrative example, but it is certainly possible to construct more complex lattices with more than two possible security levels [79].

Example 5. [Sensitivity] The semiring of real numbers with the standard ordering $(\mathbb{R}, 1, *, 0, +, \leq)$ can be leveraged to capture a notion of numerical sensitivity in programs/logic [39, 46], where a program is k-sensitive (for $k \in \mathbb{R}$) in a variable if a change ϵ in its inputs to x produces at most a change of $k\epsilon$ in the output of the program. This instantiation of the system tracks sensitivities as grades where additional dependent-type-based mechanisms are needed to lift program values into the types, e.g., $\mathsf{scale} : (k : \mathbb{R}) \to \Box_k \mathbb{R} \to \mathbb{R}$.

Finally, Figure 3.3 displays the rules for introducing and eliminating tensor products and the multiplicative unit, which are standard,

though it is important to remember that products are *linear* and so it is not possible to freely discard either side: both elements must be used when consuming a product. The other connectives from linear logic (such as the additive disjunction $\oplus$ and the additive conjunction &) can be incorporated in an equally standard fashion; we do not focus on them here in order to simplify our proofs, as the way they are handled would be very similar to the way we deal with the tensor product rules in all of the forthcoming proof cases.

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B} \otimes_I$$

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C} \otimes_E$$

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : B}\ 1_E$$

Figure 3.3: Typing rules for linear products and unit

Note that in the presence of products, we are unable to exactly represent the exponential modality ! from Chapter 2 by naïvely using the ω grade from the 'none-one-tons' semiring or any similar grades that allow for unrestricted usage, as one might expect at first glance. This is because no matter which grade we choose from whichever semiring, we are able to "push" the grade into the tensor product, allowing derivation of $\Box_r(A \otimes B) \to (\Box_r A \otimes \Box_r B)$, as in the following simplified Granule code.

```
push : ∀ {a b : Type, s : Semiring, r : s}
       . (a, b) [r] → (a [r], b [r])
push [(x, y)] = ([x], [y])
```

Granule ✗

However, in linear logic it is not possible to derive $!(A \otimes B) \multimap !A \otimes !B$; indeed, the following Granule code is ill-typed.

```
push' : ∀ {a b : Type} . !(a, b) → (!a, !b)
push' !(x, y) = (!x, !y)
```

Granule ✗

To acquire some intuition for this, we can think of a type `!a` as allowing for the consumer of its value to choose how many times they use that value. This means that if we have a pair of values with type `(!a, !b)` then we can make separate, different choices for how to use the components of that product; for example, we could choose to use the value of type `!a` twice and the value of type `!b` three times.

Conversely, if we are given a value of type `!(a, b)` then we are only able to choose *once* how many times to use the *entire* product, so both components must be used the same number of times. Therefore, this type gives the consumer strictly less freedom, and so we cannot in general construct a value of the former type from a value of the latter.

Many graded type systems share the quality of allowing grading to be pushed through products, so reconciling these with linear logic requires some additional structure on the semiring [63]. In practice, Granule's version of the 'none-one-tons' semiring does incorporate this additional structure, which prevents the push behaviour and allows ! to be represented in Granule using the ω grade. The polymorphic `push` function above is not directly typeable in Granule proper, without additionally limiting the input grade to come from a semiring that allows this behaviour.

We do not linger on this any further here, as the distinction will not be particularly important for our purposes. In Chapter 4 we limit the focus to linearity and non-linearity by using the ! modality as in Chapter 2, and then from Chapter 5 onwards we incorporate full grading as seen in the present chapter.

Example 6. The following gives an example derivation assuming the natural number semiring:

$$\dfrac{\dfrac{\dfrac{}{x : A \otimes A \vdash x : A \otimes A}\,\textsc{var} \quad \dfrac{}{y : [A]_1 \vdash y : A}\,\textsc{var}'}{y : [A]_1, x : A \otimes A \vdash (x, y) : (A \otimes A) \otimes A}\,\otimes_I}{y : [A]_1 \vdash \lambda x.(x, y) : A \otimes A \multimap (A \otimes A) \otimes A}\,\textsc{abs} \quad \dfrac{\dfrac{}{\varnothing \vdash v : A} \quad \dfrac{}{y : [A]_1 \vdash y : A}\,\textsc{var}'}{y : [A]_1 \vdash (v, y) : A \otimes A}\,\otimes_I}{y : [A]_2 \vdash (\lambda x.(x, y))\,(v, y) : (A \otimes A) \otimes A}\,\textsc{app}$$

where (VAR') is a synonym for (VAR) followed by the (DER) rule.

As a reminder of Granule syntax, the following captures the same idea as this example:

```
exampleNat : ∀ {a : Type} . a → a [2] → ((a, a), a)
exampleNat v yb = let [y] = yb in (λx → (x, y)) (v, y)
```
Granule

Recall that $\Box_r A$ is instead written postfix as `A [r]`. In Granule, it is also possible to fold elimination into pattern matching: we can 'unbox' (eliminate) a graded modality to provide a graded variable to the body of the function (the analogue to t_2 in the elimination rule). See the below code, which is equivalent:

```
exampleNat' : ∀ {a : Type} . a → a [2] → ((a, a), a)
exampleNat' v [y] = (λx → (x, y)) (v, y)
```
Granule

Example 7. The following gives an example derivation assuming the security level semiring:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}\,\textsc{var}}{x : [A]_{\mathsf{Lo}} \vdash x : A}\,\textsc{der}}{x : [A]_{\mathsf{Hi}} \vdash [x] : \Box_{\mathsf{Hi}} A}\,\textsc{pr}}{x : [A]_{\mathsf{Lo}} \vdash [x] : \Box_{\mathsf{Hi}} A}\,\textsc{approx}$$

We can type a function which maps public (Lo) values to private (Hi) values, using approximation as above. A function mapping from private to public values is derivable, but cannot use its input, as required for a non-interference property to hold [2, 79].

NOTE ON LINEAR HASKELL As mentioned at the start of this section, we aim for the extensions we develop throughout this work to be compatible with other graded type systems after some adaptation. One particular system of interest is the calculus underlying the recent extension which introduces linear types to Haskell [19]. This extension uses a graded type system below the surface in order to implement linearity; all function types are given a 'multiplicity' annotation r written `(a %r -> b)` akin to a type $\Box_r A \multimap B$ here, where the annotation can be either `'One` or `'Many` representing either linear or unrestricted usage respectively. As described at the start of this chapter, work on formalising the connection between systems like this where all function types come with a grade (called 'graded base') and systems such as Granule's where values are linear by default and graded values are wrapped inside a modality (called 'linear base') is ongoing [137].

3.4 OPERATIONAL SEMANTICS

We define an operational model for our core calculus to make the meaning of linearity and grading more concrete, and to prove that our type system enforces the standard desired properties.

$$\dfrac{}{(\lambda x.t_2)\, t_1 \rightsquigarrow t_2[t_1/x]}\;\beta_{\mathsf{L}} \qquad \dfrac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_2}\;\textsc{congApp}_{\mathsf{L}}$$

Figure 3.4: Reduction rules for the linear λ-calculus (call-by-name)

The rules in Figure 3.4 are the standard call-by-name reduction rules for the linear λ-calculus. Beta reductions are enacted through substituting the argument into the body of the function, while we can reduce on the left-hand side of any application. Later chapters will

demonstrate how to adjust this semantics to behave in a call-by-value manner, when this is required.

$$\frac{}{\mathbf{let}\,[x] = [t_1]\,\mathbf{in}\,t_2 \rightsquigarrow t_2[t_1/x]}\ \Box\beta_{\mathrm{L}}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \rightsquigarrow \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2}\ \textsc{congLet}_{\mathrm{L}}$$

Figure 3.5: Reduction rules for the graded modality (call-by-name)

The rules in Figure 3.5 are perhaps the most interesting, as these show the rules for graded modal introduction (called *promotion*) and elimination, which unwraps the box modality using a let binding. Note that there is no rule for evaluating *inside* of a graded modal introduction, since the semantics is call-by-name.

$$\frac{}{\mathbf{let}\,(x,y) = t_1 * t_2\,\mathbf{in}\,t_3 \rightsquigarrow t_3[t_2/y][t_1/x]}\ \otimes\beta_{\mathrm{L}}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\,(x,y) = t_1\,\mathbf{in}\,t_2 \rightsquigarrow \mathbf{let}\,(x,y) = t_1'\,\mathbf{in}\,t_2}\ \textsc{congProd}_{\mathrm{L}}$$

$$\frac{}{\mathbf{let}\,() = ()\,\mathbf{in}\,t_2 \rightsquigarrow t_2}\ \textsc{Unit}\beta_{\mathrm{L}}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 \rightsquigarrow \mathbf{let}\,() = t_1'\,\mathbf{in}\,t_2}\ \textsc{congUnit}_{\mathrm{L}}$$

Figure 3.6: Reduction rules for linear products and unit (call-by-name)

Figure 3.6 shows how to evaluate introduction and elimination for linear products and unit, which follow a similar scheme to the graded modal introduction and elimination rules. The only exception is that unit elimination is trivial and so does not require any substitution.

3.4.1 Metatheory

We now prove some of the standard properties of linear and graded calculi for the core system we have described, such that we can extend the proofs with each new addition in later chapters. Proofs of all the statements that follow are provided in Appendix a. We first establish some key results showing the admissibility of weakening and substitution, which are leveraged in later proofs:

Lemma 1 (Admissibility of weakening)**.**

$$\Gamma \vdash t : A \implies (0 \cdot \Gamma'), \Gamma \vdash t : A$$

Lemma 2 (Linear substitution)**.** *Given* $\Gamma' \vdash t' : S$ *and* $\Gamma, y : S \vdash t : T$ *then* $\Gamma' + \Gamma \vdash t[t'/y] : T$*.*

Lemma 3 (Graded substitution)**.** *Given* $[\Gamma'] \vdash t' : S$ *and* $\Gamma, y : [S]_r \vdash t : T$ *(where* $[\Gamma']$ *represents a context* Γ' *containing only graded assumptions) then* $\Gamma + r \cdot \Gamma' \vdash t[t'/y] : T$*.*

We define a notion of *value*, with a value being a term that is irreducible under our particular choice of semantics.

$$v ::= \lambda x.t \mid [t] \mid (t_1, t_2) \mid () \qquad \text{(value terms sub-grammar)}$$

We then consider the two key properties for syntactic type safety: progress and preservation.

Lemma 4 (Value lemma)**.** *Given* $\Gamma \vdash v : A$ *then depending on the type, the shape of* v *can be inferred:*

- $A = A' \to B$ *then* $v = \lambda x.t$
- $A = \Box_r A'$ *then* $v = [t]$
- $A = A' \otimes B'$ *then* $v = (t_1, t_2)$
- $A = 1$ *then* $v = ()$

Theorem 1 (Progress)**.** *Values in our model are given above.*

Given $\Gamma \vdash t : A$*, then* t *is either some value* v*, or there exists a term* t' *such that* $t \rightsquigarrow t'$*.*

Theorem 2 (Type preservation)**.** *Given* $\Gamma \vdash t : A$ *and a single-step reduction* $t \rightsquigarrow t'$*, then we can construct the typing derivation* $\Gamma \vdash t' : A$*.*

Note that the preservation theorem does not examine the consistency of grades; this requires an extended property, known as *conservation*. We will examine the relative feasibility of proving conservation for call-by-name and call-by-value semantics in Chapters 4 and 5 respectively.

Part II

UNIQUENESS TYPES & GRADED RESOURCES

4 ONE OF A KIND
UNIQUENESS AND SHARING

4.1 INTRODUCTION

Linear types [50, 142] and uniqueness types [15, 126] are two influential and long-standing flavours of substructural type system. As these approaches have developed, it has become clear in the community (both in folklore and the literature) that these are closely related ideas. For example, the chapter on substructurality in *Advanced Topics in Types and Programming Languages* [148] describes uniqueness types as "a variant of linear types". This framing is supported by various works which make reference to "a form of linearity (called uniqueness)" [97] or other such statements of equality or similarity [106].

But reading a different set of papers gives a contrasting impression that linearity and uniqueness are not the same but in some sense *dual* to one another, and with different behaviour for some applications. Recent work on linear types for Haskell [19] describes the two concepts as being "at their core, dual" and later having a "weak duality". The impression that these two approaches behave differently is backed up by much of the theoretical work on uniqueness types. One paper states that "although both linear logic and uniqueness typing are substructural logics, there are important differences" [141], closely followed by a tantalising hint that "some systems based on linear logic are much closer to uniqueness typing than to linear logic".

It is clear, at least, that both linear types and uniqueness types are indeed *substructural* type systems: they both restrict structural rules (in particular, contraction and weakening) of type systems that are the Curry-Howard counterparts to regular intuitionistic logic. As mentioned in Chapter 1, this captures the well-known maxim that "not all things in life are free" [146]; many kinds of data behave *resourcefully*, and are subject to constraints on their usage.

Thanks to these clear benefits, notions of substructurality are slowly but surely making their way into the programming ecosystem, with languages such as Haskell [19], Idris [27], Clean [126], Rust [69], ATS [156], and Granule [103] all having type systems that behave substructurally in some way. What is not clear, however, is what exactly the relationship is between these varying systems; for instance, it is not obvious how to relate linearity and uniqueness. Linear types, though they themselves come in various forms, are most often based on the linear logic of Girard [50]; as we saw in Chapter 2, they generally treat values as resources which must be used exactly once and never again. On the other hand, uniqueness types are named as such because they

aim to ensure that values are guaranteed to have at most one reference to them [112, 124, 126, 128, 140, 141], with a view towards allowing them to be safely updated in-place. Do these two requirements always coincide, or are there cases where they diverge?

In this chapter, we resolve this long-standing confusion, building on two distinct bodies of literature to develop an accurate understanding of the contexts in which linear and uniqueness types behave the same or behave differently, and their relative strengths and weaknesses. Our primary contributions are as follows:

- In Section 4.2, we discuss the contrasting understandings of the relationship between linearity and uniqueness, and draw together aspects of these viewpoints to intuitively describe the link between these concepts.
- In Section 4.3, we formalise these notions by developing a unified calculus and type system that incorporates linear, unique, and Cartesian (unrestricted or non-unique/non-linear) types all at once, building on the core calculus described in Chapter 3. We extend our operational model with a heap semantics which allows us to prove various key operational guarantees for both linearity and uniqueness.
- In Section 4.4, as a proof of concept, we implement uniqueness types into Granule which already has a linear basis, introducing a third flavour of modality alongside the graded comonadic (coeffectful) and graded monadic (effectful) modalities already present in the language. The implementation enables the classic primary use of uniqueness: access to safe in-place update in a functional language without working inside a monad.
- In Section 4.4.2, we confirm the performance benefits of uniqueness types by benchmarking the performance of arrays which allow for in-place update. We generate impure Haskell code from our Granule implementation in order to demonstrate that further efficiency can be gained via adding uniqueness types even when your language is already linear at its core.

We defer discussion of related work to Chapter 5, which will relate the concepts explored in this chapter to ideas in Rust and extend this chapter's development to allow for more flexibility when managing references. Future work will be explored in Chapter 8. Various additional details are collected in Appendix b, including proofs and collected reduction rules for the operational semantics.

4.2 KEY IDEAS

It is clear that linear and uniqueness types both involve restricting the substructural rules of intuitionistic logic, but what remains unclear is the exact relationship between the two concepts. This section discusses two widespread understandings of their relationship, both of which are accurate in some respects but fail to capture some key similarities and differences. We then combine aspects of both viewpoints to systematically relate linearity and uniqueness.

4.2.1 Are Linearity and Uniqueness the Same?

Perhaps the most well-known substructural types are *linear* types, which have been studied for decades in the literature [142, 148] as the Curry-Howard counterpart of linear logic [50]. Chapter 1 explained how linearity is steadily making its way into the mainstream via extensions to languages like Haskell [19]. Examples of linearity in this chapter will focus on the functional language Granule [103] (whose syntax resembles Haskell), since values in Granule are linear by default making the examples less complex, and also because Granule will be the foundation upon which we build our unified calculus.

As we explored in Chapter 2, linearity is a claim about the consumption of a resource: a linear type is a contract, which says that we must consume a value that we are given exactly once. Recall the following classic example (mentioned in Chapter 1) of a function which cannot be represented using linear types, assuming an interface where `eat : Cake → Happy` and `have : Cake → Cake`:

```
impossible : Cake → (Happy, Cake)
impossible cake = (eat cake, have cake)
```

Granule ✗

The above function is ill-typed (since Granule's function type → is linear) and the Granule compiler will brand it with a linearity error; this is because the value of type `Cake` passed into the function is a linear resource, and the body of the function requires us to duplicate it (via contraction), which is forbidden. This is why linear types remind us of the familiar aphorism: *you can't have your cake and eat it too*.

Uniqueness types, on the other hand, are aimed at ensuring that values have only a single reference to them, which is a useful property for ensuring the safety of updating data in-place. But is this uniqueness restriction really so different from the constraints of linearity?

One of the most familiar languages featuring uniqueness types is Clean [126], which uses uniqueness for mutable state and input/output, in contrast to languages such as Haskell which use monads for similar purposes. We shall use Clean for our uniqueness examples

for the moment, before we introduce our own implementation of uniqueness in Section 4.4. Consider the following in Clean:

```
impossible :: *Coffee -> (*Awake, *Coffee)
impossible coffee = (drink coffee, keep coffee)
```

Clean ✗

We use coffee instead of cake to distinguish unique values from the linear values of the Granule example, but notice this function has exactly the same structure as the previous example. The operator `*` denotes a unique type, since unrestricted values are the default in Clean. When presented with this function Clean gives a uniqueness error; the argument of type `*Coffee` is duplicated, and so we can no longer guarantee there is only one reference to it upon exiting the function. Think of a `*Coffee` as having been freshly poured; we cannot continue acting as though it is fresh once some of it has been drunk!

So far, it seems that the concepts of linearity and uniqueness are very similar after all, as is often claimed. However, neither of these examples uses *unrestricted* values; we only see values that are linearly typed or uniquely typed. In fact, in a setting where all values must be linear, we can also guarantee that every value is unique, and vice versa! Intuitively, if it is never possible to duplicate a value, then it will never be possible for said value to have multiple references [141]. It is when we also have the ability for unrestricted use (non-linear/non-unique) that differences between linearity and uniqueness begin to arise, as we will soon see.

Much of the classic literature on linear types makes mention of the idea that linearity can be used for tracking whether a value has only one reference, though we know by now that this more accurately describes *uniqueness*; indeed, one of the oldest such papers by Wadler, which has been (rightly) hugely influential, states that "values of linear type have exactly one reference to them, and so require no garbage collection" [142, p.2]. However, systems akin to the one discussed by Wadler [11, 55, 125] crucially separate values into two completely distinct and (mostly) disconnected linear and non-linear worlds. In this context, a linear value can never have been duplicated previously and thus must also obey the conditions required for uniqueness. Therefore, it is correct to say that a value of linear type has exactly one reference in such a system.

This issue is further discussed in a later article by Wadler [143], though uniqueness types had yet to be invented and so the concept is never referred to by this name. Linear types based on linear logic are defined in Section 3 of said article, for which linearity behaves as we understand it: a value having linear type guarantees that it will not be duplicated or discarded, but the notion of *dereliction* allows a non-linear variable to be used linearly going forwards [50]. As Wadler states, "dereliction means we cannot guarantee *a priori* that a variable of linear type has exactly one pointer to it" [143, p.7], and so we

cannot guarantee uniqueness of reference in a system based upon linear logic. In Section 7, Wadler goes on to define *steadfast* types, where dereliction and promotion are again restricted to recover the uniqueness guarantee in addition to the linearity restriction[1].

However, never being able to duplicate or discard any value is an overly restrictive view of data, preventing many valid uses of various kinds of information, and so languages with linear types therefore generally do provide a mechanism for non-linearity rather than working in the 'steadfast' style. In Granule, we can use the ! modality to rewrite the previous example into one that is well-typed:

```
possible : !Cake → (Happy, Cake)
possible lots = let !cake = lots in (eat cake, have cake)
```
Granule

We can think of `!Cake` values as representing an infinite amount of cake, which is made available once we eliminate the modality (via the `let`) to get an unrestricted (non-linear) variable `cake`. The functions `eat` and `have` are linear functions, so each application in isolation views `cake` linearly, by an implicit use of dereliction in the type system. Crucially, from an unrestricted value we can produce a linear value, so we can impose the restriction of linearity whenever we like. However it is not possible to produce an unrestricted value from a linear one. This restriction means that linear types are useful for representing resources such as file handles, as in the following example:

```
twoChars : (Char, Char) <IO>
twoChars = let                              -- do-notation like syntax
    h       ← openHandle ReadMode "someFile";
    (h, c1) ← readChar h;
    (h, c2) ← readChar h;
    ()      ← closeHandle h
    in pure (c1, c2)
```
Granule

Here, we open a file handle, read two characters from it, and then close it. The `<IO>` modality here can in essence be thought of as equivalent to the IO monad in Haskell, though in Granule this is in actuality a *graded* effect that allows for more precision [103]. The linearity of the handle ensures that once we have created it, we must close it properly, and also that we cannot duplicate it along the way. But linearity is less useful in other circumstances. As an example, consider the case of mutable arrays. Discarding an array will not cause any problems[2],

1 The concept of steadfastness coincides with the notion of "necessarily unique" used in languages such as Clean, where a necessarily unique value is one that is unique and also can never be made non-unique [126].

2 One may worry that discarding an array could cause space leaks, but this can be tempered via garbage collection. If a non-linear value will no longer be used we know statically that it can be garbage collected, and thus it is harmless to reuse the space occupied by this value going forwards. This will allow us to update unique objects such as arrays destructively without being concerned about referential transparency.

so a linear array would be too restrictive and not allow for some valid use cases; affine types allow discarding behaviour by adding back in weakening [134]. But in order to be able to mutate an array we need to be able to guarantee that no other references to it exist, and in this sense even linearity is not necessarily strong enough.

Depending on the particular choice of type system and operational semantics, any linear value could have previously been a non-linear one that was duplicated any number of times before being specialised (via dereliction) to a linear type. For representing mutable arrays, we are better served by considering uniqueness types.

Uniqueness behaves differently to linearity in the context of a system with the ability to describe unrestricted values. If we have an unrestricted value, we certainly cannot produce a unique one from it which would violate the guarantee of uniqueness; we cannot claim that a value has only one reference to it when it could have been duplicated and manipulated elsewhere. But conversely, if we have a unique value, there is no harm in dropping this guarantee and producing an unrestricted value; a non-unique value does not need to make any promises about how many references may exist. In Clean we can write:

```
possible :: *Coffee -> (Awake, Coffee)
possible coffee = (drink coffee, keep coffee)
```

Clean

Here, we require that the input is unique (it has type `*Coffee`), so for the function to be well-typed we can no longer claim this value is unique once it reaches the output, as it has been duplicated along the way (and it now must have type `Coffee`). Note that in Clean no modality means that a value is unrestricted, unlike in Granule where types with no modality mean that their values are linear by default. The information here is flowing in the opposite direction than for linearity; the `possible` function in Clean would be ill-typed if we replaced unique values with linear ones, and vice versa for the earlier Granule example. This directionality allows us to represent mutable arrays much more easily with uniqueness. For example, the following destructively fills a real-valued array:

```
fill :: *{Real} Int -> *{Real}
fill a1 0 = a1
fill a1 i
    # f  = toReal i
    # a2 = {a1 & [i - 1] = f}    // write f to index i-1 in a1
    = fill a2 (i - 1)            // recurse with unique array a2
```

Clean

Here, we take in a unique array of floating point numbers and some unrestricted integer value, and fill the first cells of the array with the numbers up to that value. Here we know that it is safe to write to the array because it is unique, so no other references to it can exist

elsewhere. Once we are finished with the array later on, however, it is fine to discard it, as with an array in most other functional programming languages; this would not be possible if our array was linearly typed. This however does mean that uniqueness types are not appropriate for the earlier example of file handles—we cannot ensure that a unique file handle is closed, as it can be discarded at any time.

In summary, linearity and uniqueness provide the same guarantees unless a system also has a notion of unrestricted value (non-linear or non-unique). The complementary but distinct use cases shown above make it clear that it would be valuable to have linearity and non-linearity alongside uniqueness and non-uniqueness in a single language, but this has previously not been possible. Our main contribution is a core calculus that allows linearity and uniqueness to coexist and interact, demonstrated also via an implementation in the Granule language. Next we consider the question of duality, and how to formally describe how linearity and uniqueness differ.

4.2.2 Are Linearity and Uniqueness Dual?

It is common in folklore and the literature to describe linearity and uniqueness as dual to one another (see e.g., [96, 134]) but rigorous versions of this statement are more rarely seen. The earliest formalisation of uniqueness is from Harrington's 'uniqueness logic' [57], which we use as a foundation for much of the following. Harrington constructs a logic which is much like linear logic, but instead of the ! modality for non-linearity it includes a $\circ$ modality for non-uniqueness which differs from non-linearity in its introduction rule.

Harrington's formulation of uniqueness is in the format of a sequent calculus, so we compare and contrast with a similar presentation of linearity. Remember that in linear logic, the ! modality acts as a comonad. This means that the introduction of ! on the right of a sequent means that all formulae on the left of a sequent must also have ! applied, whilst introduction of ! on the left is unrestricted:

$$\frac{!\Gamma \vdash P}{!\Gamma \vdash !P}\,!_{\mathrm{R}} \qquad \frac{\Gamma, P \vdash Q}{\Gamma, !P \vdash Q}\,!_{\mathrm{L}}$$

(also known as *storage* and *dereliction* respectively [51]). In contrast, the non-uniqueness modality $\circ$ of Harrington acts as a monad, meaning that introduction of $\circ$ on the right is unrestricted but introduction of $\circ$ on the left of a sequent means that all formulae on the right of the sequent must also have $\circ$ applied:

$$\frac{\Gamma \vdash P}{\Gamma \vdash P^{\circ}}\,{}^{\circ}\mathrm{R} \qquad \frac{\Gamma, P \vdash Q^{\circ}}{\Gamma, P^{\circ} \vdash Q^{\circ}}\,{}^{\circ}\mathrm{L}$$

The non-uniqueness modality $\circ$ then has the following structural rules for contraction and weakening, which are conspicuously identical to those for the ! modality representing non-linearity:

$$\frac{\Gamma, P^\circ, P^\circ \vdash R}{\Gamma, P^\circ \vdash R}\circ_{\mathrm{C}} \qquad \frac{\Gamma \vdash R}{\Gamma, P^\circ \vdash R}\circ_{\mathrm{W}} \qquad\qquad \frac{\Gamma, !P, !P \vdash R}{\Gamma, !P \vdash R}!_{\mathrm{C}} \qquad \frac{\Gamma \vdash R}{\Gamma, !P \vdash R}!_{\mathrm{W}}$$

One might be tempted to think that because the introduction rules for $\circ$ behave dually to those of !, the modalities are simply dual to one another, and thus non-uniqueness is equivalent to linear logic's ? (sometimes called the "why not" modality). But since the contraction and weakening rules for $\circ$ are the same as those for !, this is not quite the case; $\circ$ behaves dually to ! in some ways but not in others. Formally, $\circ$ is a monad while ! is a comonad, but both are comonoidal, whereas ? is monoidal.

Linear logic allows us to derive $!P \vdash P$ (from dereliction), which agrees with our notion of linearity where non-linear values can be restricted to behave linearly going forwards but if we have a linear value it must remain linear; uniqueness logic conversely allows us to derive $P \vdash P^\circ$, formalising our concept of uniqueness where we can forget the uniqueness guarantee and turn a unique value into a non-unique one, but if we have a non-unique value we cannot go back.

We can now make more precise the intuitive notion we have developed, which suggests that linear types provide a restriction on what can be done with a value 'in the future' whilst uniqueness types provide a guarantee about what has been done with a value 'in the past'. The distinction becomes clearer when we consider substitutions, which are generated by β-reductions.

The definition of syntactic substitution is the same whether we are working with linear logic or uniqueness logic, as the rules for functions are identical, but the difference arises when thinking about what it is possible to know about a value in one logic compared to the other. Given a linear value, we know that substituting this value into an expression will preserve linearity, as there is no way to transform a linear value into a non-linear one. Conversely, given a unique expression then we know that any values substituted in will not affect the uniqueness guarantee, as there is no way to transform a non-unique value into a unique one. Thus 'future' refers to outgoing substitutions, while 'past' refers to incoming substitutions.

So if linearity and uniqueness do in fact behave the same in some ways but not all, and they do in fact behave dually in some ways but not all, then what is the overall takeaway? What statement can we make about the relationship between their behaviour that reconciles these two viewpoints?

Takeaway. *Linearity and uniqueness behave dually with respect to composition, but identically with respect to structural rules, i.e., their internal plumbing.*

In other words, internally the non-linear and non-unique modalities are both comonoidal, so they allow for the same behaviour of contraction and weakening for values that are wrapped inside them.

But the duality arises upon considering how we can map into and out of these modalities; we can map out of the non-linear modality and retrieve a linear value, but we can never map into it, giving the modality its familiar comonadic structure. Conversely, we can map a unique value into the non-unique modality to allow for contraction and weakening, but we can never map back out of it, which explains the dual monadic behaviour of this modality.

It is this understanding of the similarities and differences between linearity and uniqueness that will allow us to unify them, and have values of both flavours present in a single type system, which will be our goal for the next section.

4.3 THE LINEAR–CARTESIAN–UNIQUE CALCULUS

We now consider how to represent both linearity and uniqueness in the same system. The first choice to make is whether our base values will be linear or unique[3], as this will influence the directionality of the modalities we need to include in the calculus. Here we present a system where linearity is the base and uniqueness is a modality, as opposed to one where uniqueness is the base and linearity is a modality, for two reasons.

- The first reason is pragmatic; we later implement our approach in Granule, which already has linear values as the default. Therefore, including uniqueness as an additional modality in the system will require far fewer changes to the language, since a unique base would most likely require a redesign. Moreover, languages with uniqueness types like Clean generally have non-unique values as their default, with uniqueness having to be specifically annotated; a system with a uniqueness modality will also map more closely onto these languages than one where uniqueness is the basis.

- The second reason is that developing a sound calculus with a unique base is more complex. Consider such a hypothetical calculus with a modality $\circ$ representing unrestricted values and a modality $\bullet$ representing linear values. If we construct a product of linear values $(a^\bullet, b^\bullet)$, then this product is unique (rather than linear), so we can promote to an unrestricted product $(a^\bullet, b^\bullet)^\circ$ and freely duplicate the product, though the values contained

3 We choose a substructural basis over an unrestricted one since this more closely maps to both linear and uniqueness logic, where values have substructural behaviour by default unless they are wrapped in a modality.

> within are linear. A linear base avoids this problem (among others) as products being linear by default means that their usage is maximally restricted, so there is no circumventing either a uniqueness guarantee via their construction or a linearity restriction via their duplication[4].

Given a linear basis, we formalise the idea that we can map from unique to non-unique and from non-linear to linear. The key insight is that *we treat non-linearity and non-uniqueness as the same state* as both these states are unrestricted; we can do anything we like with, and have no guarantees for, an unrestricted value. We write $*P$ for a P with a uniqueness guarantee, similar to the syntax of Clean and to avoid confusion with Harrington's $\circ$ modality for non-uniqueness. The resulting calculus, which we call the Linear-Cartesian-Unique calculus (LCU for short), builds on the (intuitionistic multiplicative exponential) linear logic of Chapter 2 with additional rules for uniqueness.

SYNTAX LCU's syntax is that of the linear λ-calculus with multiplicative products and unit (first line of syntax below) with additional terms for introducing and eliminating the ! modality and working with the uniqueness modality (second line):

$$\begin{aligned} t ::= {} & x \mid \lambda x.t \mid t_1\, t_2 \mid (t_1, t_2) \mid \mathsf{let}\,(x,y) = t_1 \,\mathsf{in}\, t_2 \mid \mathsf{unit} \mid \mathsf{let}\,\mathsf{unit} = t_1 \,\mathsf{in}\, t_2 \\ & \mid\, !t \mid \mathsf{let}\,!x = t_1 \,\mathsf{in}\, t_2 \mid \mathsf{share}\, t \mid \mathsf{clone}\, t_1 \,\mathsf{as}\, x \,\mathsf{in}\, t_2 \mid *t \end{aligned} \qquad \text{(terms)}$$

The meaning is explained in the next section with reference to typing.

4.3.1 Typing

Typing judgments are of the form $\Gamma \vdash t : A$, with types A defined:

$$A, B ::= A \multimap B \mid A \otimes B \mid 1 \mid\, !A \mid *A \qquad \text{(types)}$$

Thus our type syntax comprises linear function types $A \multimap B$, linear multiplicative products $A \otimes B$, a linear multiplicative unit 1, the non-linearity modality $!A$ and the uniqueness modality $*A$.

Typing contexts are defined as in Chapter 2:

$$\Gamma ::= \varnothing \mid \Gamma, x : A \mid \Gamma, x : [A] \qquad \text{(contexts)}$$

which are either empty, or contexts extended with a linear assignment $x : A$ or contexts extended with a non-linear assignment denoted $x : [A]$. The comma operator , concatenates disjoint contexts.

4 A similar problem arises from the application of unique functions, and this has been a thorn in the side of developers of uniqueness type systems for some time. The solution applied in Clean is that any function with unique elements in its closure is "necessarily unique", meaning it cannot be subtyped into a non-unique function and applied multiple times. Handily, this coincides with the notion of a linear function, which is why our calculus having a linear base also avoids this problem.

All of the typing rules from Chapter 2 are preserved. We extend these with the rules for our uniqueness modality which has two syntactic constructs, *share* and *clone*:

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathsf{share}\, t : !A}\ \textsc{share} \qquad \frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\, t_1\, \mathsf{as}\, x\, \mathsf{in}\, t_2 : !B}\ \textsc{clone}$$

The share rule maps a unique value to a non-linear value, allowing a uniqueness guarantee to be forgotten. In terms of the operational semantics (see Section 4.3.4), this causes evaluation of t before the share. Next, the clone rule says that a non-linear value of type A can be copied to produce a unique A which is used by t_2. The input is required to be non-linear so that we cannot circumvent a linearity restriction by cloning a linear value and immediately sharing it. These rules in turn are accompanied by the 'necessitation' rule that says values can be assumed unique as long as they have no dependencies:

$$\frac{\emptyset \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

The share and clone rules in this logic suggest a monad-like relationship between the ! and $*$ modalities, with the share rule representing the 'return' of the monad and the clone rule likewise acting as the 'bind'. The $*$ modality is not in itself a monad (or indeed, a comonad like !). Rather, it acts as a functor over which the ! modality becomes a *relative monad* [9], in much the same way that the $\circ$ modality for non-uniqueness from Harrington's work acts as a monad [57].

A relative monad comprises a functor J and an object mapping T, along with an operation $\eta : JX \to TX$ and a mapping from $JX \to TY$ arrows to $TX \to TY$ with axioms analogous to the monad axioms. Thus, here J is the uniqueness modality $*$ and T the non-linearity modality !. If one imagines the dual version of this logic where the basis is unique, the hypothetical linearity modality would act as a functor making the non-uniqueness modality into a *relative comonad* [5, 104] in much the same way.

4.3.2 Equational Theory

One way of understanding the meaning of the LCU calculus is to see its equational theory (which we later prove sound against its operational model). The calculus has the standard $\beta\eta$-equalities for the multiplicative linear λ-calculus fragment, which includes the following $\beta\eta$ rules for !:

$$\mathsf{let}\, !x = !t\, \mathsf{in}\, t' \equiv [t/x]t' \qquad (\beta!)$$

$$\mathsf{let}\, !x = t\, \mathsf{in}\, !x \equiv t \qquad (\eta!)$$

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

$$\frac{}{[\Gamma] \vdash \mathsf{unit} : 1}\ 1_I \qquad \frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

$$\frac{\Gamma_1 \vdash t_1 : 1 \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let\ unit} = t_1\ \mathsf{in}\ t_2 : B}\ 1_E$$

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ (x, y) = t_1\ \mathsf{in}\ t_2 : C}\ \otimes_E$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A] \vdash t : B}\ \textsc{der} \qquad \frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash\ !t : !A}\ !_I$$

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ !x = t_1\ \mathsf{in}\ t_2 : B}\ !_E$$

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathsf{share}\ t : !A}\ \textsc{share} \qquad \frac{\varnothing \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \textsc{clone}$$

Figure 4.1: Collected typing rules for LCU calculus

along with the following equalities on the uniqueness fragment:

$$\mathsf{clone}\ t\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{share}\ x) \equiv t \qquad \text{(unitR)}$$

$$\mathsf{clone}\ (\mathsf{share}\ v)\ \mathsf{as}\ x\ \mathsf{in}\ t' \equiv [v/x]t' \qquad \text{(unitL)}$$

$$\mathsf{clone}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{clone}\ t_2\ \mathsf{as}\ y\ \mathsf{in}\ t_3) \equiv \mathsf{clone}\ (\mathsf{clone}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2)\ \mathsf{as}\ y\ \mathsf{in}\ t_3 \qquad \text{(assoc)}$$

The first axiom states that cloning a non-linear t into a unique value x and immediately sharing it to be non-linear is equivalent to just t. The second axiom states that sharing a unique value v and cloning it to a unique x in the scope of t' is the same as just substituting in that v for x. The last axiom gives associativity of cloning under the side condition that x is free in t_3. These equations are exactly the relative monad axioms [9], though we specialise (unitL) slightly by restricting to values to account for the reduction semantics.

An ardent reader might question whether these axioms fully characterise our uniqueness modality, or whether some more comprehensive

set of axioms might hold. We considered it likely that the ! modality would act as a relative monad in our calculus due to its relationship with the non-uniqueness modality from Harrington's uniqueness logic, which is monadic in nature; in our case ! obeys similar rules except for the presence of the $*$ modality for unique values. These were the particular axioms we chose to verify for that reason.

It would be of interest to justify this choice further by exploring some form of extended denotational model for our broader calculus. We leave this to further work, and thus will discuss the notion briefly later in Chapter 8.

The typability of these axioms relies on the admissibility of linear and non-linear substitution shown in Section 4.3.5 on the metatheory of the calculus.

4.3.3 Exploiting Uniqueness for Mutation

A key use for uniqueness is that this allows mutation to be used safely—the original pun behind Wadler's "Linear Types can Change the World" [142]. To illustrate this idea, and consider its soundness in Section 4.3.4, we extend the LCU calculus with a primitive array type:

$$A ::= \ldots \mid \mathsf{Array}\ A \mid \mathbb{N} \mid \mathbb{F}$$

where $\mathbb{N}$ are natural numbers used for sizes and indices and $\mathbb{F}$ floating-point values. The calculus is also extended with operations for floating-point arrays, typed by axiomatic rules (with built-in weakening):

$$\begin{array}{lll}
[\Gamma] \vdash & \mathsf{newArray} & : \mathbb{N} \multimap *(\mathsf{Array}\ \mathbb{F}) \\
[\Gamma] \vdash & \mathsf{readArray} & : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F}) \\
[\Gamma] \vdash & \mathsf{writeArray} & : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F}) \\
[\Gamma] \vdash & \mathsf{deleteArray} & : *(\mathsf{Array}\ \mathbb{F}) \multimap 1
\end{array}$$

These operations provide the interface for exploiting unique array references, where writeArray performs mutation as the type system guarantees that uniquely typed values have not been duplicated in the past (Section 4.3.5). We ignore out-of-bounds exceptions as this is an orthogonal issue, which could be solved using indexed types. We elide rules for typing numerical terms here.

Our implementation in Section 4.4 replays these ideas in a practical setting. The next section gives the operational heap model for the calculus, where the semantics of mutation is made concrete. Note that the semantics we present in this chapter is call-by-name, whereas Granule has call-by-value semantics by default. A call-by-name semantics can be enabled in Granule by writing `language CBN` at the top of a program; then, the execution will mirror that of the model described below. We will discuss how the functionality of uniqueness types can be captured in a call-by-value setting in Section 5.5 of Chapter 5, as this will require some additional development.

4.3.4 Operational Heap Model

We define an operational model for the LCU calculus to make the meaning of uniqueness and linearity more concrete, and to prove that our type system enforces the desired properties. As mentioned above, the semantics is call-by-name; it resembles the small-step operational semantics for the graded λ-calculus illustrated in Chapter 3 but extended with a notion of *heaps*, both to capture the idea of a memory reference to arrays as well as to give a way to track resource usage on program variables outside of the types.

We adapt the model of Choudhury et al. [32], which was used to track resource usage in a pure language with graded types. Our model applies this idea to a non-graded setting, extended to include mutation and reference counting for uniqueness. To prove that linearity and uniqueness are respected (soundness), the heap semantics incorporates some typing information in order to ease the theorem statements and proofs as shown in Section 4.3.5.

Single-step reductions in the operational model are of the form:

$$H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta \qquad \text{(single-step judgment form)}$$

where H is the incoming heap which provides bindings to variables that appear in t and array allocations. The result of the reduction is a new term t' with an updated heap H', as well as two additional pieces of information: Γ gives us a 'binding context' recording the typing of any binders that were encountered (or 'opened') during reduction, and Δ gives us a 'usage context' containing an account of how variables were used. Usage contexts are defined as:

$$\Delta ::= \emptyset \mid \Delta, x : r \quad \text{(usage contexts)}$$

$$r ::= 1 \mid \omega \quad \text{(usage/reference counter)}$$

where r is a usage marker that says a variable was used either once (denoted 1) or used more than once (denoted ω). Usage has a preorder $\leq$ where $1 \leq \omega$.

We extend the syntax of terms with a value form a representing runtime array references to the heap. In order to account for their type, the syntax of contexts is extended to include assumptions $a : \mathsf{Array}\ A$ which are treated as a different syntactic category of variables. Additional runtime typing rules for array reference terms a are provided akin to a variable rule as below.

$$\frac{}{[\Gamma], a : \mathsf{Array}\ A \vdash a : \mathsf{Array}\ A}\ \mathsf{arr}$$

$$\frac{}{[\Gamma], a : \mathsf{Array}\ A \vdash *a : *(\mathsf{Array}\ A)}\ *\mathsf{arr}$$

Heaps are defined as follows akin to a context but containing two kinds of 'allocations' for variables x and for array references a:

$$H ::= \emptyset \mid H, x \mapsto_r (\Gamma \vdash t : A) \mid H, a \mapsto_r \mathbf{arr} \qquad \text{(heaps)}$$

In the case of extending the heap with a variable allocation for x, the heap records that x can be used according to r and that it maps to a term t, along with its typing which is only present to aid the metatheory. For brevity, we sometimes write $x \mapsto_r t$ instead of $x \mapsto_r (\Gamma \vdash t : A)$ when the typing is not important. In the case of an array reference a, the heap records the number of references currently held to it, where r is again used (representing either one reference 1 or many ω), and describes the heap-only array representation term **arr** pointed to by that reference (whose syntax we introduce later along with the relevant rules).

Multiple reductions are composed from zero or more single-step reductions, with judgments of the form $H \vdash t \;\Rightarrow\; H' \vdash t' \mid \Gamma \mid \Delta$ given by two rules capturing empty reduction sequences and extending a sequence at its head:

$$\frac{}{H \vdash t \;\Rightarrow\; H \vdash t \mid \emptyset \mid \emptyset}\ \textsc{refl}$$

$$\frac{\begin{array}{c} H \vdash t_1 \;\rightsquigarrow\; H' \vdash t_2 \mid \Gamma_1 \mid \Delta_1 \\ H' \vdash t_2 \;\Rightarrow\; H'' \vdash t_3 \mid \Gamma_2 \mid \Delta_2 \end{array}}{H \vdash t_1 \;\Rightarrow\; H'' \vdash t_3 \mid \Gamma_1, \Gamma_2 \mid \Delta_1 + \Delta_2}\ \textsc{ext}$$

In the case of EXT the binding contexts are disjoint (since we treat binders as unique in a standard way) but the usage contexts are added as follows:

$$\Delta_1 + \emptyset = \emptyset \qquad \Delta_1 + (\Delta_2, x : r) = \begin{cases} (\Delta_1 + \Delta_2), x : r & x \notin \mathsf{dom}(\Delta_1) \\ (\Delta_1' + \Delta_2), x : \omega & \Delta_1 = \Delta_1', x : r' \end{cases}$$

i.e., if a variable x appears in both usage contexts then in the resulting context $x : \omega$ since for the purposes of our counting we are interested in counting 0 uses (via absence in Δ) or 1 use or many uses (ω).

4.3.4.1 Heap model

The reduction rules are collected in Appendix b, but we explain the core rules for the single-step reduction relation here. Unlike Chapter 3's semantics, variables have a reduction, with two possibilities:

$$\frac{}{H, x \mapsto_1 t \vdash x \rightsquigarrow H \vdash t \mid \emptyset \mid x : 1}\ \textsc{var1}$$

$$\frac{}{H, x \mapsto_\omega t \vdash x \rightsquigarrow H, x \mapsto_\omega t \vdash t \mid \emptyset \mid x : 1}\ \textsc{var}\omega$$

Both reduce a variable x to the term t which is assigned to x in the heap. In the top rule, we started out with a heap capability of 1 (linear) so after the reduction we remove x from the heap. In the bottom rule, we have a heap capability of ω (non-linear) so we preserve the assignment to x in the outgoing heap.

β-reduction is then given as follows:

$$\frac{\Gamma \vdash t' : A}{H \vdash (\lambda x.t)\, t' \rightsquigarrow H, x{\mapsto_1}(\Gamma \vdash t' : A) \vdash t \mid x : A \mid \emptyset} \rightsquigarrow_\beta$$

Rather than using a substitution, the body term is the result under a heap extended with x assigned to the (typed) argument term t'. This heap binding is given a resource capability of 1 since functions are linear. In the output, we remember that a linear binding has been opened up in the scope of the term. An inductive rule allows an application to reduce on the left:

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma \mid \Delta}{H \vdash t_1\, t_2 \rightsquigarrow H' \vdash t_1'\, t_2 \mid \Gamma \mid \Delta} \rightsquigarrow_{\text{APP}}$$

We elide the rules for products and unit which follow much the same scheme; one congruence to evaluate the reduction of an elimination form and one to enact a β reduction. For the ! modality, this scheme gives us the $!\beta$ rule which creates a non-linear binding of x to t_1:

$$\frac{[\Gamma] \vdash t_1 : A}{H \vdash \mathsf{let}\, !x = !t_1\, \mathsf{in}\, t_2 \rightsquigarrow H, x{\mapsto_\omega}([\Gamma] \vdash t_1 : A) \vdash t_2 \mid x : [A] \mid \emptyset} \rightsquigarrow_{!\beta}$$

The more interesting rules are for the uniqueness aspects of the language. Sharing (which maps a unique value type $*A$ to a non-linear value $!A$) has a congruence rule and a reduction to enact a share:

$$\frac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash \mathsf{share}\, t \rightsquigarrow H' \vdash \mathsf{share}\, t' \mid \Gamma \mid \Delta} \rightsquigarrow_{\mathsf{share}}$$

$$\frac{\mathsf{dom}(H) \equiv \mathsf{arrRefs}(v)}{H, H' \vdash \mathsf{share}\,(*v) \rightsquigarrow ([H]_\omega), H' \vdash !v \mid \emptyset \mid \emptyset} \rightsquigarrow_{\beta\mathsf{share}}$$

The key operational behaviour is in the bottom rule here, where the incoming heap is split into two parts, where H is such that it provides the allocations for all array references in v (enforced by the premise here). The unique value $*v$ is wrapped to be non-linear in the result $!v$ and thus all of its array references are now marked as 'many' via $[H]_\omega$ which replaces all reference counts with ω, e.g.:

$$H', a{\mapsto_1}\mathbf{arr} \vdash \mathsf{share}\,(*a) \rightsquigarrow H', a{\mapsto_\omega}\mathbf{arr} \vdash !a \mid \emptyset \mid \emptyset$$

Thus, sharing enacts the idea that a reference is no longer unique and may be used many times (and hence now is a non-linear value).

Cloning then has three reductions; a congruence (elided), a reduction which forces evaluation under the non-linear modality, and a β-reduction to enact cloning to a unique value:

$$\frac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash \mathsf{clone}\,!t\,\mathsf{as}\,x\,\mathsf{in}\,t_2 \rightsquigarrow H' \vdash \mathsf{clone}\,!t'\,\mathsf{as}\,x\,\mathsf{in}\,t_2 \mid \Gamma \mid \Delta}\rightsquigarrow_{\mathsf{clone!}}$$

$$\frac{\Gamma \vdash v : A \qquad \mathsf{dom}(H') \equiv \mathsf{arrRefs}(v) \qquad (H'', \theta) \equiv \mathsf{copy}(H')}{H, H' \vdash \mathsf{clone}\,!v\,\mathsf{as}\,x\,\mathsf{in}\,t \rightsquigarrow H, H', H'', x{\mapsto_1}(\Gamma \vdash *\theta(v) : *A) \vdash t \mid x : *A \mid \emptyset}\rightsquigarrow_{\mathsf{clone}\beta}$$

The $\rightsquigarrow_{\mathsf{clone!}}$ rule evaluates under ! so that the first term can be reduced to a value v to be copied in the next rule. The $\rightsquigarrow_{\mathsf{clone}\beta}$ rule enacts cloning where $\mathsf{dom}(H') \equiv \mathsf{arrRefs}(v)$ marks the part of the heap with array references coming from v. Then $\mathsf{copy}(H')$ copies the arrays in this part of the heap, creating a heap fragment H'' and a renaming θ which maps from the old array references to the references of the new copies. This renaming is applied to v in the newly bound unique variable x. Thus the value $\theta(v)$ refers to any freshly cloned arrays.

Lastly, the semantics of the array primitives uses an array representation on the heap, where **arr** is some array object and $\mathbf{arr}[i] = v$ indicates that the i^{th} element is bound to the value v, and we write $a\#H$ for an array reference a which is fresh for heap H:

$$\frac{a\#H}{H \vdash \mathsf{newArray}\,n \rightsquigarrow H, a{\mapsto_1}\mathbf{arr} \vdash *a \mid \emptyset \mid \emptyset}$$

$$\frac{}{H, a{\mapsto_r}(\mathbf{arr}[i] = v) \vdash \mathsf{readArray}\,(*a)\,i \rightsquigarrow H, a{\mapsto_r}(\mathbf{arr}[i] = v) \vdash (v, *a) \mid \emptyset \mid \emptyset}$$

$$\frac{}{H, a{\mapsto_r}\mathbf{arr} \vdash \mathsf{writeArray}\,(*a)\,i\,v \rightsquigarrow H, a{\mapsto_r}(\mathbf{arr}[i] = v) \vdash *a \mid \emptyset \mid \emptyset}$$

$$\frac{}{H, a{\mapsto_r}\mathbf{arr} \vdash \mathsf{deleteArray}\,(*a) \rightsquigarrow H \vdash \mathsf{unit} \mid \emptyset \mid \emptyset}$$

Thus newArray creates a fresh array reference a and allocates a new array on the heap with a single reference count. The readArray and writeArray primitives work as expected to read and destructively update the array referenced by a, whose reference count is arbitrary but unchanged by the reduction. Lastly deleteArray deallocates the array. Noticeably, the rules do not enforce uniqueness; but as we see in the next section, well-typed programs do preserve uniqueness.

4.3.5 Metatheory

Proofs of all the statements that follow are provided in Appendix b. We first establish some key results showing the admissibility of substitution and weakening, which are leveraged in later proofs:

Lemma 5 (Linear substitution). *If* $\Gamma' \vdash t' : A$ *and* $\Gamma, x : A \vdash t : B$ *then* $\Gamma' + \Gamma \vdash [t'/x]t : B$.

Lemma 6 (Non-linear substitution). *If* $[\Gamma'] \vdash t' : A$ *and* $\Gamma, x : [A] \vdash t : B$ *then* $[\Gamma'] + \Gamma \vdash [t'/x]t : B$.

Lemma 7 (Weakening is admissible). *If* $\Gamma \vdash t : A$ *then* $\Gamma, [\Gamma'] \vdash t : A$.

Next, the heap model allows us to establish the key properties of well-typed programs respecting linearity and uniqueness restrictions. We first define when a heap is *compatible* with a typing context:

Definition 3 (Heap-context compatibility). *A heap* H *is compatible with a typing context* Γ *if* H *contains assignments for every variable in the context and the typing contexts of the terms in the heap are also compatible with the heap. The relation is defined inductively as:*

$$\frac{H \bowtie \Gamma}{H, a \mapsto_r \mathbf{arr} \bowtie \Gamma, a : \mathsf{Array}\ A}\ \textsc{Ref} \qquad \frac{}{\emptyset \bowtie \emptyset}\ \textsc{Empty}$$

$$\frac{H \bowtie (\Gamma_1 + \Gamma_2) \quad \Gamma_2 \vdash t : A \quad x \notin \mathsf{dom}(H)}{(H, x \mapsto_r (\Gamma_2 \vdash t : A)) \bowtie (\Gamma_1, x : A)}\ \textsc{Lin}$$

$$\frac{H \bowtie (\Gamma_1 + [\Gamma_2]) \quad [\Gamma_2] \vdash t : A \quad x \notin \mathsf{dom}(H)}{(H, x \mapsto_\omega ([\Gamma_2] \vdash t : A)) \bowtie (\Gamma_1, x : [A])}\ \omega$$

Thus, a heap compatible with $\Gamma_1, x : A$ contains an assignment for x marked with a usage annotation r which can be either 1 for linear or ω for non-linear use. Note that non-linear values can be used linearly, as captured by dereliction (the DER typing rule). However, a non-linear assumption must have a heap assignment marked with ω (rule ω), where the dependencies of the assigned term t must all be non-linear in the remaining compatibility judgment on the rest of the heap.

From a heap (and likewise from a typing context) we can also extract usage information. This is useful for focusing on resource usage:

Definition 4 (Usage context extraction). *For a context* Γ *or heap* H *we can extract usage information denoted* $\overline{\Gamma}$ *or* $\overline{H}$ *defined as:*

$$\begin{aligned}
\overline{\emptyset} &= \emptyset \\
\overline{(\Gamma, x : [A])} &= \overline{\Gamma}, x : \omega \\
\overline{(\Gamma, a : A)} &= \overline{\Gamma} \\
\overline{(\Gamma, x : A)} &= \overline{\Gamma}, x : 1 \qquad \textit{(contexts)}
\end{aligned}$$

$$\begin{aligned}
\overline{\emptyset} &= \emptyset \\
\overline{(H, x \mapsto_r (\Gamma \vdash t : A))} &= \overline{H}, x : r \\
\overline{(H, a \mapsto_r t)} &= \overline{H} \qquad \textit{(heaps)}
\end{aligned}$$

We now give the two main theorems about our calculus which give us the properties that linearity is respected (called *conservation*, Theorem 3) and that uniqueness is respected (Theorem 4).

Theorem 3 (Conservation). *For a well-typed term* $\Gamma \vdash t : A$ *and all* Γ_0 *and* H *such that* $H \bowtie (\Gamma_0 + \Gamma)$ *and a reduction* $H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma_1 \mid \Delta$ *we have:*

$$\exists \Gamma'.\ \Gamma' \vdash t' : A \ \ \wedge \ \ H' \bowtie (\Gamma_0 + \Gamma') \ \ \wedge \ \ (\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{\Gamma_1})$$

The first conjunct is regular type preservation, linked with heap compatibility in the second conjunct. The last conjunct expresses the core of conservation: that resource usage accrued in this reduction, given by Δ, plus remaining resources given in the heap H' are approximated (via $\sqsubseteq$, the pointwise lifting of $\leq$) by the original resources given in the heap H plus the specification of the resources from any variable bindings Γ_1 encountered along the way. The context Γ_0 accounts for bindings not described by Γ, and is key to the inductive proof here.

We then establish that all heap references have only one reference to them at the end of execution.

Theorem 4 (Uniqueness). *For a well-typed term* $\Gamma \vdash t : *A$ *and all* Γ_0 *and* H *such that* $H \bowtie (\Gamma_0 + \Gamma)$ *and given a multi-reduction to a value* $H \vdash t \ \Rightarrow \ H' \vdash *v \mid \Gamma' \mid \Delta$, *for all* $a \in \mathsf{arrRefs}(v)$ *(array references in* v*) we have:*

$$\begin{aligned} a{\mapsto_1}t' \in H \implies \exists t''.a{\mapsto_1}t'' \in H' \\ \wedge \quad a \notin \mathsf{dom}(H) \implies \exists t''.a{\mapsto_1}t'' \in H' \end{aligned}$$

i.e., any array references contributing to the final term that are unique in the incoming heap stay unique in the resulting term, and any new array references contributing to the final term are also unique.

Notice that there is a certain duality between the conservation theorem and the uniqueness theorem which mirrors the relationship between linearity and uniqueness. The statement of conservation is a generalised way to say that if a variable is linear then it will always be used in a linear way, or in other words that linearity restrictions will always be upheld; conversely, the uniqueness theorem tells us that if a variable is unique then it must always have been used in a unique way, or in other words that it does not have multiple references.

One important point to notice is that the additional rules (share and clone) that we include for unique types are in fact trivial cases when it comes to the uniqueness theorem since they can never output a value with a unique type. This makes sense as the idea behind these additional rules is to mediate the interaction between uniqueness and non-uniqueness, and this interaction can only ever go in the direction of producing values that are non-unique.

Irreducible values of the heap model v are given by:

$$v ::= (t_1, t_2) \mid \mathsf{unit} \mid *t \mid \ !t \mid \lambda x.t \mid i \mid a \mid p$$

(value terms sub-grammar)

where p are partially-applied primitives, such as newArray, readArray, or readArray $(*a)$.

A sub-result of conservation is type preservation, complemented by a separate progress result in Theorem 5 to give syntactic type safety:

Theorem 5 (Progress)**.** *Given $\Gamma \vdash t : A$, then t is either a value, or if $H \bowtie \Gamma_0 + \Gamma$ there exists a heap H', term t', usage context Δ, and context Γ' such that $H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma' \mid \Delta$.*

Finally, we see that the operational semantics, extended to full β-reduction (i.e., all congruences), supports the equational theory:

Theorem 6 (Soundness with respect to the equational theory)**.** *For all t_1, t_2 such that $\Gamma \vdash t_1 : A$ and $\Gamma \vdash t_2 : A$ and $t_1 \equiv t_2$ and given H such that $H \bowtie \Gamma$, there exists a value (irreducible term) v and $\Gamma_1, \Gamma_2, \Delta_1, \Delta_2$ such that there are full β-reductions to the same value*

$$H \vdash t_1 \Rightarrow_\beta H' \vdash v \mid \Gamma_1 \mid \Delta_1 \quad \wedge \quad H \vdash t_2 \Rightarrow_\beta H' \vdash v \mid \Gamma_2 \mid \Delta_2$$

4.4 IMPLEMENTATION

4.4.1 Frontend

The implementation of uniqueness types in Granule follows much the same pattern as the logic defined earlier. Granule already possesses a *semiring graded necessity modality*, where for a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$, there is a family of types $\{\Box A_r\}_{r \in \mathcal{R}}$. We represent the ! from linear logic (and our calculus) via the pre-ordered semiring $\{0, 1, \omega\}$ (none-one-tons [94]) with $!A = \Box_\omega A$. (As discussed earlier, capturing the precise behaviour of ! requires some additional structure which is present on this semiring in Granule [63], but again we do not linger on this here.)

This semiring allows us to represent both linear and non-linear use: variables graded with 1 must be used linearly, with 0 must be discarded, and a grade of ω permits unconstrained use à la linear logic's ! modality.

(In Granule, $\Box_\omega A$ can be written as the type `A [Many]`, but we syntactically alias this to `!A` for simplicity and ease of understanding.)

As in LCU, uniqueness is represented by a new modality, which we call $*$ to match the calculus (and so that the syntax of programs involving uniqueness will be familiar to Clean users). The uniqueness modality wraps a value that behaves 'linearly' (and so cannot be duplicated or discarded), with the key difference being that we provide primitive functions which allow ! to act as a relative monad over unique values. The primitives have the following type signatures:

```
uniqueReturn : ∀ {a : Type} . *a → !a                -- share
uniqueBind : ∀ {a b : Type} . (*a → !b) → !a → !b    -- clone
```

Granule

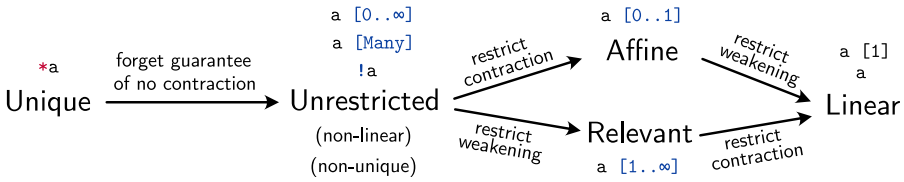

Figure 4.2: Relationship between various flavours of substructural types

The `uniqueReturn` function here implements the SHARE rule from the calculus (acting as the 'return' of the relative monad), and similarly the `uniqueBind` function implements the CLONE rule (acting as the 'bind').

We provide syntactic sugar for both of these primitives for convenience, with syntax designed to evoke the rules from the LCU calculus; `share x` is equivalent to writing `uniqueReturn x`, while `clone t1 as x in t2` is equivalent to writing `uniqueBind (λx → t2) t1`.

```
sip : *Coffee → (Coffee, Awake)
sip fresh = let !coffee = share fresh in
                (keep coffee, drink coffee)
```
Granule

Here, sharing converts the unique `Coffee` value into an unrestricted one, so that it can be duplicated and used twice for the two separate functions. Note however that the uniqueness guarantee is lost in the process, so both of the output values are non-unique (and here they also happen to be linear).

Figure 4.2 illustrates the relationship between uniqueness, linearity and other common forms of substructural typing in Granule proper. We implemented a built-in library for primitive floating point arrays in Granule, matching the interface for arrays of floats that was introduced as an extension to the LCU calculus in Section 4.3.3, with operations typed as follows:

```
newFloatArray    : Int → *FloatArray
readFloatArray   : *FloatArray → Int → (Float, *FloatArray)
writeFloatArray  : *FloatArray → Int → Float → *FloatArray
deleteFloatArray : *FloatArray → ()
```
Granule

The `writeFloatArray` primitive updates an array destructively in place since we have a guarantee that no other references exist to the array which has been passed in. In the next section, we use this set of primitives to evaluate the performance of our implementation, by measuring the performance gains from allowing for in-place updates in this fashion. We have another set of *immutable* primitives akin to the above (but written with a suffix `I`) which work with non-unique arrays), and thus do not perform mutation. These are typed as follows:

```
newFloatArrayI    : Int → FloatArray
readFloatArrayI   : FloatArray → Int → (Float, FloatArray)
writeFloatArrayI  : FloatArray → Int → Float → FloatArray
deleteFloatArrayI : FloatArray → ()
```

Granule

The following shows an example of `clone`, where a new array is created and a copy of this shared `FloatArray` on line 3 is deleted, leaving the original (immutable) instance of the array unaffected on line 4:

```
let x = newFloatArray 10 in
let [y] = share x in
let [()] = clone [y] as y' in
    (let () = deleteFloatArray y' in [()])
in readFloatArrayI y 10
```

Granule

One might wonder about the possibility of applying promotion at the point of creating a new array, by writing something like `let [x] = [newFloatArray 10] in ...` – at first glance it seems like this might allow for a unique reference to be used in an unrestricted way. Because of our call-by-name semantics, this is not in fact the case; each usage of the variable `x` results in a separate call to `newFloatArray`, and so we end up creating distinct references to distinct arrays. A call-by-value semantics would have different behaviour, and so naïvely allowing this would indeed lead to the possibility of breaking a uniqueness guarantee. We will explore this further in Chapter 5.

4.4.2 Compilation and Evaluation

As part of our implementation of uniqueness types in Granule, as described in Section 4.4.1, we also implemented a simple compiler that translates programs into Haskell. This compiler preserves the value types, but erases all of Granule's substructural types (linear, unique, graded, etc.). As a result, we can take advantage of both Granule's flexible type system and Haskell's libraries and optimizing compiler. In this chapter, all performance results were measured by compiling Granule programs to Haskell, and compiling the resulting Haskell with GHC 9.0.1. The measurements were collected on a MacBook with a 2 GHz quad-core Intel i5 processor and 16 GB of RAM.

As mentioned in Section 4.1, one motivation for using uniqueness types is to do the kind of in-place mutation necessary for efficient programming with arrays.

To check that our implementation is reasonable, we carried out an evaluation using an array processing benchmark. The benchmark recursively allocates and sums up lists of arrays of various sizes, with the goal of demonstrating the benefits of uniqueness types for arrays in functional programming. Each iteration of the benchmark allocates

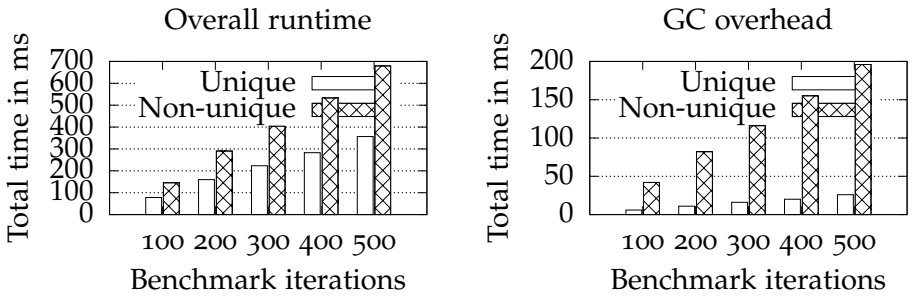

Figure 4.3: Performance of mutable vs. immutable arrays in Granule

a list of a thousand arrays, populates the arrays with values, then traverses the list to sum them up.

We prepared two versions of this benchmark: one with functional in-place updates and manual (safe) deletion of unique arrays, and one with non-unique, immutable, garbage collected arrays and updates via copying. The overall performance of these two benchmarks is shown in Figure 4.3, with lower bars/numbers representing better performance. The results, while not surprising, do confirm that array-handling is generally more efficient when in-place mutation is allowed.

Additionally, in Figure 4.3, we compared the time spent in garbage collection between the two versions of the benchmark. Because our implementation allocates unique data outside of GHC's heap, and uniqueness types allow programmers to directly de-allocate objects in memory, the unique version of the benchmark spends significantly less time in garbage collection. For this benchmark, the unique arrays are outside of the garbage collected heap and directly de-allocated, while other incidental objects (closures, lists, and so on) are still handled by the garbage collector.

Of course, this is a somewhat contrived benchmark. Real-world Haskell libraries, for example, typically provide functional high-level interfaces for array manipulation while using *unsafe* code to mutate arrays internally. The popular `vector` library[5] is one example, and `repa` [70] is another. Additionally, there is significant prior work on improving the efficiency of functional programs operating on arrays (for example, using combinators like `map` and `fold` along with aggressive fusion [38, 70, 88]), which we will not dwell on. The main point is that, at some stage, arrays must be mutated. Rather than having this happen through unsafe code, or via external C or Fortran, uniqueness types give us a way to do that mutation directly in our functional language, efficiently and safely.

Crucially, in these comparisons, all versions of the programs are implemented in the same language: Granule. With our extensions, the language is expressive enough to encompass a variety of programming

5 https://hackage.haskell.org/package/vector

approaches. Functional programmers may freely mix and match from a variety of options for data management and manipulation. Object lifetimes may be either manually or automatically managed, and object contents may allow in-place mutation or be immutable.

4.5 CONCLUSION

Linearity and uniqueness are both well-studied concepts with similar substructural foundations, but differing benefits; linearity enables the careful management of resourceful data, while uniqueness offers the possibility of safe in-place updates. By formalising the relationship between these two ideas, building on two distinct bodies of literature, we have shown that there is value in having both linear and unique types in the same type system. This is the first step on the road towards properly understanding the relationships between more advanced substructural type systems, such as the fine-grained resource tracking of Granule and Idris and the complex memory management provided by Rust; we will aim to make more progress on this lofty goal as we move forwards into Chapter 5.

Moreover, we implemented this system in the graded modal setting of the Granule language and provided benchmarks to demonstrate the efficiency gains that can be accessed via adding uniqueness to a language that already has a linear basis. The opportunities to incorporate uniqueness types into languages outside of Granule are apparent, and this chapter offers both a theoretical underpinning for uniqueness as it relates to linearity as well as clear validation of the performance benefits that a system which unifies linearity, uniqueness and unrestricted behaviour can offer.

The overall direction of this thesis is to demonstrate that graded modal types can be used to verify global program properties such as memory and communication safety. Thus far, we have shown that uniqueness types can be employed alongside linear types to ensure some simple memory safety properties. The next chapter will illustrate how moving from just uniqueness to a new flavour of graded modality can allow for the verification of more precise guarantees.

5 DIVIDE AND CONQUER
OWNERSHIP AND BORROWING

5.1 INTRODUCTION

The Rust programming language has dramatically grown in popularity in recent years, having been adopted as the second official language of the Linux kernel[1], and deployed in production code by companies including AWS, Huawei, Google, Microsoft and Mozilla, all of whom are founding members of the Rust Foundation.[2] This popularity is in large part due to its focus on memory safety; Rust finds a happy medium between systems programming languages like C which offer precise control but little in the way of safety guarantees and the contrasting approach of higher-level languages like Java or Go where memory is managed automatically through garbage collection.

The intricate ownership system which characterises Rust's approach to memory management is inspired in part by the literature on using linear types for tracking resource usage [142]. The precise theoretical relationship between linear types and the properties that can be enforced by Rust's borrow checker remains unclear, however, particularly since Rust lies outside the traditional functional paradigm to which linearity is most suited, opting for a more mixed imperative and functional approach.

As was explored in Chapter 3, the Granule research language allows various properties of data use and data flow to be tracked smoothly in a single type system via an indexed modality $!_r$ with some algebraic structure on the indices r [47, 103]. In this work, we seek a shared understanding between the precise resource reasoning of graded type systems like Granule's and concepts of uniqueness, borrowing, and ownership as typified by Rust but also appearing in various other languages such as C++, even though these concepts require the ability to capture global properties about programs.

There already exist many models of ownership given from a type-theoretic perspective. For example, λ_{Rust} developed as part of the RustBelt framework provides a low-level model of Rust's ownership and borrowing systems suitable for formally verifying properties of Rust programs [69]. Oxide [151] and FR [107] provide more high-level abstract models. Our focus is *not* on replicating these efforts; we do not aim to directly model all the details of Rust's particular approach to ownership. Rather, we offer a different and complemen-

1 https://www.zdnet.com/article/rust-takes-a-major-step-forward-as-linux-s-second-official-language/

2 https://foundation.rust-lang.org/news/2021-02-08-hello-world/

tary perspective: a single system that relates general ownership and borrowing ideas to linear and graded types. This integration enables type, resource, and memory safety to be enforced via a single type system.

A natural place to start is *uniqueness types*, which provide the concept that a uniquely typed value has exactly one reference, and therefore is safe to mutate [16, 126, 141]. This gives a simplistic model of *ownership* without borrowing—a resource with a unique owner may be modified by that owner, but a broken guarantee of uniqueness can never be recovered. The development of Chapter 4 illustrates a type system that captures the relationship between linear and unique types: linear values are restricted from being copied or discarded in the future, whilst unique values are guaranteed to have never been copied in the past. We build upon that system as our foundation here but leverage the generalisations of grading.

In the ownership framework of Mycroft and Voigt [98], while uniqueness allows for memory-safe *temporal* aliasing, borrowing extends uniqueness by introducing the potential for safe patterns of *spatial* aliasing, dramatically increasing flexibility for the programmer. Just as the non-linearity modality ! can be generalised to a graded modality $!_r$ for fine-grained reasoning about resource usage via r and its algebraic structure, we show that uniqueness types, represented through a $*$ modality as in Chapter 4, can be generalised to a graded modality $\&_p$ to capture this idea that borrowing is a controlled relaxation of the uniqueness guarantee that we explored earlier.

Our approach allows many ideas from Rust, such as immutable and mutable borrows, partial borrows and reborrowing to be explained explicitly in a functional setting, all through the application of a form of grading based on the elegant fractional permissions of Boyland [26]. Our work integrates and relates, within a unified framework, the substructural type systems of linearity, uniqueness, grading, ownership, and borrowing. This offers a pathway to expressing memory safety properties naturally in functional languages like Haskell or Idris.

The outline of the chapter is as follows. First, in Section 5.2 we recap the key concepts of uniqueness and borrowing using Rust as a convenient exemplar. We discuss how various programs making use of these ideas will later be rewritten as Granule programs through our unified type system, in order to motivate the rest of our work. We then move on to the primary contributions of this chapter:

- In Section 5.3, we adapt the graded calculus of Chapter 3 slightly to account for a call-by-value semantics and introduce a notion of *existential type*. We then extend this calculus with the uniqueness modality of Chapter 4, again with some adaptation needed for the adjusted semantics.

- In Section 5.4, we further extend the notion of uniqueness developed in Chapter 4 to allow for multiple immutably borrowed

references at a time, by annotating reference types with additional information, similarly to Boyland's fractional permissions. We show that this allows for a more fine-grained approach to tracking uniqueness in much the same way that grading increases expressivity over pure linearity.

  - In Section 5.4.1, we develop an equational theory for our extension to Granule's type system, showing that the modality for unique ownership induces a *relative functor* over the new modality for mutable and immutable borrowing.
  - In Section 5.4.2, we discuss how using distributive laws can allow for borrowing only part of a larger data structure while leaving the rest uniquely owned, enabling a much wider variety of practical programming patterns.

- In Section 5.5, we detail the semantics for the calculus developed thus far, and prove various key properties—both standard notions like progress and preservation but also borrow safety and uniqueness. The semantics is call-by-value, representing a practical system. This departs from the semantics of Chapters 3 and 4 and indeed from most previous operational models of grading in the literature.

Sections 5.6 and 5.7 discuss the many and varied areas of similar research that have inspired this chapter. Full definitions along with proofs of all theorems can be found in Appendix c.

5.2 KEY OWNERSHIP AND BORROWING CONCEPTS

We first give an overview of the six key patterns of ownership and borrowing which we capture in our system in the rest of this chapter, using simple Rust examples to demonstrate the patterns in action. The examples rely on a single base value of type `Colour(u32, u32, u32)` - a struct containing three unsigned integers, representing a colour with red, green, and blue components. We mark negative examples (those that do not type check) with ✗ and positive examples with ✓.

OWNERSHIP The first crucial concept is *owned* values, where a value is 'owned' by a particular *identifier*. Below, the value `Colour(220, 20, 60)` is owned by the identifier `scarlet`. Each value can only be owned by a single identifier at any given time. Rust enforces this via *move semantics*: on the second line, ownership of the value is *moved* to the identifier `x`. Now the identifier `scarlet` no longer owns the value, so attempting to use it again on the third line gives an error.

```
let scarlet = Colour(220, 20, 60);
let x = scarlet;
let y = scarlet; // error: use of moved value: ‘scarlet‘
```

Rust ✗

It is fairly clear that the idea here in some way relates to linearity, since linear values can only be used once which restricts them to being passed around sequentially; indeed, much of the literature on Rust makes mention of linear (or affine) types. We demonstrate through our unified type system in Section 5.3 that in fact ownership can be better understood as an extension of *uniqueness* types, since each value having a single owner implies that the owner holds the unique reference to said value. As discussed in Chapter 4, these are similar to linear types in some ways but with important differences.

IMMUTABLE BORROWING Borrowing generalises the concept of ownership, permitting multiple references to point to a single value simultaneously. Rust's *borrow checker* manages all borrows that exist at a given time, ensuring that memory safety properties are maintained until values are eventually returned to their owners. *Immutable* borrows are one flavour of reference: any number of these can exist at a given time, but mutation of a value is disallowed through an immutable borrow, since this could result in data races. Below, two immutable borrows (`x` and `y`) both reference the original `persimmon` value.

```
let persimmon = Colour(252, 118, 5);
let x = &persimmon; // & denotes a borrow of ‘persimmon‘
let y = &persimmon; // ok!
```

Rust ✓

MUTABLE BORROWING It is also possible to borrow values and retain the capacity to mutate them, but this comes at a price: as mentioned above, allowing mutation through multiple references simultaneously is harmful to memory safety, so in order to prevent this only a single mutable borrow is permitted at a time. Much like owned values, mutably borrowing a value guarantees the sole capability for destructive access to the underlying data. This is demonstrated below, where attempting to create two *mutable* borrows following the pattern of the above example is disallowed.

```
let mut viridian = Colour(52, 161, 128);
let x = &mut viridian;
let y = &mut viridian;
// error: cannot borrow ‘viridian‘ as mutable more than once
// ... code using ‘x‘ and ‘y‘
```

Rust ✗

Note, however, that the first three lines copied verbatim into a Rust file will in fact compile successfully unless additional code is introduced that makes use of the variables `x` and `y`. This is due to a feature

called *non-lexical lifetimes*, through which the Rust compiler is able to *infer* that allowing two mutable borrows is safe as long as one of them is never used. We do not linger on this, since we will not capture this behaviour in our type system. Our goal is to embed some essential ownership and borrowing patterns *explicitly*. However, we will mention the possibility for extending our system with more advanced ideas in Chapter 8.

MIXING MUTABLE AND IMMUTABLE BORROWS Rust also disallows immutable borrows of values that are already borrowed mutably, since this invites similar problems: e.g., a value could be updated through the mutable borrow whilst being read through the immutable borrow. Hence, the following example wherein the second reference is borrowed immutably rather than mutably is also forbidden by the borrow checker.

```
let mut cerulean = Colour(3, 109, 230);
let x = &mut cerulean;
let y = &cerulean;
// error: cannot borrow ‘cerulean‘ as immutable
//        because it is also borrowed as mutable
```

Rust ✗

In Section 5.4, we demonstrate that both mutable and immutable borrows can be represented through a graded generalisation of uniqueness typing to represent a borrower's level of access.

PARTIAL BORROWING A useful pattern for working with larger data structures is to borrow only *part* of the structure while the original owner retains access to what remains. This is valuable for allowing one part of a program to work with a particular piece of data without restricting access to the entire structure, enabling tasks to be carried out in parallel more easily. In the below example, the red and green components of the `indigo` value are simultaneously borrowed mutably. The Rust compiler allows this since the references point to disjoint parts of the original struct:

```
let mut indigo = Colour(32, 36, 209);
let r = &mut indigo.0;
let g = &mut indigo.1; // ok!
```

Rust ✓

In Section 5.4.2, we take advantage of Granule's functional nature to present a cohesive way of managing this kind of partial borrow at the type level, including the possibility for working with disjoint components of the original structure concurrently.

REBORROWING The final pattern we aim to capture in our system, also essential for practical programming, is the notion of *reborrowing*, through which it is possible to create a borrow of an identifier that

itself references another value. We will show in Section 5.4 that this behaviour falls out of our generalisation of uniqueness naturally, without a need for additional constructs. To illustrate the general idea, consider the following example; here, the red component of the `amethyst` value is borrowed mutably as `r`, then another immutable borrow `x` is created pointing to `r`. The value can now only be read through `x`—both `r` and `amethyst` are inaccessible until the borrow is complete.

```
let mut amethyst = Colour(98, 1, 170);
let r = &mut amethyst.0;
let x = & *r; // ok!
```

Rust ✓

5.3 OWNED IS UNIQUE

The development in this chapter is based on the core calculus described in Chapter 3, which is also similar to the LCU calculus from Chapter 4 with the exception that we now reintroduce general coeffect grades rather than just using the standard bang modality to represent non-linearity. We rapidly recap that calculus below; everything remains the same with the sole exception of the promotion rule, which requires an explicit restriction that the value to be promoted must not be a *resource allocator*. This is designed to avoid problems that otherwise arise when promoting values with certain behaviours in a call-by-value setting; we will discuss this in detail when we describe our interface for mutable arrays in Section 5.4.

Terms in the graded calculus are given as follows:

$$t ::= x \mid \lambda x.t \mid t_1\, t_2 \mid (t_1, t_2) \mid \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 \mid () \mid \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 \\ \mid [t] \mid \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \qquad \text{(terms)}$$

with the syntax of types:

$$A, B ::= A \multimap B \mid A \otimes B \mid \mathsf{unit} \mid \Box_r A \qquad \text{(types)}$$

and contexts:

$$\Gamma ::= \varnothing \mid \Gamma, x : A \mid \Gamma, x : [A]_r \qquad \text{(contexts)}$$

The typing rules are largely the same as those presented in Chapter 3, with the exception of the additional restriction on promotion that was mentioned above, which we will expand upon later.

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

$$\frac{\Gamma \vdash t : A \qquad \neg \mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\, [x] = t_1\, \mathbf{in}\, t_2 : B}\ \textsc{elim}$$

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

The first extension we make to the core calculus is to represent *ownership*: values that have a unique owner can mutate the value freely because they are in possession of the only reference that exists. It turns out this matches closely with the idea of *uniqueness* types; if it is possible to guarantee that a reference to a value is unique (i.e., it is the only reference that exists), then whichever part of the program (e.g., a given thread or process) holds that reference must be the owner.

Uniqueness types were introduced into a linear setting in Chapter 4, but we will extend this in two ways: first, we incorporate existentially typed identifiers into the typing, which will become important when multiple references need to be tracked, and second, as mentioned above, we generalise the rules of the previous chapter to allow for a graded necessity modality parameterised by an arbitrary semiring rather than the simple non-linearity ! modality.

As with uniqueness typing, the crucial insight for integrating owned values with linear and graded values is to consider any linear or graded value to have no ownership information attached; these values have their memory managed in other ways, such as explicitly by the programmer or implicitly by a garbage collector. We then introduce a modality, written $*$ (akin to Chapter 4's uniqueness modality) to type values that are *uniquely owned*, meaning that the type system must ensure only one reference exists at any given time. We extend the syntax given in Chapter 3 with terms and types:

$$\begin{aligned} t &::= \ldots \mid \mathbf{share}\, t \mid \mathbf{clone}\, t_1\, \mathbf{as}\, x\, \mathbf{in}\, t_2 \\ &\quad \mid \mathbf{pack}\, \langle id, t \rangle \mid \mathbf{unpack}\, \langle id, x \rangle = t_1\, \mathbf{in}\, t_2 \\ A, B &::= \ldots \mid *A \mid \exists id.A \qquad \text{(terms \& types, extended)} \end{aligned}$$

Importantly, we introduce existential types and type variables (restricted to a particular kind) to track identifiers associated with resources (whose dataflow paths may fork and join, due to the borrowing patterns we discuss later). These identifiers explicitly name and identify a resource at the type level such that references to different resources cannot be interchanged or joined as they will have different types, distinguished by the identifier. Existentially quantified types $id : \mathsf{Name}$ are members of an abstract kind of names Name.

Existential types have standard introduction and elimination typing forms, but restricted only to type variables of kind Name (whose kind is omitted here for simplicity due to this restriction):

$$\frac{\Gamma \vdash t : A \quad id \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathbf{pack}\ \langle id', t\rangle : \exists id.A[id/id']}\ \textsc{pack}$$

$$\frac{\Gamma_1 \vdash t_1 : \exists id.A \quad \Gamma_2, id, x : A \vdash t_2 : B \quad id \notin \mathsf{fv}(B)}{\Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ \langle id, x\rangle = t_1\ \mathbf{in}\ t_2 : B}\ \textsc{unpack}$$

We also extend the syntax of contexts as follows:

$$\Gamma ::= \ldots \mid \Gamma, id \qquad \text{(contexts, extended)}$$

Context addition and multiplication are augmented similarly (with the cases presented in Chapter 3 reiterated for ease of comparison):

$$\begin{aligned}
\emptyset + \Gamma &= \Gamma \\
\Gamma + \emptyset &= \Gamma \\
(\Gamma, x : A) + \Gamma' &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma'| \\
\Gamma + (\Gamma', x : A) &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma| \\
(\Gamma, x : [A]_r) + (\Gamma', x : [A]_s) &= (\Gamma + \Gamma'), x : [A]_{(r+s)} \\
(\Gamma, id) + (\Gamma', id) &= (\Gamma + \Gamma'), id
\end{aligned}$$

(context addition, extended)

$$r \cdot \emptyset = \emptyset \qquad r \cdot (\Gamma, x : [A]_s) = (r \cdot \Gamma), x : [A]_{r*s} \qquad r \cdot (\Gamma, id) = (r \cdot \Gamma), id$$

(context multiplication, extended)

Note that in many cases here, x may be id with $A = \mathsf{Name}$ implicitly.

The two constructs for this uniqueness modality $*$ similarly to Chapter 4 provide sharing and cloning, with types:

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathbf{share}\, t : \Box_r A}\ \textsc{share}$$

$$\frac{\begin{array}{l} 1 \sqsubseteq r \qquad \mathsf{cloneable}(A) \\ \Gamma_1, \overline{id} \vdash t_1 : \Box_r A \qquad \Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id}]) \vdash t_2 : B \end{array}}{(\Gamma_1 + \Gamma_2), \overline{id} \vdash \mathbf{clone}\, t_1\, \mathbf{as}\, x\, \mathbf{in}\, t_2 : B}\ \textsc{clone}$$

The SHARE construct allows a guarantee of unique ownership to be discarded, though this means that memory must now once again be managed automatically just as with any other Granule values where ownership is not tracked. Any grade r can be selected here, though the shared value must eventually be either discarded (requiring $0 \sqsubseteq r$ later in a derivation) or fully consumed via pattern matching.

The CLONE construct makes a *deep copy* of the value t_1, where we take a value that does not have an owner and make a unique copy of it; we take ownership of the copy, binding it the scope of t_2. We can guarantee uniqueness since this is now the only reference that exists to the copied value. We must update any identifiers along the way, however, to ensure that this is understood as a separate value to the original, in case other references still exist. This is mediated by existential types, where all identifiers $\overline{id}$ used in typing t_1 are renamed and bound under an existential quantifier.

Note, however, that cloning requires some additional conditions for soundness. The predicate $\mathsf{cloneable}(A)$ ensures the value is of a resource type—such as a reference—or a product of resources. (The full definition of cloneable is presented in Section 5.4, where we discuss said resource types in more depth.) This is necessary to prevent values which do not come with identifiers from being managed via ownership, since once we introduce borrowing in Section 5.4 this would lead to borrows of different values of the same type (such as two floating point numbers) being indistinguishable at the type level and thus interchangeable. However, this restriction does allow the output of the clone to be of any type, rather than necessarily a value with unrestricted behaviour as was the case in Chapter 4.

The side-condition $1 \sqsubseteq r$ explains that we must be able to accommodate a usage of the input t_1 since **clone** consumes the t_1 value once to copy it.

Our implementation of these ideas in Granule follows the above typing. We recap the initial ownership example from Section 5.2 here in our new extension of Granule using a data type `Colour` which acts as an alias for a triple of type `(Int, (Int, Int))`.

The following code illustrates that linear types already require us to obey the laws of *move semantics* for owned values, where we move ownership of `scarlet` to `x` and then to `y`:

```
exampleMove : *Colour → *Colour
exampleMove scarlet = let x = scarlet in
            -- ownership "moved" to x
                  -- let y = scarlet in ...
            -- would cause an error: non-linear use of ‘scarlet‘
                     let y = x in y
            -- but "moving" ownership from x to y is allowed
```

Granule

We can have multiple variables which all point to the initial value of type `Colour` via the **share** operation; this explicitly exempts the value from being managed by the ownership system. Here, we share at the interval grade `0..2`, capturing the idea that the value must be used somewhere between zero and two times (where here it happens to be used twice in this next example):

```
exampleShare : *Colour → (Colour, Colour)
exampleShare scarlet =
    let [s] : (Colour [0..2]) = share scarlet in
        let x = s in
        let y = s in (x, y)
```
Granule

The **share** and **clone** operations have the following equations showing their interaction:

$$\textbf{clone}\,(\textbf{share}\,v)\,\textbf{as}\,x\,\textbf{in}\,t \equiv t[\textbf{pack}\,\langle \overline{id}, v\rangle / x] \qquad (\beta_*)$$

$$\textbf{clone}\,t_1\,\textbf{as}\,x\,\textbf{in}\,(\textbf{clone}\,t_2\,\textbf{as}\,y\,\textbf{in}\,t_3) \equiv \textbf{clone}\,(\textbf{clone}\,t_1\,\textbf{as}\,x\,\textbf{in}\,t_2)\,\textbf{as}\,y\,\textbf{in}\,t_3 \quad (x \notin \mathsf{FV}(t_3)) \qquad (*\text{assoc})$$

The (β_*) axiom states that sharing a value v (term with no further reductions—see Section 5.5) and cloning it to create a new owned x in the scope of t is equivalent to substituting the original v for x in t (with its identifiers packed in an existential). The $(*\text{assoc})$ axiom is associativity of cloning.

The SHARE and CLONE rules from Chapter 4 are presented below, for reference. This chapter differs in the introduction of grading into both rules, the renaming during a clone, the additional restrictions on cloning, and the resulting more flexible output type in the clone rule.

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathsf{share}\,t : !A}\ \textsc{share} \qquad \frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\,t_1\,\mathsf{as}\,x\,\mathsf{in}\,t_2 : !B}\ \textsc{clone}$$

The presence of identifiers necessitates the existential typing. The need for identifiers is shown in the next section once we move to a generalisation of uniqueness with the ability to separate *immutable borrows* from *mutable borrows*, where existential types will allow for tighter control of the *lifetime* of borrowed values with identifiers.

5.4 IMMUTABLY BORROWED IS FRACTIONALLY UNIQUE

We now generalise from the above system which incorporates uniqueness into the core calculus to a system that allows for the uniqueness guarantees to be temporarily broken in controlled ways, such that we can continue to ensure memory safety and more closely approximate Rust-like ownership and borrowing rules. We take inspiration from

the fractional permissions of Boyland [26] (which themselves served as partial inspiration for Rust's notion of ownership) and from recent literature on integrating linear types with fractional permissions [89].

Just as grading non-linearity gives a more precise account of resource usage, we introduce borrowing as a graded form of non-*uniqueness* denoted $\&_p A$, allowing precise control of references:

$$\begin{aligned} A, B &::= \ldots \mid \&_p A && \text{(types, extended)} \\ p, q &::= * \mid f \qquad (\textit{where } f \in \mathbb{Q}, 0 < f \leq 1) && \text{(permissions)} \end{aligned}$$

where p is a new form of grade for tracking borrowing, called a *permission*. Permissions are either rational numbers f between 0 and 1 or a special permission $*$ representing uniqueness (described below). A *mutable borrow* is represented by $\&_1 A$ allowing mutable access to a value of type A while preserving the guarantee that it will be eventually returned to its original owner. Note that 0 is excluded: the typing rules we provide can never produce a value with permission 0.

For working with the borrowing graded modality $\&_p A$, terms are extended as follows:

$$t ::= \ldots \mid \textbf{withBorrow}\ t_1\ t_2 \mid \textbf{split}\ t \mid \textbf{join}\ t_1\ t_2 \quad \text{(terms, extended)}$$

The creation of mutable borrows is via **withBorrow** with the typing:

$$\frac{\Gamma_1 \vdash t_1 : *A \qquad \Gamma_2 \vdash t_2 : \&_1 A \multimap \&_1 B}{\Gamma_1 + \Gamma_2 \vdash \textbf{withBorrow}\ t_1\ t_2 : *B}\ \textsc{withBorrow}$$

Here, we allow for a uniquely owned $*A$ value to be borrowed and manipulated in a mutable way by some function t_2 which expects a $\&_1 A$ as input, so long as it returns the mutable borrow as a $\&_1 B$ in the output so that the original owner can reclaim this as a unique reference ($*B$). Encapsulating the mutable borrow behaviour inside a linear continuation ensures that it is impossible to construct a closed term with a borrowed type, which means borrowed references to values must eventually return full access to the owner of said value (in other words, our calculus only supports *lexical* lifetimes).

UNIQUENESS AS UNIQUELY BORROWED We absorb the development of Section 5.3 and of Chapter 4 by the inclusion of $* \in p$ and a type identity $*A \equiv \&_* A$. Thus, the borrowing modality captures both unique and borrowed values and hence functions polymorphic in their permission can range over both kinds of ownership. This enables us to later define operations that can work over uniquely owned or mutably borrowed values without needing multiple primitives.

IMMUTABLE BORROWS Mutable borrows, graded with permission 1, provide full access to a value, permitting reads and writes (shown later for resources). By contrast, *immutable borrows* allow multiple references

to a value as long as they cannot be mutated (which would otherwise violate memory safety, e.g., allowing data races). The **split** and **join** constructs mediate between mutable and immutable borrows:

$$\frac{\Gamma \vdash t : \&_p A}{\Gamma \vdash \textbf{split}\ t : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\ \textsc{split}$$

$$\frac{\Gamma_1 \vdash t_1 : \&_p A \qquad \Gamma_2 \vdash t_2 : \&_q A \qquad p + q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \textbf{join}\ t_1\ t_2 : \&_{p+q} A}\ \textsc{join}$$

where **split** takes a borrowed reference and splits it into two borrowed references to the same value, each graded by half the original permission.[3] Resources with fractional permission < 1 will only permit non-mutating operations (shown later) and thus splitting mutably borrowed resources ($p = 1$) leads to two immutably borrowed references to the resource. The dual **join** (re)combines two borrows into one with their permissions added, recombining two immutable borrows back to a mutable borrow when $p + q = 1$. Due to the interfaces for manipulating resources (shown later), the type A above will always contain a resource identifier *id*, so it is impossible to combine references to two different values. The addition and halving operations are defined only for fractions f, not $*$.

These rules do not encompass every detail of Rust's intricate ownership system–they instead aim to represent the *core* concepts of ownership and borrowing in such a way that they can integrate into an existing graded type system, leaving room for potential extensions. In particular, as was mentioned in Section 5.2, we only support *lexical* lifetimes, meaning that there is a class of programs which are accepted by Rust's compiler but cannot be represented in this calculus. We focus on being able to represent notions of ownership *explicitly* in the types, rather than using a complex static analysis like Rust's borrow checker to *infer* which programs are safe.

The following code demonstrates how all of these new primitives can be put together in a simple Granule program. More practical examples will be presented when we introduce interfaces for resources. This particular example, which is purely illustrative, assumes a function `observe` which operates over borrowed values. The precise behaviour of this function is elided, but it does not perform any mutation and so it is polymorphic in the permission:

3 SPLIT could be generalised to produce a vector with any number n of $\&_{p/n} A$ elements, or even to arbitrary permissions q and q' such that $q + q' = p$, but here we restrict to halving for simplicity.

```
observe : ∀ {p : Fraction} . & p Colour → & p Colour

exampleBorrow : *Colour → *Colour
exampleBorrow persimmon = withBorrow (λb →
                            let (x, y) = split b   in
                            let     x' = observe x in
                            join (x', y) in b) persimmon
```
Granule

In `exampleBorrow`, we *mutably* borrow the value `persimmon` as `b`, split this into two *immutable* borrows `x` and `y`, and apply the `observe` function to `x` before rejoining the borrows and returning the value.

The unsafe patterns demonstrated by the `viridian` and `cerulean` examples in Section 5.2 are, as desired, not representable in Granule. It is impossible for two mutable borrows or for a mutable borrow and an immutable borrow to coexist, since splitting a borrow must reduce its permission by the nature of the `split` primitive. We can, however, represent *reborrowing*; this simply involves continuing to split immutable borrows further, where recovering the original value now requires collecting all of the borrows once more. The order in which the borrows are rejoined is immaterial, as illustrated below.

```
exampleReborrow : *Colour → *Colour
exampleReborrow amethyst = withBorrow (λb →
                             let (x, y) = split b in
                             let (l, r) = split x in
                             let     x' = join (r, l) in
                             join (y, x') in b) amethyst
```
Granule

The benefits of uniqueness typing for memory management are well-trodden in prior work, and were described in more detail in Chapter 4; as a reminder, the idea is that the guarantee of a unique reference both allows safe in-place updates but also obviates the need for garbage collection, since memory can be reclaimed as soon as a reference is deleted. The system described here augments this by allowing increased flexibility while preserving the same guarantees—multiple references can be created via `split` but they are tracked precisely through fractional grades so that uniqueness is still guaranteed once they are re`join`ed, with in-place update only allowed for references with unique access (graded 1 or $*$).

RESOURCES Enforcing which behaviours should be allowed for different permissions on a given resource is mediated by said resource's interface. We will give two in-depth examples in this chapter to demonstrate the flexibility of our ownership and borrowing interface. The first (similar to Chapter 4) is mutable arrays of floats that can be created, read from, written to and deleted (each at differing levels of access). This allows us to illustrate one of the key practical benefits of ownership, which is that unique access to an owned value allows

for safe mutation (the original pun behind "Linear Types can Change the World" [142]). The second (new) example will be polymorphic mutable references which store a pointer to a value of any type. First, we extend our calculus with a primitive type of resources, along with some additional base types used to parameterise said resources:

$$A ::= \ldots \mid Res_{id}\ A \mid \mathbb{N} \mid \mathbb{F}$$

$$Res_{id}\ A ::= \mathsf{Array}_{id}\ A \mid \mathsf{Ref}_{id}\ A \qquad \text{(types, extended)}$$

where $\mathbb{N}$ are natural numbers used for sizes and indices and $\mathbb{F}$ are floating-point numbers (which we treat as inherently non-linear, much as the `f32` and `f64` types implement the `Copy` trait in Rust), and resources *Res* (ranging over arrays or references for our purposes) are indexed by an identifier *id*. Our extended interface for mutable arrays provides the following primitives (with built-in weakening in the context):

$$\begin{aligned}
0 \cdot \Gamma \vdash\ & \textbf{newArray} && : \mathbb{N} \multimap \exists id.*(\mathsf{Array}_{id}\ \mathbb{F}) \\
0 \cdot \Gamma \vdash\ & \textbf{readArray} && : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap (\mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})) \\
0 \cdot \Gamma \vdash\ & \textbf{writeArray} && : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \\
& && \quad (\textit{where}\ p \equiv 1 \ \vee\ p \equiv *) \\
0 \cdot \Gamma \vdash\ & \textbf{deleteArray} && : *(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathsf{unit}
\end{aligned}$$

where, unless bound, *id* and *p* are metavariables for identifier types and permissions respectively. Thus, we treat these primitives as part of the typing rules rather than functions that are in scope. Note that the implementation allows for full universal quantification over identifiers and permissions, enabling polymorphism; for example, the Granule type signature for **readArray** is as follows.

```
readFloatArray : ∀ {p : Fraction, id : Name}
  . & p (FloatArray id) → Int → (Float, & p (FloatArray id))
```

Granule

Note that the kind `Fraction` includes both fractional permissions and also the special permission representing unique ownership. This interface resembles Granule's existing interface for array mutation using modal uniqueness types described in Chapter 4, but with some crucial differences owing to the introduction of borrowing. The previous interface is presented below for comparison.

$$\begin{aligned}
[\Gamma] \vdash\ & \textbf{newArray} && : \mathbb{N} \multimap *(\mathsf{Array}\ \mathbb{F}) \\
[\Gamma] \vdash\ & \textbf{readArray} && : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F}) \\
[\Gamma] \vdash\ & \textbf{writeArray} && : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F}) \\
[\Gamma] \vdash\ & \textbf{deleteArray} && : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathsf{unit}
\end{aligned}$$

First, array types are now indexed by an identifier pointing to a particular array object in the heap, generated by the existential type in

newArray and within the scope of the CLONE rule. This allows us to keep track of which array is being referenced now that multiple borrows pointing to the same array are permitted to exist. The other key difference is that reading and writing no longer require unique ownership: they are both permitted for borrows with varying permission levels. Reading can be carried out at any permission, since this is safe no matter how many references exist, but writing is restricted to either a mutably borrowed or uniquely owned array.

The following example illustrates the application of these various primitives in Granule and demonstrates the usage of existentially typed identifiers from the perspective of the programmer.

```
exampleArray : Float
exampleArray = unpack <id , a> = newFloatArray 3 in
                  let a'       = writeFloatArray a 1 4.2;
                      (f, a'') = readFloatArray a' 1;
                      ()       = deleteFloatArray a'' in f
```
Granule

As per the typing rules of existentials presented in Section 5.3, once we unpack the existential in the second line the type variable `id` must not occur in the type of the body. In this example, the array is deleted and never returned in the body's result, thus satisfying the type restrictions of `unpack`. We can also apply the `clone` construct to make a deep copy of a unique array as described in Section 5.3:

```
exampleClone : ()
exampleClone = unpack <id , a> = newFloatArray 3 in
                 clone (share a) as x in
                   unpack <id' , a'> = x in (deleteFloatArray a')
```
Granule

The interface for polymorphic references is similar, with primitives:

$$
\begin{array}{lll}
0 \cdot \Gamma \vdash & \textbf{newRef} & : A \multimap \exists id.*(\mathsf{Ref}_{id}\ A) \\
0 \cdot \Gamma \vdash & \textbf{readRef} & : \&_p(\mathsf{Ref}_{id}\ (\Box_{r+1} A)) \multimap A \otimes \&_p(\mathsf{Ref}_{id}\ (\Box_r A)) \\
0 \cdot \Gamma \vdash & \textbf{swapRef} & : \&_p(\mathsf{Ref}_{id}\ A) \multimap A \multimap A \otimes \&_p(\mathsf{Ref}_{id}\ A) \\
 & & (\textit{where}\ p \equiv 1 \ \vee \ p \equiv *) \\
0 \cdot \Gamma \vdash & \textbf{freezeRef} & : *(\mathsf{Ref}_{id}\ A) \multimap A
\end{array}
$$

The crucial differences follow from the value encapsulated in the reference being of any type, rather than only floats which are inherently duplicable and discardable. Thus, **swapRef** must enforce linear usage for both the existing and new values. Similarly **freezeRef** must return the value in the reference to obey linearity. The next example illustrates this interface, by creating a reference to a float, updating the value and then deleting the reference.

```
exampleReference : (Float, Float)
exampleReference = unpack <id , ref> = newRef 0.0 in
                     let (x, ref')  = swapRef ref 42.0;
                                  y = freezeRef ref' in (x, y)
```
Granule

In order to allow for non-linear behaviours, we also provide a **readRef** primitive which requires the value to be of a graded type which permits the additional usage accrued with the appropriate type-level accounting; this can be applied as follows. Note that the inline type signature for the tuple on the second line of `exampleGraded` is only needed because Granule's existing type inference is quite weak, and as such needs to be supplied with additional information.

```
test : Float [6]
test = [42.0]

exampleGraded : (Float [4], Float [2])
exampleGraded =  unpack <id , r> = newRef test in
 let ([x], r') : (Float [4], ∗(Ref id (Float [2]))) = readRef r;
                 [y] : Float [2] = freezeRef r' in ([x], [y])
```
Granule

One important benefit of introducing polymorphic references is that they allow standard Granule values such as integers or floating-point numbers to be treated uniquely and managed by the ownership system, since otherwise this would be prevented by the cloneable predicate described in Section 5.3. This allows for the use of general imperative programming patterns as might be applied in languages like Rust (where ownership is the default) to be transferred to the setting of Granule (where functional patterns are more typical). Our references have more safety than traditional ML-style references when values can be accessed through multiple references simultaneously.

We are now able to give the full inductive definition of the cloneable predicate, displayed below. As described in Section 5.3, this permits only resources and products containing resources to be cloned, ensuring that all cloneable values come with an identifier. Note that in order to preserve both linearity and this condition, for a polymorphic reference to be cloneable it must itself point either to a cloneable value or to a value that is freely copyable, such as a floating-point number.

Definition 5 (Copyable predicate)**.** *Predicate definition:*

$$\frac{}{\mathsf{copyable}(\mathsf{unit})} \qquad \frac{}{\mathsf{copyable}(\mathbb{N})} \qquad \frac{}{\mathsf{copyable}(\mathbb{F})}$$

$$\frac{\mathsf{copyable}(A) \quad \mathsf{copyable}(B)}{\mathsf{copyable}(A \otimes B)}$$

Definition 6 (Cloneable predicate)**.** *Predicate definition:*

$$\frac{}{\mathsf{cloneable}(\mathsf{Array}_{id}\ \mathbb{F})} \qquad \frac{\mathsf{cloneable}(A)\ \vee\ \mathsf{copyable}(A)}{\mathsf{cloneable}(\mathsf{Ref}_{id}\ A)}$$

$$\frac{\mathsf{cloneable}(A) \quad \mathsf{cloneable}(B)}{\mathsf{cloneable}(A \otimes B)}$$

RESOURCE ALLOCATORS Here, the restriction introduced on the promotion rule in Section 6.2 meaning that 'resource allocators' are forbidden from being promoted becomes crucial for ensuring soundness in the call-by-value setting of this work. Consider the following pseudocode (eliding packing and unpacking of existentials), which would be allowed in Granule given unrestricted promotion:

```
let [x] : ((*(Array id Float)) [2]) = [newArray 1] in
let () = deleteArray x in writeArray x 0 1.0
```

Granule ✗

On the first line, this program creates a reference `x` to a new array of size 1, but under a promotion, with the type explaining that we want to use the resulting value twice (given by the explicit type signature here). This promotion and subsequent unboxing then allows two uses of the array on the second line.

Under a call-by-name semantics, as in Chapter 4 (and accessible in Granule via the extension `language CBN`), this program executes successfully and produces an array which contains the value written on line 2. The key is that call-by-name reduction substitutes the call to `newArray` into the two uses of the variable `x`, and so these point to two entirely separate arrays. However, under the call-by-value semantics of this chapter (and Granule's default, originally chosen for simplicity and to avoid pitfalls caused by laziness with side effects), the first line is fully evaluated, and so both uses of `x` point to the same array. Thus the second line attempts to write to an array after it has been deleted, which would cause a runtime error. In a call-by-value setting this program must not be permitted.

The solution we apply here (also used in Chapter 6 and Chapter 7) is to syntactically restrict promotion to terms which do not allocate resources, i.e., that do not use **newArray** or **newRef** in a reduction position. The predicate $\mathsf{resourceAllocator}(t)$ classifies precisely these terms, as a kind of specialised "value restriction"; note in particular that $\neg\mathsf{resourceAllocator}(\lambda x.\mathbf{newArray}\ t)$ since reduction does not happen underneath an abstraction (Section 5.5 defines the reduction semantics). The full inductive definition is given below.

Definition 7 (Resource allocating terms)**.** *Predicate definition:*

$$\frac{}{\mathsf{resourceAllocator}(\textbf{\textit{newRef}})} \qquad \frac{}{\mathsf{resourceAllocator}(\textbf{\textit{newArray}})}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(t_1\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}((\lambda x.t_1)\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ () = t_1\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ () = t_1\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ (x, y) = t_1\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ (x, y) = t_1\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}((t_1, t_2))} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}((t_1, t_2))}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ [x] = t_1\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ [x] = t_1\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t)}{\mathsf{resourceAllocator}([t])} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{share}\ t_1)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{clone}\ t_1\ \textbf{\textit{as}}\ x\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{clone}\ t_1\ \textbf{\textit{as}}\ x\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{withBorrow}}\ t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{withBorrow}}\ t_1\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{split}}\ t_1)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{join}}\ t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{join}}\ t_1\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{push}\ t_1)} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{pull}\ t_1)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{pack}}\ \langle id', t_1\rangle)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{unpack}}\ \langle id', x\rangle = t_1\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{unpack}}\ \langle id', x\rangle = t_1\ \textbf{\textit{in}}\ t_2)}$$

Other work resolves this same problem relating to promotion of resource allocators through different techniques. Originally, the linear types extension to Haskell got around this difficulty by only ever allocating resources inside a specialised continuation, which is passed around at every step until deallocation to prevent the allocator itself from ever being used non-linearly. See the interface for mutable arrays in Linear Haskell below, reproduced from the original paper [19].

```haskell
newMArray :: Int -> (MArray a ⊸ Unrestricted b) ⊸ b
write     :: MArray a ⊸ (Int, a) -> MArray a
read      :: MArray a ⊸ Int -> (MArray a, Unrestricted a)
freeze    :: MArray a ⊸ Unrestricted (Array a)
```

Haskell

Note in particular the continuation-passing style employed in the `newMArray` function. More recently, this strategy has been generalised by introducing a "linear constraint" [129] called `Linearly`. This constraint, which must itself be used in a linear fashion, is assumed whenever a new resource is allocated. A continuation is still necessary for the initial assumption of `Linearly`, but the same qualification may now be used generically for varying resource types.

5.4.1 Equational Theory

The withBorrow typing suggests a close relationship between $\&_* A$ and $\&_1 A$. By analogy to the notion of a relative monad used in Chapter 4, $\&_* A$ acts as a 'relative functor' with regard to $\&_1 A$, where **withBorrow** equates to mapping a function involving mutable borrows onto a function between uniquely owned values. The following axioms for (relative) functors then hold:

$$\textbf{withBorrow}\ (\lambda x.x)\ t \equiv t \qquad \text{(\&unit)}$$

$$\textbf{withBorrow}\ (\lambda x.f\ (g\ x))\ t \equiv \textbf{withBorrow}\ f\ (\textbf{withBorrow}\ g\ t) \qquad \text{(\&assoc)}$$

The first axiom states that borrowing a reference to an owned value and simply returning it via the identity function without making use of it is equivalent to doing nothing at all, as one might expect. The second axiom is an associativity axiom, giving us the result that borrowing a value to apply one function and then borrowing the value again to apply a second function is equivalent to simply borrowing the value once and applying the functions in sequence.

Lastly, **split** and **join** additionally have an isomorphism:

$$(\textbf{let}\ (x, y) = (\textbf{split}\ t)\ \textbf{in}\ (\textbf{join}\ x\ y)) \equiv t \qquad \text{(\&rejoin)}$$

$$\textbf{split}\ (\textbf{join}\ t_1\ t_2) \equiv (t_1, t_2) \qquad \text{(\&resplit)}$$

5.4.2 Partial Views via Distributive Laws

One particularly useful borrowing pattern when writing practical programs is the ability to take a composite data structure and borrow only *part* of it, such that the original owner retains access to the remaining structure. This introduces the possibility of multiple threads working with parts of a data structure in parallel, where it is no longer necessary to take ownership of the full data structure in order to mutate the only part for which access is required.

Our core calculus provides some capacity for structured data in the form of product types. We introduce two additional constructs for distributing the borrowing graded modality into and out of products, enabling the pattern of partial borrowing:

$$\frac{\Gamma \vdash t : \&_p(A \otimes B)}{\Gamma \vdash \mathbf{push}\, t : (\&_p A) \otimes (\&_p B)}\ \textsc{push}$$

$$\frac{\Gamma \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma \vdash \mathbf{pull}\, t : \&_p(A \otimes B)}\ \textsc{pull}$$

We can derive an operation akin to **withBorrow** for borrowing one component of a product: use **push** to move ownership onto the values inside the product, use **withBorrow** to borrow the required component, and then once we are done, use **pull** to recover the original pair. We demonstrate this concept in action with the following example. We assume a function `alter : & 1 Int → & 1 Int` which operates over borrowed components of `Colour` and requires a whole permission, in order to illustrate the idea. We follow the above scheme for partial borrowing:

```
examplePartial : *Colour → *Colour
examplePartial indigo = let (r, p) = push indigo in
                        let     r' = withBorrow alter r in
                        pull (r', p)
```

Granule

This notion extends gracefully to structures containing more than two components. For example, to borrow a single element from a unique tuple of three values, leaving the remaining two values available to their original owner, we can use nested products, e.g. $\&_*(A \otimes (B \otimes C))$, and **push** and **pull** to then access just one part, similarly to in `indigo`. We may need to **push** twice in order to distribute the modality over the entire product depending on which component we are borrowing, but in the same way as above, we can use **withBorrow** to extract the desired data (for example, $\&_1 A$) while applying **pull** to recover the remaining structure (in this case, of type $\&_*(B \otimes C)$).

Partial borrowing allows safe mutation of disjoint parts of a single data structure *concurrently* without risking a data race. The following example illustrates this using a `par` function for spawning two concurrent threads; we leave this as primitive for now, but it will be defined later in Chapter 6.

```
par : ∀ {a b : Type} . (() → a) → (() → b) → (a, b)
    -- Granule's parallel composition

exampleConcurrent : *Colour → *Colour
exampleConcurrent indigo = let (r, p) = push indigo in
                           let (g, b) = push p in
                           let (r', b') = par (λ() →
                                     withBorrow alter r)
                                              (λ() →
                                     withBorrow alter b) in
                           let     p' = pull (g, b') in
                           pull (r', p')
```

Granule

Here, the first couple of lines push the borrow all the way into the data structure of type `Colour`. Then, the `par` construct is used to mutate two components of the `Colour` in parallel, before the modality is pulled back out in order to reconstruct the original structure.

Building on this core concept in the more expressive setting of Granule's full type system could involve notions such as borrowing from any algebraic data type, or borrowing a 'slice' of multiple values. One low-hanging fruit is to extend the `push` and `pull` primitives in order to derive generic operations for arbitrary data structures, following patterns described in prior work [65]. We do not formalise this again here, but the approach allows for writing programs like:

```
exampleDeriving : ∀ {p : Fraction} . & p (Maybe Colour)
    → Either () (& p Colour)
exampleDeriving octarine = case (push@Maybe octarine) of
                             Nothing → Left ();
                             Just x  → Right x
```

Granule

Here, the borrow modality is distributed into the `Maybe` data type using a derived `push` operation. This general pattern for partial borrowing relates closely to McBride's notion of computing the *derivative* of a data type by finding its type of one-hole contexts [93], where the derivative describes the structure that remains after borrowing one element. Under this interpretation, borrowing a slice of multiple values would be akin to taking the derivative of a data type *with respect to* another type—an idea that is outside the scope of this thesis to explore further, but which we have discussed elsewhere [90].

PARALLEL SUM EXAMPLE To conclude this section, we present one last example which combines all of the elements we have introduced here in a more practical setting. The result will be a function which splits a unique mutable array of floats into two immutably borrowed halves and computes their sums concurrently, returning the total sum inside a polymorphic reference.

First, in order to implement this function elegantly we will need two additional basic functions for working with arrays and references.

Within Granule it is possible to derive a `writeRef` operation which destructively mutates the value inside a reference, as long as the type of said value allows for it to be discarded freely; this makes use of Granule's existing mechanism for deriving a `drop` operation (of type `t → ()`) for `Droppable` values [65]. Granule also provides an additional primitive for taking the length of an array, which is useful for iteration but was elided from our formal calculus for brevity. The types of both of these operations are given below.

```
lengthFloatArray : ∀ {id : Name, p : Fraction} .
    & p (FloatArray id) → (!Int, & p (FloatArray id))
writeRef         : ∀ {id : Name, a : Type} .
   {Droppable a} ⇒  a → & 1 (Ref id a) → & 1 (Ref id a)
```

Granule

where `!t` is a synonym for `t [Many]` representing arbitrary use via the $\{0, 1, \omega\}$-semiring (as was used heavily in Chapter 4), and `Droppable` is a predicate classifying types whose values can be discarded. We then define an auxiliary function for iterating through an array of floats and summing all values between two given indices:

```
sumFromTo : ∀ {id : Name, p : Fraction} .
    & p (FloatArray id) → !Int → !Int → (Float,
    & p (FloatArray id))
sumFromTo array [i] [n] =
  if i == n then (0.0, array)
    else
      let (x, a) = readFloatArray array i;
          (y, arr) = sumFromTo a [i+1] [n]
      in  (x + y, arr)
```

Granule

Finally, we define the core `parSum` function as described above. Note in particular the usage of `withBorrow` to borrow an initial reference to the mutable array, `split` and `join` to manage the forking dataflow when computing the sums for the two separate halves in parallel, and `push` and `pull` to distribute modalities into and out of products:

```
parSum : ∀ {id id' : Name} . *(FloatArray id) → *(Ref id' Float)
                           → *(Ref id' Float, FloatArray id)
parSum array ref = let
  ([n], array) : (!Int, *(FloatArray id))
    = lengthFloatArray array;
  compIn
    = pull (ref, array)
  in withBorrow (λcompIn →
                let (ref, array)     = push compIn;
                    (array1, array2) = split array;
            -- Immutable borrow happens here

            -- Compute in parallel
                    ((x, array1), (y, array2)) =
                                par (λ() →
                                  sumFromTo array1 [0] [div n 2])
                                    (λ() →
                                  sumFromTo array2 [div n 2] [n]);

            -- Update the reference
                    ref'    = writeRef ((x : Float) + y) ref;
                    compOut = pull (ref', join (array1, array2))
            -- End immutable borrow

                  in compOut) compIn
```

Granule

The full code of this example is given in Appendix c, including full definitions for all auxiliary functions and an illustration of how this function could be applied in practice to a freshly created pair of an array (which is generated from a length-indexed vector) and a polymorphic reference.

5.5 SEMANTICS AND METATHEORY

We define an operational semantics here for the core calculus which serves to further explain the details of arrays, mutation, copying, and borrowing. We follow the pattern of Chapter 4, adapting the approach of Choudhury et al. [32] for giving an operational semantics to a graded system, with some degree of accounting for grades and references in order to relate the dynamic semantics back to the static semantics of typing (Section 5.5.3). With the exception of Granule and Multi-Graded Featherweight Java [20], much of the preceding work on operational models for graded systems is based on call-by-name, including the development of Chapter 3. Going forwards we move to call-by-value, for the purpose of describing real-world practical functional languages; while call-by-name was useful for illustrating the principles of uniqueness in a simpler setting that is closer to typical semantics for linear calculi and does not require additional complications, call-by-name is often prohibitively expensive with unpredictable performance and poor interaction with side effects.

5.5.1 Preliminary Definitions

VALUES We first define the subset of terms that are *values* in the semantics, i.e., normal forms (terms that have no further reduction), via the grammar:

$$v ::= (v_1, v_2) \mid () \mid [v] \mid \lambda x.t \mid n \mid p \mid \textbf{pack}\ \langle id, v \rangle$$

(value terms sub-grammar)

i.e., pairs of values, the unit value, boxed values, abstractions, natural numbers n, primitives p which may also be partially applied to other values, e.g., **newArray**, **readArray**, **readArray** v and so on, and also existential values.

RUNTIME TERMS AND TYPING We extend the syntax of terms (and values) with several runtime representations which appear only in the semantics, i.e., they cannot be written by users in programs:

$$t ::= \ldots \mid *t \mid \textbf{unborrow}\ t \mid \mathit{ref} \mid [t]_r \qquad \text{(runtime terms)}$$
$$v ::= \ldots \mid *v \mid \textbf{unborrow}\ v \mid \mathit{ref} \mid [v]_r \qquad \text{(runtime values)}$$

where $*t$ represents unique and borrowed terms, **unborrow** t is the inverse to borrowed terms (used to implement **withBorrow**), and *ref* are polymorphic references to resources bound in the heap. Furthermore, the syntax for promotion is augmented with an annotation $[t]_r$ of the grade r at which the term t is promoted, i.e., Church-style with respect to grades, and thus typing this grade-annotated version of promotion produces $r \cdot \Gamma \vdash [t]_r : \Box_r A$. This annotation is only needed to prove that type preservation respects resourcing (as per Choudhury et al. [32]) and can be ignored when actually computing the reduction behaviour of a term using the operational model.

In order to type runtime terms with resource references, the syntax of contexts is extended to include assumptions $\mathit{ref} : \mathit{Res}_{id}\ A$ which are treated as a different syntactic category of variables. A *runtime context* γ is a context containing only these references, as follows.

$$\gamma ::= \emptyset \mid \gamma, \mathit{ref} : \mathit{Res}_{id}\ A$$

Context scalar multiplication and addition extend as follows (eliding a symmetric case for brevity):

$$(\Gamma, \mathit{ref} : \mathit{Res}_{id}\ A) + (\Gamma', \mathit{ref} : \mathit{Res}_{id}\ A) = (\Gamma + \Gamma'), \mathit{ref} : \mathit{Res}_{id}\ A$$

(runtime context addition)

$$(\Gamma, \mathit{ref} : \mathit{Res}_{id}\ A) + \Gamma' = (\Gamma + \Gamma'), \mathit{ref} : \mathit{Res}_{id}\ A \quad (\textit{where ref} \notin \Gamma')$$
$$r \cdot (\Gamma, \mathit{ref} : \mathit{Res}_{id}\ A) = (r \cdot \Gamma), \mathit{ref} : \mathit{Res}_{id}\ A$$

(runtime context multiplication)

Importantly, references are not treated linearly here since they may be shared in the runtime context (such as via the **split** operation). Runtime terms are typed by the following rules:

$$\frac{\gamma \vdash t : A}{0 \cdot \Gamma, \gamma \vdash *t : \&_p A}\ \text{NEC} \qquad \frac{\Gamma \vdash t : \&_1 A}{\Gamma \vdash \mathbf{unborrow}\ t : *A}\ \text{UNBORROW}$$

$$\frac{}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash ref : Res_{id}\ A}\ \text{REF}$$

HEAPS AND CONFIGURATIONS Our semantics again does not use syntactic substitution; instead *heaps* map program variables to values, resource references to identifiers *id*, and identifiers to resource values:

$$H ::= \emptyset \mid H, x \mapsto_r v \mid H, ref \mapsto_p id \mid H, id \mapsto v_r \qquad \text{(heaps)}$$
$$v_r ::= \mathbf{arr} \mid \mathbf{ref}(v) \qquad \mathbf{arr} ::= \mathsf{init} \mid \mathbf{arr}[n] = v \quad \text{(heap resource terms)}$$

Thus, a heap can be extended in three ways: (1) with an assignment of a program variable x to a value v, storing the grade of the variable r; (2) with an assignment of a resource reference *ref* to an identifier *id* with permission p; (3) with an assignment of an identifier *id* to a resource term. Identifiers and references are separated since we may need multiple references to point to the same identifier, e.g., in the case of immutable borrows. One can think of identifiers like abstract memory addresses in the semantics.

Array terms **arr** in the heap are either an empty array init or an array storing value v at index n. Reference values $\mathbf{ref}(v)$ in the heap store a value v. These runtime resource terms in the heap are typed according to straightforward rules presented in Appendix A.

A *configuration* comprises a pair of a *heap* H and a term t, written as $H \vdash t$, where $\mathsf{fv}(t) \subseteq \mathsf{dom}(H)$, and $\mathsf{refs}(t) \subseteq \mathsf{dom}(H)$ (where refs is the set of all resource references *ref* in the term).

5.5.2 Single-Step Reduction

Single-step reductions map source configurations to target configurations, with the judgment:

$$H_1 \vdash t_1 \rightsquigarrow_s H_2 \vdash t_2$$

where H_1 and H_2 are input and output heaps respectively, t_1 is the source term and t_2 the target. The grade s denotes the *usage context* of this rule. We explain the reduction rules for our operational semantics in detail. Throughout, $x \# \bar{t}$ means that x is a fresh name with respect to some terms $\bar{t}$.

LAMBDA CALCULUS The λ-calculus core of the operational semantics has rules:

$$\frac{\exists r'.\ s + r' \sqsubseteq r}{H, x \mapsto_r v \vdash x \ \rightsquigarrow_s \ H, x \mapsto_r v \vdash v} \ \rightsquigarrow_{\text{VAR}}$$

$$\frac{y \# \{H, v, t\}}{H \vdash (\lambda x.t)\, v \ \rightsquigarrow_s \ H, y \mapsto_s v \vdash t[y/x]} \ \rightsquigarrow_{\beta}$$

$$\frac{H \vdash t_1 \ \rightsquigarrow_s \ H' \vdash t_1'}{H \vdash t_1\, t_2 \ \rightsquigarrow_s \ H' \vdash t_1'\, t_2} \ \rightsquigarrow_{\text{APPL}} \qquad \frac{H \vdash t_2 \ \rightsquigarrow_s \ H' \vdash t_2'}{H \vdash v\, t_2 \ \rightsquigarrow_s \ H' \vdash v\, t_2'} \ \rightsquigarrow_{\text{APPR}}$$

In the $\rightsquigarrow_{\text{VAR}}$ rule, a variable x is reduced to a value v which was assigned to x in the heap. The grade r is preserved in the output heap, with the side condition in the premise ensuring that the grade r will be enough to capture the usage s required by the reduction. In the $\rightsquigarrow_{\beta}$ rule, rather than using a substitution to enact β-reduction, the resulting term is the body of the function t with the heap extended with a freshened binder y assigned to the argument value v, annotated by the grade s which parameterises the reduction. The body term is appropriately renamed to use the freshened variable y (to prevent variable capture issues). The remaining two reductions are congruences, giving left to right reduction.

EXISTENTIAL TYPES AND NAMES The semantics of existentials is standard, with a beta rule:

$$\frac{y \# \{H, v, t\}}{H \vdash \mathbf{unpack}\ \langle id, x\rangle = \mathbf{pack}\ \langle id', v\rangle\ \mathbf{in}\ t \ \rightsquigarrow_s \ H, y \mapsto_r v \vdash t[y/x]} \ \rightsquigarrow_{\exists\beta}$$

and two standard congruence rules for pack and unpack (elided for brevity).

TENSORS AND UNITS Tensor products have the following rules for their introduction and elimination forms, with three congruence rules and one β-rule:

$$\frac{H \vdash t_1 \ \rightsquigarrow_s \ H' \vdash t_1'}{H \vdash (t_1, t_2) \ \rightsquigarrow_s \ H' \vdash (t_1', t_2)} \ \rightsquigarrow_{\otimes\text{L}} \qquad \frac{H \vdash t_2 \ \rightsquigarrow_s \ H' \vdash t_2'}{H \vdash (v, t_2) \ \rightsquigarrow_s \ H' \vdash (v, t_2')} \ \rightsquigarrow_{\otimes\text{R}}$$

$$\frac{H \vdash t_1 \ \rightsquigarrow_s \ H' \vdash t_1'}{H \vdash \mathbf{let}\,(x, y) = t_1\ \mathbf{in}\ t_2 \ \rightsquigarrow_s \ H' \vdash \mathbf{let}\,(x, y) = t_1'\ \mathbf{in}\ t_2} \ \rightsquigarrow_{\text{LET}\otimes}$$

$$\frac{x' \# \{H, v_1, v_2, t\} \qquad y' \# \{H, v_1, v_2, t\}}{H \vdash \mathbf{let}\,(x, y) = (v_1, v_2)\ \mathbf{in}\ t \ \rightsquigarrow_s \ H, x' \mapsto_s v_1, y' \mapsto_s v_2 \vdash t[y'/y][x'/x]} \ \rightsquigarrow_{\otimes\beta}$$

In the case of ($\rightsquigarrow_{\otimes\beta}$), we extend the heap with assignments for (fresh names) x' and y' to v_1 and v_2 respectively, continuing on with

the body term t. Similarly to the β-rule for functions, the freshening avoids variable capture. The rules for the unit type are similar:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\ () = t_1\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\ () = t_1'\ \mathbf{in}\ t_2}\ \rightsquigarrow_{\text{LETUNIT}}$$

$$\frac{}{H \vdash \mathbf{let}\ () = ()\ \mathbf{in}\ t \rightsquigarrow_s H \vdash t}\ \rightsquigarrow_{\beta\text{UNIT}}$$

Example 8. The following gives a reduction sequence for the term $(\lambda x.(x,y))\,(v,y)$ from Example 6 under a heap $H = y{\mapsto_2}v$, until a normal form is reached:

$$\begin{aligned}
& y{\mapsto_2}v \vdash (\lambda x.(x,y))\,(v,y) \\
(\rightsquigarrow_{\text{APPL}}, \rightsquigarrow_{\otimes\text{R}}, \rightsquigarrow_{\text{VAR}}) \rightsquigarrow_1\ & y{\mapsto_2}v \vdash (\lambda x.(x,y))\,(v,v) \\
(\rightsquigarrow_{\beta}) \rightsquigarrow_1\ & y{\mapsto_2}v, x'{\mapsto_1}(v,v) \vdash (x',y) \\
(\rightsquigarrow_{\text{APPL}}, \rightsquigarrow_{\otimes\text{L}}, \rightsquigarrow_{\text{VAR}}) \rightsquigarrow_1\ & y{\mapsto_2}v, x'{\mapsto_1}(v,v) \vdash ((v,v),y) \\
(\rightsquigarrow_{\text{APPL}}, \rightsquigarrow_{\otimes\text{R}}, \rightsquigarrow_{\text{VAR}}) \rightsquigarrow_1\ & y{\mapsto_2}v, x'{\mapsto_1}(v,v) \vdash ((v,v),v)
\end{aligned}$$

GRADED MODALITIES The rules for graded modalities are structured similarly to the standard lambda calculus rules above, but we need to do some additional management of grades in the heap.

$$\frac{H \vdash t \rightsquigarrow_{s*r} H' \vdash t'}{H \vdash [t]_r \rightsquigarrow_s H' \vdash [t']_r}\ \rightsquigarrow_{\Box}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{LET}\Box}$$

$$\frac{y\#\{H,v,t\}}{H \vdash \mathbf{let}\,[x] = [v]_r\,\mathbf{in}\,t \rightsquigarrow_s H, y{\mapsto_{(s*r)}}v \vdash t[y/x]}\ \rightsquigarrow_{\Box\beta}$$

In the $\rightsquigarrow_{\Box}$ rule, to construct a reduction with the required grade s then we need to be able to reduce inside the box at grade $s * r$, to account for the additional usage required by the r-graded modality. In the $\rightsquigarrow_{\Box\beta}$ rule, the x in the heap is annotated not only with s as in the regular $\rightsquigarrow_{\beta}$ rule but with $s * r$, again to account for the additional usage the modality requires.

ARRAYS For brevity, we show here in the body of the chapter only the reduction rules for arrays; the corresponding (and very similar) rules for polymorphic references are included in Appendix c. For simplicity, we do not explicitly track the sizes of arrays in our semantics.

New arrays are created by the **newArray** primitive, with reduction:

$$\frac{ref\#H \qquad id\#H}{H \vdash \mathbf{newArray}\,n \rightsquigarrow_s H, ref{\mapsto_1}id, id \mapsto \mathsf{init} \vdash \mathbf{pack}\,\langle id, *ref\rangle}\ \rightsquigarrow_{\text{NEWARRAY}}$$

where ref and id are fresh for the heap H. The resulting array is initialised to the empty array (init), and the result is a unique array

reference $*ref$ packed in an existential over id. In the heap, ref is marked with the whole permission 1.

Arrays are read and written via:

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{arr}[n] = v \vdash \mathbf{readArray}\,(*ref)\,n \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{arr}[n] = v \vdash (v, *ref)}\ \rightsquigarrow_{\text{READARRAY}}$$

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\,(*ref)\,n\,v \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{arr}[n] = v \vdash *ref}\ \rightsquigarrow_{\text{WRITEARRAY}}$$

For **writeArray**, p should be 1 or $*$, but this is mediated by the type system rather than being enforced in the semantics; the following rule is similar in this regard for deleting unique arrays:

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{arr} \vdash \mathbf{deleteArray}\,(*ref) \rightsquigarrow_s H \vdash ()}\ \rightsquigarrow_{\text{DELETEARRAY}}$$

SHARING AND CLONING Sharing reduces permissions to 0 and cloning involves copying heap terms:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{share}\,t \rightsquigarrow_s H' \vdash \mathbf{share}\,t'}\ \rightsquigarrow_{\text{SHARE}}$$

$$\frac{\mathsf{dom}(H) \equiv \mathsf{refs}(v)}{H, H' \vdash \mathbf{share}\,(*v) \rightsquigarrow_s ([H]_0), H' \vdash [v]}\ \rightsquigarrow_{\text{SHAREB}}$$

$$\frac{\mathsf{dom}(H') \equiv \mathsf{refs}(v) \qquad (H'', \theta, \overline{id}) = \mathsf{copy}(H') \qquad y \# \{H, v, t\}}{H, H' \vdash \mathbf{clone}\,[v]_r\,\mathbf{as}\,x\,\mathbf{in}\,t \rightsquigarrow_s H, H', H'', y \mapsto_s \mathbf{pack}\,\langle \overline{id}, *(\theta(v)) \rangle \vdash t[y/x]}\ \rightsquigarrow_{\text{CLONEB}}$$

In the $\rightsquigarrow_{\text{SHAREB}}$ rule, the incoming heap is split into two parts, where H is such that it provides the allocations for all resource references in v (enforced by the premise). The unique value $*v$, once unwrapped, is wrapped in the graded box modality in the result as $[v]$, and thus all its references are now annotated with 0 in the heap via $([H]_0)$, e.g.:

$$H', ref \mapsto_1 id, id \mapsto v_r \vdash \mathbf{share}\,(*ref) \rightsquigarrow H', ref \mapsto_0 id, id \mapsto v_r \vdash [ref]$$

This means that uniqueness and borrowing information no longer needs to be tracked for the value going forwards in the reduction, since we know it will no longer have any such guarantees offered by the type system.

The $\rightsquigarrow_{\text{CLONEB}}$ rule enacts a 'deep copy', where $\mathsf{dom}(H') \equiv \mathsf{refs}(v)$ marks the part of the heap with resource references coming from v. Then $\mathsf{copy}(H')$ copies the resources in this part of the heap, creating a heap fragment H'' and a renaming operator θ which maps from old references to new copied references. This renaming is applied to v in the freshly bound variable y, such that the value $*(\theta(v))$ refers to any newly copied resources. Lastly, we pack the renamed unique value with new identifiers $\overline{id}$ generated by copy.

We elide the straightforward congruence rule for **clone**.

BORROWING We elide the congruence rules for **withBorrow** which ensure that we reduce the two argument terms left to right until they are values, after which we can reduce as follows:

$$\frac{y\#\{H, v, t\}}{H \vdash \textbf{withBorrow}\ (\lambda x.t)\ (*v) \rightsquigarrow_s H, y{\mapsto_s}(*v) \vdash \textbf{unborrow}\ t[y/x]}\ {\rightsquigarrow}_{\text{WITH\&}}$$

Here, the β-reduction that comes from applying the function $\lambda x.t$ to the value $*v$ is enacted by extending the heap with $*v$ which acts as a borrowed term now in the context of t. For a term to escape **withBorrow** it must eventually be 'unborrowed', encapsulated by the runtime term **unborrow** above, which then is itself reduced by the following rules:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \textbf{unborrow}\ t \rightsquigarrow_s H' \vdash \textbf{unborrow}\ t'}\ {\rightsquigarrow}_{\text{UNBORROW}}$$

$$\frac{}{H \vdash \textbf{unborrow}\ (*v) \rightsquigarrow_s H \vdash *v}\ {\rightsquigarrow}_{\text{UN\&}}$$

Aside from the congruence, which is standard, the ${\rightsquigarrow}_{\text{UN\&}}$ rule is philosophically 'unwrapping' a borrowed value and then treating it as unique once more, by wrapping with $*$, but the runtime term $*v$ is used to represent both unique and borrowed v values.

PUSH AND PULL The reduction rules for **push** and **pull** are fairly simple; they simply distribute the modality into or out of the product term in a way that matches the typing of the given rule.

$$\frac{}{H \vdash \textbf{push}\ (*(v_1, v_2)) \rightsquigarrow_s H \vdash (*v_1, *v_2)}\ {\rightsquigarrow}_{\text{PUSH}*}$$

$$\frac{}{H \vdash \textbf{pull}\ (*v_1, *v_2) \rightsquigarrow_s H \vdash *(v_1, v_2)}\ {\rightsquigarrow}_{\text{PULL}*}$$

The heap is left unchanged. Push and pull also have congruence rules which are straightforward.

SPLIT AND JOIN There are two primary reduction rules for each of **split** and **join**: one for the case where the terms are resource references and one for the case where the terms are pairs of values:

$$\frac{ref_1 \# H \quad ref_2 \# H}{H, ref \mapsto_p id, id \mapsto v \vdash \textbf{split}\ (*ref) \rightsquigarrow_s H, ref_1 \mapsto_{\frac{p}{2}} id, ref_2 \mapsto_{\frac{p}{2}} id, id \mapsto v \vdash (*ref_1, *ref_2)} \rightsquigarrow_{\text{SPLITREF}}$$

$$\frac{ref \# H}{H, ref_1 \mapsto_p id, ref_2 \mapsto_q id, id \mapsto v \vdash \textbf{join}\ (*ref_1)\ (*ref_2) \rightsquigarrow_s H, ref \mapsto_{(p+q)} id, id \mapsto v \vdash *ref} \rightsquigarrow_{\text{JOINREF}}$$

$$\frac{\begin{array}{c} H \vdash \textbf{split}\ (*v) \rightsquigarrow_s H' \vdash (*v_1, *v_2) \\ H' \vdash \textbf{split}\ (*w) \rightsquigarrow_s H'' \vdash (*w_1, *w_2) \end{array}}{H \vdash \textbf{split}\ (*(v, w)) \rightsquigarrow_s H'' \vdash (*(v_1, w_1), *(v_2, w_2))} \rightsquigarrow_{\text{SPLIT}\otimes}$$

$$\frac{\begin{array}{c} H \vdash \textbf{join}\ (*v_1)\ (*v_2) \rightsquigarrow_s H' \vdash *v \\ H' \vdash \textbf{join}\ (*w_1)\ (*w_2) \rightsquigarrow_s H'' \vdash *w \end{array}}{H \vdash \textbf{join}\ (*(v_1, w_1))\ (*(v_2, w_2)) \rightsquigarrow_s H'' \vdash *(v, w)} \rightsquigarrow_{\text{JOIN}\otimes}$$

In the reference case, **split** removes the initial reference from the heap and generates two fresh references pointing to the same identifier as the original reference. These are each annotated with half of the permission belonging to the original reference, to match the typing. As one might expect, **join** for references behaves dually; it deletes two existing references from the heap, and generates one fresh reference, with its permission being the sum of the constituent parts.

For pairs, the reduction rules are defined inductively: as long as we can split or join on the two components of the pair, we are allowed to reduce on the overall pair itself. In this way, we construct a reduction which ensures that all references contained within the pair are split or joined as required, no matter how deeply nested the pair may be.

Join and split also have straightforward and elided congruences.

MULTI-STEP REDUCTIONS Lastly, we define a relation that composes single-step reductions into a sequence of reductions, called a multi-reduction:

$$\frac{}{H \vdash t \Rightarrow_s H \vdash t}\ \textbf{REFL}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_2 \qquad H' \vdash t_2 \Rightarrow_s H'' \vdash t_3}{H \vdash t_1 \Rightarrow_s H'' \vdash t_3}\ \textbf{EXT}$$

5.5.3 Theorems

We now consider the relationship between typing and the operational semantics. First, we must build a foundation via checking some preliminary results and noting some crucial definitions.

Two useful results, which extend those found in Chapter 3, are that substitution is admissible for our calculus, coming in both linear and graded variants:

Lemma 8 (Linear substitution is admissible, extending Chapter 3)**.** *If* $\Gamma_1 \vdash t_1 : A$ *and* $\Gamma_2, x : A \vdash t_2 : B$ *then* $\Gamma_2 + \Gamma_1 \vdash t_2[t_1/x] : B$.

Lemma 9 (Graded substitution is admissible, extending Chapter 3)**.** *If* $[\Gamma_1] \vdash t_1 : A$ *and* $\Gamma_2, x : [A]_r \vdash t_2 : B$ *(where* $[\Gamma_1]$ *represents a context* Γ_1 *containing only graded assumptions) and* $\neg\mathsf{resourceAllocator}(t_1)$ *then* $\Gamma_2 + r \cdot \Gamma_1 \vdash t_2[t_1/x] : B$.

TYPE SAFETY Key to ensuring type safety is the notion of *heap compatibility with a typing context*, which refines the corresponding definition from Chapter 4.

Definition 8 (Heap compatibility)**.** *A heap* H *is compatible with free variable context* Γ*, denoted* $H \bowtie \Gamma$*, if the grades in the heap match those of the context, and any values stored in the heap have their resources accounted for in the rest of the heap. The relation is defined inductively over the syntax of heaps and contexts:*

$$\frac{}{\emptyset \bowtie \emptyset}\ \textsc{empty} \qquad \frac{H \bowtie \emptyset}{H, ref \mapsto_p id \bowtie \emptyset}\ \textsc{GCArr}$$

$$\frac{H, id \mapsto v_r \bowtie \Gamma + \gamma \qquad \gamma \vdash v_r : Res_{id}\ A}{H, ref \mapsto_p id, id \mapsto v_r \bowtie (\Gamma, ref : Res_{id}\ A)}\ \textsc{extRes}$$

$$\frac{H \bowtie \Gamma + s \cdot \Gamma' \qquad x \notin \mathsf{dom}(H) \qquad \Gamma' \vdash v : A \qquad \exists r'.\ s + r' \equiv r}{(H, x \mapsto_r v) \bowtie (\Gamma, x : [A]_s)}\ \textsc{ext}$$

$$\frac{H \bowtie \Gamma + \Gamma' \qquad x \notin \mathsf{dom}(H) \qquad \Gamma' \vdash v : A \qquad \exists r'.\ 1 + r' \equiv r}{(H, x \mapsto_r v) \bowtie (\Gamma, x : A)}\ \textsc{extLin}$$

*Thus, a context extended with a runtime type of a resource reference (*extRes*) is compatible with a heap which contains that reference ref, pointing to a resource term with the corresponding identifier id. In the premise of (*extRes*), the resource term with its identifier is preserved in the heap since there may be other references pointing to it (e.g., generated from a split). The (*GCArr*) rule then allows heap compatibility to 'garbage collect' any remaining resource references.*

*The (*ext*) rule says that a context with graded assumption* $x : [A]_s$ *is compatible with a heap as long as the heap contains a binding for* x *to some value* v *and heap grade* r *that can accommodate the usage of* s *(via the constraint* $\exists r'.\ s + r' \equiv r$*) and as long as the free variables* Γ' *of* v *are also*

*compatible with the heap (scaled by s to reflect the usage of x). The (*EXTLIN*) rule is similar to (*EXT*), but* effectively *where $s = 1$; the variable x is used in a linear fashion.*

Example 9. As an illustrative example, the context $x : [A]_1, y : [B]_2$ is compatible with the heap $x{\mapsto_7}v_1, y{\mapsto_2}v_2$ assuming the typing $x : [A]_3 \vdash v_2 : B$, with heap compatibility derivation:

$$\dfrac{\dfrac{\dfrac{}{\emptyset \bowtie \emptyset}\ \textsc{empty}}{\emptyset, x{\mapsto_7}v \bowtie x : [A]_7}\ \textsc{ext} \qquad x : [A]_3 \vdash v_2 : B}{(\emptyset, x{\mapsto_7}v_1, y{\mapsto_2}v_2) \bowtie (x : [A]_1, y : [B]_2)}\ \textsc{ext}$$

Heap compatibility allows us to establish key properties for our calculus in conjunction with the operational semantics. We begin with syntactic type safety, by verifying progress and preservation. These are largely standard, though preservation links with heap compatibility in its second conjunct.

Theorem 7 (Progress)**.** *Given $\Gamma \vdash t : A$, then t is either a value, or for all grades s and contexts Γ_0 then if $H \bowtie \Gamma_0 + s \cdot \Gamma$ there exists a heap H' and term t' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$.*

Theorem 8 (Preservation)**.** *For a well-typed term $\Gamma \vdash t : A$, under a restriction that polymorphic reference resources are restricted to non-function types, and for all s, Γ_0, and H such that $H \bowtie (\Gamma_0 + s \cdot \Gamma)$ and a reduction $H \vdash t \rightsquigarrow_s H' \vdash t'$, then we have:*

$$\exists \Gamma', H'.\ \ \Gamma' \vdash t' : A \ \ \wedge \ \ H' \bowtie (\Gamma_0 + s \cdot \Gamma')$$

Note the caveat to preservation: references $\mathsf{Ref}_{id}\ A$ are restricted such that A cannot be of function type, or some other composite type involving functions. The restriction is needed for preservation since it works at the granularity of a single reduction, and so cannot rule out the possibility that a reference is storing a λ term with free variables. This considerably complicates reasoning about resources and heaps, so we rule it out for this theorem. Importantly, this is not a restriction that needs to be made on the calculus and its implementation as a whole: for deterministic CBV reductions starting from a closed term (i.e., a complete program) then all β-redexes are on closed values and hence this problem does not exist in the context of an overall reduction sequence. However, to make preservation work for a single reduction, on potentially open terms, this minor restriction is needed locally. This does not affect any of the examples discussed in the paper.

In addition to preservation, Chapter 4 also proved a conservation property, ensuring that resource usage is respected. In particular, we showed that resource usage accrued in a given reduction plus remaining resources in the resulting heap are approximated by the

resources in the original heap plus the specified resource usage from any variable bindings encountered along the way. The semantics of Chapter 4 was call-by-name, which is more natural for conservation; checking that a modified form of conservation still holds in a call-by-value setting is future work, though various other graded type systems with similar properties are also call-by-value [22, 103]. Our inclusion of semiring grades in the heap here goes towards being able to extend the semantics to full accounting in future work, although they could be elided here and the above results would still be provable. The tracking of permission grades is however crucial to the next results.

We now move on to establishing perhaps the most interesting properties here, which are related to the safety of ownership and borrowing and how these notions relate to the references present in the heap. First, we ensure that taking a single step preserves the total of fractional permission annotations on references (unless we have stopped tracking ownership information for the given reference, in which case it should be annotated with 0):

Lemma 10 (Borrow safety)**.** *For a well-typed term $\Gamma \vdash t : A$ and all Γ_0, s, and heaps H such that $H \bowtie (\Gamma_0 + s \cdot \Gamma)$, and given a single-step reduction $H \vdash t \rightsquigarrow_s H' \vdash t'$ then for all $id \in \mathsf{dom}(H)$:*

$$\sum_{\substack{\forall ref \in \mathsf{refs}(t). \\ ref \mapsto_p id \in H}} p \;= 1 \quad \Longrightarrow \quad \sum_{\substack{\forall ref \in \mathsf{refs}(t'). \\ ref \mapsto_{p'} id \in H'}} p' \;\in \{0, 1\}$$

i.e., for all resources with identifier id in the incoming heap and all references in the term pointing to this resource, if the sum of all permissions pointing to this resource are 1 *in the incoming heap then either this is preserved in the outgoing heap* or *the total permissions in the output heap is* 0*, i.e., this resource has now been fully shared and has no ownership tracking.*

Furthermore, any resources in the outgoing heap that did not appear in the initial heap with references in the final term should have permissions summing to 1*. That is, for all $id' \in \mathsf{dom}(H') \wedge id' \notin \mathsf{dom}(H)$:*

$$\sum_{\substack{\forall ref' \in \mathsf{refs}(t'). \\ ref' \mapsto_q id' \in H'}} q \;= 1$$

When we extend this result to multi-step reductions, we are able to restrict the result further, and prove that if the resulting term is of unique type, then not only is the sum of fractions preserved throughout the overall reduction (which we verify inductively using the above result) but also that in the final term the reference to the resource we are considering must itself be unique.

Theorem 9 (Multi-reduction borrow safety). *For a well-typed term* $\Gamma \vdash t : A$ *and all* Γ_0, s, *and* H *such that* $H \bowtie (\Gamma_0 + s \cdot \Gamma)$, *and multi-step reduction* $H \vdash t \Rightarrow_s H' \vdash v$, *then for all* $id \in \mathsf{dom}(H)$:

$$\sum_{\substack{\forall ref \in \mathsf{refs}(t). \\ ref \mapsto_p id \in H}} p = 1 \implies \exists! ref'.ref' \mapsto_1 id \in H'$$

i.e., for all resources with identifier id in the incoming heap and all references in the term pointing to this resource, if the sum of all permissions pointing to this resource are 1 *in the incoming heap then their total permission of* 1 *is preserved from the incoming heap to the resulting term, with this permission now contained in a single reference* ref'.

Furthermore, any new references in the final term should uniquely point to an identifier, and thus have permission 1. *That is, for all* $id' \in \mathsf{dom}(H') \wedge id' \notin \mathsf{dom}(H)$ *then:*

$$\forall ref \in \mathsf{refs}(v).\ \exists! ref'.\ ref' \mapsto_1 id' \in H'$$

The uniqueness theorem presented in Chapter 4 now follows as a direct corollary of the multi-step borrow safety theorem we described.

Corollary 1 (Uniqueness). *For a well-typed term* $\Gamma \vdash t : *A$ *and all* Γ_0, s, *and* H *such that* $H \bowtie (\Gamma_0 + s \cdot \Gamma)$ *and multi-reduction to a value* $H \vdash t \Rightarrow_s H' \vdash *v$, *for all* $id \in \mathsf{dom}(H)$ *then:*

$$\begin{aligned} &\forall ref \in \mathsf{refs}(t).(ref \mapsto_1 id \in H \implies ref \mapsto_1 id \in H') \\ \wedge\quad &\forall id' \in \mathsf{dom}(H') \wedge id' \notin \mathsf{dom}(H).\ \forall ref \in \mathsf{refs}(v).\ \exists! ref'.ref \mapsto_1 id' \in H' \end{aligned}$$

i.e., any references contributing to the final term that are unique in the incoming heap stay unique in the resulting term, and any new references contributing to the final term are also unique.

Finally, we see that the operational semantics we have defined (extended to full β-reduction) supports the equational theory we have gradually developed in this work. Note that the proof includes only those equations defined by the $\equiv$ relation presented throughout Sections 5.3 and 5.4; the remaining equations, presented in the appendix, are as standard for a graded linear calculus.

Theorem 10 (Soundness with respect to the equational theory). *For all* t_1, t_2 *such that* $\Gamma \vdash t_1 : A$ *and* $\Gamma \vdash t_2 : A$ *and* $t_1 \equiv t_2$ *and given* H *such that* $H \bowtie \Gamma$, *there exist multi-reductions to values that are equal under full* β*-reduction and evaluating any references to the value they point to in the resulting heaps (written* $H'(v_1)$ *and* $H'(v_2)$*):*

$$H \vdash t_1 \Rightarrow_1 H' \vdash v_1 \quad \wedge \quad H \vdash t_2 \Rightarrow_1 H'' \vdash v_2 \quad \wedge \quad H'(v_1) \equiv H''(v_2)$$

Full proofs for the above theorems may be found in Appendix c, along with collected typing and reduction rules.

5.6 RELATED WORK

LINEAR AND GRADED TYPES Much of the prior literature on linear and graded types was discussed in Chapters 2 and 3, but we briefly recap it here. The notion of linearity originated with Girard's linear logic [50], which treats information resourcefully by restricting the structural rules of intuitionistic logic. This was soon adopted by programming language researchers who were interested in resourceful behaviour of data, rapidly developing into the concept of *linear types* [142, 145], but it took substantial time for linearity to emerge into the realm of practical programming. Lately, linear types have seen a renaissance due to their adoption as an extension to Haskell [19]; other languages incorporating linearity (or affinity, which is similar but allows weakening) include ATS [156], Alms [134] and Mezzo [12].

This binary view (linear vs. non-linear) was later refined by the notion of *bounded* linear logic [51], which introduced a family of operators indexed by a polynomial giving an upper bound on the usage of a resource. Further generalisations of this idea to track a broader range of increasingly fine-grained properties are what led to the introduction of *graded* types, which allow for annotating values with precise information about how they interact with their environment.

Granule's particular approach to graded modalities draws heavily from literature on *coeffects*, which describe how programs *depend* on their context [108], though it also incorporates ideas from effect systems which we focus on less here. Coeffect systems developed concurrently with other work which approached the same target from a different angle—through attempting to find generalisations of bounded linear logic [29, 49]. Other instances of graded types include Quantitative Type Theory (QTT) [10, 94] upon which the type system for Idris 2 is built [27], the core calculus underlying Linear Haskell [19, 129], and others [2, 3, 47, 153].

UNIQUENESS Uniqueness types originated in the Clean language [112, 126], where they are used in lieu of monadic computation and for efficiency gains offered by in-place update, though they have also been applied in other settings such as Single-Assignment C [124]. In Clean, computation is based on graph rewriting and reduction; constants are graphs, and functions are graph rewriting formulas. The type system also has a rather different feel to those offered by more recent functional programming languages, which makes it more difficult to capture the benefits of Clean-style uniqueness in a modern setting, hence the value in our pursuit of this goal. Recent work has attempted to capture benefits of uniqueness while allowing a more modern programming style, such as Cogent [101] and Mercury [128].

Theoretical work on understanding uniqueness began with Harrington's uniqueness logic [57]; this was followed by a substantial

amount of theoretical groundwork on Clean's particular strategy for uniqueness, particularly in works by de Vries among others [140, 141]. These papers clarified the distinction between Clean's type system and systems based on the λ-calculus. Further work made headway on distinguishing uniqueness from other substructural systems, allowing for applications such as polymorphic programming and concurrency [138, 139]. This laid a foundation for Granule's orthogonal approach to uniqueness, allowing for uniqueness and linearity to be integrated in a single system, which lead to the extensions developed in this thesis.

More recently, there has been work on incorporating uniqueness into the OCaml language through a system of *modes*, which have some theoretical relationship to graded types [85]. Consider the below code:

```
type ('a, 'b) pair = { fst : 'a; snd : 'b }
let update : ((('a, 'b) pair) @ unique) -> 'c -> ('a, 'c) pair =
    fun pair c -> { overwrite pair with snd = c }
```

OCaml

Here, the `@ unique` annotation declares the pair to be unique, and so the `overwrite` operation is permitted to destructively update the second component of the pair.

REGION-BASED MEMORY MANAGEMENT Regions have been used over the years in the context of effect systems [67, 87]. One of the primary motivations of research into region types was their application in *region-based memory management* [131], which aimed to bring some of the benefits of traditional stack-based memory management to higher-order functional languages. Regions divide values based on their lifetimes, so a system with region types can safely allocate and de-allocate memory for values based on region type information, eliminating the need for garbage collection.

Early on, regions were restricted to have LIFO (last-in, first-out) lifetimes which followed the block structure of a language, but later work relaxed this constraint using uniqueness (see: static capabilities [149] and Cyclone [60]); a unique reference to a region ensures there are no aliases to the region, and that it can therefore be promptly de-allocated. Additionally, regions themselves act as a way to control aliasing, and can be thought of as equivalence classes for a "may alias" relation—in other words, values which do not share a region may not alias with one another, and so if a value does not share a region with anything else then it may be safely mutated in place.

Work on Cyclone [42] demonstrated the relationship between regions and unique pointers, observing that "unique pointers are essentially lightweight, dynamic regions that hold exactly one object." Beyond that, Rust's lifetimes are heavily based on regions, and there exists an extension of ML called Affe [116] which aims to support both linearity and borrowing using regions.

CAPABILITIES Capabilities are tokens that a function must possess in order to be able to access a particular location in memory. Capabilities are generally linear, and cannot be duplicated or discarded, in order to prevent them from being forged [54]. Implementations exist for various object-oriented languages such as Java [6] and Scala [56]. More functional languages taking inspiration from the idea of capabilities also exist [12, 97]. Recent work on linear constraints for Haskell [129], which hopes to allow for something similar to borrowing within the framework of linear Haskell, also descends from work on capabilities. Ambient capabilities can also be internalised as a comonad to capture purity within an impure language [33].

OWNERSHIP AND BORROWING Ownership was developed as a framework for understanding aliasing in object-oriented languages [36, 98], with related work introducing notions like 'islands' [61], 'balloon types' [8] and external uniqueness [35]. The intent of ownership is to give a high-level structural view of objects and references, akin to type systems which allow for a high-level structural view of data.

More recently, ownership is pervasively used in Rust in order to help ensure memory safety. Multiple formalisations for Rust's ownership model have been attempted; RustBelt [69] gives a lower-level encoding of Rust intended for formal verification while Oxide [151] is a higher-level encoding designed for theoretical work, among others [68, 107]. Rust is not the only modern language to make use of ownership; other languages like Swift incorporate similar ideas, and work on introducing ownership into languages with manual memory management is ongoing [122]. Lorenzen, Leijen, and Swierstra [84] describe how to use related ideas to determine which functional programs can be executed in-place without allocation, and recent work on *reachability types* [14, 150] scales reasoning about lifetimes and sharing into higher-order polymorphic settings, inspired by separation logic.

Rust also provides an *unsafe* mode (a superset of the safe portion of Rust). While in Rust memory is managed through ownership and borrowing by default with memory for unsafe code needing to be managed manually, in languages like Granule managing memory automatically through a garbage collector is the default. Using our extension to obviate the need for garbage collection is a special case applying to some subset of a program. If the user does not wish for ownership to be taken into consideration, they can use Granule's pre-existing type system as before.

Borrowing with a Rust-like syntax has also been incorporated into OCaml's mode system [85]. They use a *locality* mode in addition to uniqueness to ensure that references do not leave their scope, much like we use a continuation to encapsulate the lifetime of a borrow. Their definition of the borrow combinator is displayed below for reference.

```
val borrow : 'a @ unique -> ('a @ local -> 'b)
    -> ('a * 'b aliased) @ unique
let borrow x f = let result = f &x in x, { a = result }
```
OCaml

5.7 CONCLUSION

Graded type systems and ownership with borrowing are both ways of carefully and precisely managing the usage of data built upon the foundation of linear logic, but these two approaches have developed through very different pathways on the road to being incorporated in modern-day programming languages. In this work, we have developed a core calculus that captures many of the key concepts for ownership tracking in a graded setting. We connect these fine-grained substructural notions back to prior work on simpler systems such as linear and uniqueness types which sit closer to the theory, like the system we developed in Chapter 4. This has allowed us to not only better understand the relationship between these disparate approaches to resourceful reasoning but also to examine how they interact.

By developing a framework through which both ownership properties and precise grades for reasoning about data can be tracked within the setting of the Granule language, we demonstrate that careful management of both resource and memory usage are not only compatible but complementary in a functional context. This chapter represents a piece of two larger puzzles—one aiming to develop an in-depth theoretical understanding of Rust's comprehensive approach to memory management, and one aiming to expand the range of properties about programs that can be represented explicitly through graded types—and we look forward to seeing more ideas being shared (or, indeed, borrowed) across the boundary between these two closely related worlds in the future.

Part III

SESSION TYPES & GRADED CHANNELS

6 MESSAGE IN A BOTTLE
LINEAR SESSION TYPES

6.1 INTRODUCTION

Most programming languages ascribe a notion of *type* (dynamic or static) to data, classifying *what* data we are working with—integer, string, function, etc. Languages like Go, Rust and Haskell among many others extend this to the realm of communication between two participants via *typed channels*, where any process with access to a channel endpoint can send and receive values of a given type across it. These typed channels only give us information about the nature of the data being communicated, however; to go beyond this we must consider *behavioural types*, which as described in Chapter 1 capture not just *what* data is, but *how* it is calculated.

One such behavioural type system for concurrency is *session types* [62, 154], adding more structure to typed channels by incorporating a protocol that describes the precise behaviour of the processes communicating over the channel. As an example of such a protocol, consider the standard example (with added flavour) of a customer who wishes to purchase a cake from a bakery. In natural language, we could describe the protocol as follows: the customer names their desired cake (a string), the bakery provides the price (an integer), and then the customer decides if they are happy with said price. If so, they confirm delivery details (another string), but if not, they end the communication immediately. From the customer's perspective, this protocol can be described as a session type, which would look like:

```
Send String (Recv Int (Select (Send String End) End))
```

The particular constructs at play here will be explained in more depth later, but it is clear to see that this allows more information about the communication to be encoded in the type than simply the kinds of data that are being sent and received. We will also see later how the session type for the bakery can be understood as the *dual* to that of the customer, which helps to enforce certain communication properties akin to deadlock freedom.

Session types naturally fit into the more general substructural discipline of *linear types* [50, 142, 148]. By treating data as a *resource* which must be used exactly once, linear types can capture various kinds of stateful protocols of interaction. Session types are inherently substructural: channels cannot in general be arbitrarily duplicated or discarded if they are to be used according to a precise sequence of operations (the protocol). This idea has allowed logical foundations to be developed

for session types based on Curry-Howard correspondences with both intuitionistic [31, 132] and classical linear logic [73, 147]. Furthermore, linearly-typed functional languages then provide an excellent basis for session-typed programming [48]; session types can be represented in the linear lambda calculus via an additional interface for channels without any other changes to the type system [48, 80].

In this chapter, we illustrate how to embed session types into Granule's core calculus in a sound way that respects communication safety, laying the groundwork for integrating communication behaviours with the increased expressivity of graded types in Chapter 7.

A key part of this development will involve reconciling the tension between three competing requirements: (1) side effects inherent in communication primitives, (2) non-linearity, and (3) the call-by-value semantics one might expect in most programming languages (in contrast to call-by-name, which is the basis of most theoretical explanations of linear and graded type systems). Requirements (1) and (2) can be satisfied more easily in a call-by-name setting, but in a similar way to the issues discussed in Chapters 4 and 5, (3) (CBV semantics) introduces unsoundness.

Whilst a call-by-name semantics for a modal treatment of non-linearity captures the idea of a number of 'copies' of a value, a call-by-value semantics unifies these copies into just one value. In the context of concurrency, then, a non-linear channel in a CBV semantics points to a single channel and so multiple uses within a program can lead to deadlock or race-conditions. Our solution revolves around preventing resource allocating primitives (those that create channels) from being promoted to non-linear contexts, and representing multiplicity of channels through data structures. Section 6.3 describes in more detail the problem and our solution, which extends the approach described in Chapter 5 for resolving the same issues in a setting involving uniqueness and ownership.

Granule is a research programming language built upon the core of the linear λ-calculus, extended with *graded modalities* as described above. A previous iteration of session types for Granule was described by Orchard et al. [103]. However, the system described required a monadic interface to avoid some issues caused by the call-by-value semantics; we solve these problems here to provide a more general and powerful interface for session-typed programming.

Later, in Chapter 7, we will illustrate three common communication patterns and how they can be described by combining linear session types and graded modal types. All of these patterns are a variation on the more traditional "fork" primitive introduced in this chapter, with some non-linear behaviour introduced in a sound way via graded modal types, and sometimes restrictions to protocols via *type predicates*.

All of the ideas presented here are implemented in the latest versions of the Granule compiler; interested readers who wish to experiment

for themselves should feel free to download and install the most recent release[1]. We develop the operational semantics of our approach, via a core calculus (Section 6.4). Later, we will establish that type safety is preserved (which encompasses communication safety and deadlock freedom), though we postpone this until the next chapter which will directly build upon the system described here (Section 7.3).

6.2 LINEAR SESSION-TYPED CHANNELS

The implementation of session types in Granule is based on the GV calculus (originating from Gay and Vasconcelos [48], further developed by Wadler [147], for which we use Lindley and Morris' formulation [80]). The GV system extends the linear λ-calculus with a data type of channels $\mathsf{Chan}(S)$ parameterised by session types S [154], which capture the protocol of interaction allowed over the channel. Communication in GV is usually asynchronous: sending always succeeds, but receiving can block. Lindley and Morris consider a synchronous semantics [80].

Starting with a simple subset, session types can describe channels which send or receive a value of type T and then can be used according to session type S written $!T.S$ or $?T.S$ respectively, or can be closed written $\mathsf{end}_!$ or $\mathsf{end}_?$. There are then functions for sending or receiving a value on a channel, forking a process at one end of a channel returning the other, and waiting for a channel to be closed:

$$
\begin{aligned}
\mathsf{send} &: T \otimes \mathsf{Chan}(!T.S) \multimap \mathsf{Chan}(S) \\
\mathsf{fork} &: (\mathsf{Chan}(S) \multimap \mathsf{Chan}(\mathsf{end}_!)) \multimap \mathsf{Chan}(\overline{S}) \\
\mathsf{recv} &: \mathsf{Chan}(?T.S) \multimap T \otimes \mathsf{Chan}(S) \\
\mathsf{wait} &: (\mathsf{Chan}(\mathsf{end}_?) \multimap 1)
\end{aligned}
$$

where $\overline{S}$ is the *dual* session type to S, defined $\overline{!T.S} = ?T.\overline{S}$, $\overline{?T.S} = !T.\overline{S}$, $\overline{\mathsf{end}_?} = \mathsf{end}_!$ and $\overline{\mathsf{end}_!} = \mathsf{end}_?$. The fork combinator leverages the duality operation, spawning a process which applies the parameter function with a fresh channel, thus returning the dual endpoint.

The core interface in Granule correspondingly has operations with the following types, where `Protocol` is the kind of session types whose constructors and associated operations we highlight in burgundy:

```
send        : ∀ {a : Type, p : Protocol}
            . LChan (Send a p) → a → LChan p
recv        : ∀ {a : Type, p : Protocol}
            . LChan (Recv a p) → (a, LChan p)
forkLinear  : ∀ {p : Protocol}
            . (LChan p → ())  → LChan (Dual p)
close       : LChan End → ()
```

1 The latest build of the Granule compiler is always available from https://github.com/granule-project/granule/releases.

The explicit definition of `Protocol` is as follows:

```
p ::= End | Send a p' | Recv a p' | Select p1 p2 | Offer p1 p2
```

Note that Granule proper includes a couple of extra protocols for selection and offering, which will be explained in more detail below. The type-level function `Dual : Protocol → Protocol` provides the standard notion of duality for session types, defined:

```
Dual End = End

Dual (Send a p) = Recv a (Dual p)

Dual (Recv a p) = Send a (Dual p)

Dual (Select p1 p2) = Offer (Dual p1) (Dual p2)

Dual (Offer p1 p2) = Select (Dual p1) (Dual p2)
```

We can define a parallel composition combinator (like the one used earlier in the `parSum` example in Chapter 5) on thunked computations (i.e., of type `() → a`), by using `forkLinear` to run one computation in parallel and then return its result via the associated channel:

```
-- 'par' expects two thunks, 'f' and 'g'
-- and returns their resulting values in a pair
par : ∀ {a : Type, b : Type} . (() → a) → (() → b) → (a, b)
par f g =  let                    -- compute f in new thread
               c       = forkLinear (λc' → close (send c' (f ())));
               b       = g ();   -- compute g in current thread
               (a, c') = recv c;
               -- get result of f from other thread
               ()      = close c'
           in (a, b)
```

Here, the result of the thunk `g` is computed in the same thread running the `par` function, while a separate thread is forked to compute the result of `f`. Hence, the thunks are executed in parallel as desired.

CHOICE We also provide some utility operations for internal and external choice (also known as selection and offering); note that as described in the original formulation by Lindley and Morris [80] these can be defined in terms of the core functions given above, and so when it comes to our semantics we will not need to define explicit reductions for or prove properties relating to these operations, but we do provide them as primitives in the implementation for ease of use.

```
selectLeft   : ∀ {p1 p2 : Protocol} .
             . LChan (Select p1 p2) → LChan p1
selectRight  : ∀ {p1 p2 : Protocol} .
             . LChan (Select p1 p2) → LChan p2
offer        : ∀ {p1 p2 : Protocol, a : Type}
    . (LChan p1 → a) → (LChan p2 → a) → LChan (Offer p1 p2) → a
```

The following gives a brief example putting all the primitives together:

```
server : LChan (Offer (Recv Int End) (Recv () End))
  → Int [0..1] → Int
```

```
server c [d] = offer (λc →
                 let (x, c) = recv c in let () = close c in x)
                     (λc →
                 let ((), c) = recv c in let () = close c in d) c

client : ∀ {p : Protocol} . LChan (Select (Send Int End) p) → ()
client c = let c = selectLeft c;
               c = send c 42
           in close c

example : Int -- Evaluates to 42
example = server (forkLinear client) [100]
```

The `server` offers a choice between being sent an integer or a unit value. The second parameter (bound to `d`) is used as a default value in the case that a unit value is received by the server, where `Int [0..1]` denotes that this integer can be used 0 or 1 times (see Chapter 3 for more explanation of this grading). The `client` selects the left behaviour, sends an integer, then closes its side of the communication.

The final definition `example` spawns the client with a channel and connects the dual end of the channel to the server which returns the received value of `42` here.

CORE CALCULUS REPRESENTATION We briefly recap the calculus of Chapter 3 for reference, repeating its terms, types, contexts and rules.

Terms in the graded calculus are given as follows:

$$t ::= x \mid \lambda x.t \mid t_1\, t_2 \mid (t_1, t_2) \mid \mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2 \\ \mid \mathbf{unit} \mid \mathbf{let\ unit} = t_1\ \mathbf{in}\ t_2 \mid [t] \mid \mathbf{let}\ [x]\ = t_1\ \mathbf{in}\ t_2 \qquad \text{(terms)}$$

with the syntax of types:

$$A, B ::= A \multimap B \mid A \otimes B \mid \mathsf{unit} \mid \Box_r A \qquad \text{(types)}$$

and contexts:

$$\Gamma ::= \ \cdot \mid \Gamma, x : A \mid \Gamma, x : [A]_r \qquad \text{(contexts)}$$

The typing rules are largely the same as those presented in Chapter 3, with the exception of the same restriction on promotion that was employed in Chapter 5.

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

$$\frac{\Gamma \vdash t : A \qquad \neg \mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

On top of this core we introduce a type of channels indexed by session types (protocols), with grammar:

$$A := \ldots \mid \mathbf{Chan}\,P \qquad \text{(types, extended)}$$
$$P := \mathbf{Send}\ A\ P \mid \mathbf{Recv}\ A\ P \mid \mathbf{End} \qquad \text{(session types)}$$

with the duality function $\overline{P}$ on session types defined as expected:

$$\overline{(\mathbf{Send}\ A\ P)} = \mathbf{Recv}\ A\ \overline{P} \qquad \overline{(\mathbf{Recv}\ A\ P)} = \mathbf{Send}\ A\ \overline{P} \qquad \overline{\mathbf{End}} = \mathbf{End}$$

As mentioned above, we elide the offer and select types from the core calculus and the theoretical development in this chapter, since they can also be emulated using send and receive and coproducts as in the Fst presentation of GV [80].

The term language is extended with the following primitives and their typing:

$$\begin{aligned} \mathsf{send} &: \mathbf{Chan}\,(\mathbf{Send}\ A\ P) \multimap A \multimap \mathbf{Chan}\,P \\ \mathsf{receive} &: \mathbf{Chan}\,(\mathbf{Recv}\ A\ P) \multimap A \otimes \mathbf{Chan}\,P \\ \mathsf{forkLinear} &: (\mathbf{Chan}\,P \multimap \mathsf{unit}) \multimap \mathbf{Chan}\,\overline{P} \\ \mathsf{close} &: \mathbf{Chan}\,\mathbf{End} \multimap \mathsf{unit} \end{aligned}$$

We assume asynchronous communication, which is captured in the extended operational model of Section 6.4.

INVERSES One interesting consequence of introducing linear channels into Granule is that they allow us to formulate a notion of *inverse* for linear types [90]. In particular, they allow us to define an operation on types A^{-1} such that we have an isomorphism $(A^{-1})^{-1} \cong A$.

We define the multiplicative inverse of type A as the type of functions *consuming* A:

$$A^{-1} \triangleq A \multimap \mathbf{unit}$$

One direction of the involution $A \multimap (A^{-1})^{-1}$ is easy:

```
type Inverse a = a → ()

-- i.e. the type expands to a → ((a → ()) → ())
isoL : ∀ {a : Type} . a → Inverse (Inverse a)
isoL x = λf → f x
```

This direction of the involution is of course also well-defined in the linear λ-calculus and in other linearly-typed languages such as Haskell. The opposite direction $(A^{-1})^{-1} \multimap A$ is much more challenging: in fact it is not inhabited if we restrict ourselves to the linear λ-calculus or even traditional non-linear Haskell, but it is instead a sequentially-realizable function [82, 83].

A sequentially-realizable function is one which has outwardly pure behaviour but relies on a notion of local side effects; these are entirely contained within the body of the function. Traditionally, sequentially-realizable functions have only been expressible in the ML family of languages (which allow unrestricted side effects) but not at all in Haskell. However, via Granule's linear channels, it is possible to capture sufficient side effects in order to write such a function. The same approach would work equally well in other implementations of similar session-typed calculi, such as FST [81] or FREEST 2 [7].

Note that given our definition of inverse, `forkLinear` acts as a combinator relating inverse and duality; we can rewrite the type signature as follows:

$$\mathsf{forkLinear} : \mathbf{Inverse}\,(\mathbf{Chan}\,P) \multimap \mathbf{Chan}\,\overline{P}$$

Understanding this allows us to define the other direction of the involution, given below.

```
isoR : ∀ {a : Type} . Inverse (Inverse a) → a
isoR k  =
  let r = forkLinear (λs → k (λx → let c = send s x in close c));
(x, c') = recv r;
     () = close c'
  in x
```

Thus `k` has type `(a → ()) → ()`, to which the function `λx → let c = send s x in close c` is passed; this sends the input `x : a` on the channel `c` which is then closed. This channel is provided by `forkLinear`, and so `k` is applied in a process taking one end of the channel to 'sneak out' the value of type `a`. Outside this, `recv` waits to receive from the dual end of the channel returned by `forkLinear`. The remaining channel is closed and `x` is returned. A local side effect is performed within `isoR` which is not observable externally, but it would not have been possible to construct the required function without carrying out this effect.

In languages like ML, an equivalent definition can be given using mutable references which resembles Longley's **F** combinator [82].

We do not linger on the notion of inverses here as this is not a focus for this thesis, but we refer the interested reader to our functional pearl on the topic if they desire some further exploration of these ideas [90]. We do, however, provide a proof that the two directions of this involution do indeed form an isomorphism in Appendix d.5. This proof is presented verbatim from the aforementioned paper, and uses the syntax of the GV session type calculus rather than Granule's.

Session types based on linear channels can also be represented in Haskell, but they do not allow us to demonstrate a full isomorphism to the same extent. Here we illustrate the more challenging direction of the involution using the Priority Sesh library[2], a recent package for session-typed communication in Linear Haskell which is itself inspired by the GV calculus [72]. However, note that the two directions cannot quite form an involution here, since everything must be wrapped up in the linear IO monad for these session types to be used; we cannot confine the side effects to the function as in Granule.

```haskell
type Inverse' a = a %1 -> Linear.IO ()

isoR :: Inverse' (Inverse' a) %1 -> Linear.IO a
isoR k = do
   (s, r) <- new
   void $ forkIO $ k (\z -> send s z)
   recv r
```

Haskell

6.3 GRADING, CALL-BY-VALUE, AND EFFECTS

The type system described thus far is based heavily on that of Orchard et al. [103], though with some simplifications where we elide concepts that are irrelevant to this thesis. Orchard et al. also described an earlier version of session types in Granule based somewhat on the GV calculus, but the session types were not included in the language's formal semantics at the time, which resulted in some cases where combining grading and session types led to unsoundness and the possible introduction of deadlocks. This work resolves these issues and we will present a full operational semantics in Section 6.4 as part of our formal theory of session types alongside grading, but before we do this we first discuss the problems present in the original system and describe our solution.

The first and primary problem becomes apparent when promoting a primitive function which creates new channels. Consider the following example which would be allowed by the promotion rule of Chapter 3:

```
problematic : Int
```

2 Available at https://github.com/wenkokke/priority-sesh

```
problematic =
  let [c] : ((LChan (Recv Int End)) [2])
          = [forkLinear (λc → close (send c 42))];
      (n, c') = recv c;  () = close c';
      (m, c') = recv c;  () = close c'
   in (n + m)
```

On line 4, the program forks a process that sends 42 on a channel, but under a promotion, with the type explaining that we want to use the resulting value twice (given by the explicit type signature here). This promotion then allows two uses of the channel on lines 5-6.

The typical semantics for coeffect-based calculi in the literature is call-by-name [2, 29, 47, 108]. Under a call-by-name semantics, which Granule allows via the extension `language CBN`, this program executes and produces the expected result of `84`. The key is that call-by-name reduction substitutes the call to `forkLinear` into the two uses of variable `c` on lines 4 and 5, and thus we are receiving from two different channels. However, under the default call-by-value semantics (which was chosen in Granule for simplicity and to avoid complications resulting from effects), line 3 is fully evaluated (underneath the graded modal introduction), and so variable `c` on lines 4 and 5 refers to a single channel. This means that executing this program blocks indefinitely on line 5; we get an error from the underlying implementation's concurrency primitives (written in Haskell) describing a `thread blocked indefinitely in an MVar operation`. This comes from Granule's runtime which leverages these standard primitives.

In order to get around this problem with non-linear channels in a call-by-value setting, Orchard et al. [103] instead deal with channels monadically, so here we would end up with a channel of type `(Session (LChan ...)) [2]` (with the `Session` monad)[3]. To make this example well-defined would then require a distributive law which maps from this to `Session ((LChan ... ) [2])`, copying the channel, i.e.:

```
dist : ∀ {p : Protocol, s : Semiring, r : s} .
    Session ((LChan p) [r]) → (Session (LChan p)) [r]
```

Providing such a distributive law is unsound—it would enable the `problematic` example and thus indefinite blocking—and fortunately it is also *not* derivable. However, we wish to avoid the monadic programming style as it is not required when working with just linear channels. It also complicates code, requiring the programmer to adapt their syntax when working inside the monad.

Our alternative solution is that like in Chapter 5, we instead syntactically restrict promotion to terms which do not allocate resources, i.e., whose reduction does not create linear channels. In particular

3 In Granule such a channel would in fact have the type `(LChan ... <Session>) [2]`, with `<Session>` being an instance of the graded monadic `<r>` modalities which are dual to the graded comonadic `[r]` modalities. We elide discussing these further here since we will not need any monad other than `Session`.

this includes reducible expressions containing the `forkLinear` primitive which causes the difficulty here. Non-linear channels will later be re-introduced by additional primitives in Sections 7.2.1, 7.2.2 and 7.2.3 which allow for precise and carefully managed non-linear usage, allowing for grading to be combined with linear channels even in a call-by-value setting and without the extra overhead that was required when every channel had to be wrapped in the `Session` monad.

Thus, we continue to use the modified promotion of Chapter 5:

$$\frac{\Gamma \vdash t : A \qquad \neg\mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}'$$

where $\mathsf{resourceAllocator}(t)$ classifies those terms whose reduction to a value would trigger creation of a resource; such terms contain a use of the forkLinear primitive (or the other fork primitives introduced in later sections) in a reducible position. Note in particular that $\neg\mathsf{resourceAllocator}(\lambda x.\mathsf{forkLinear}\, t)$ since reduction does not happen underneath a λ-abstraction (Section 6.4 defines the reduction semantics). Definition 22 in Appendix d.1 gives the formal definition. Generally, any term whose reduction semantics creates channels is classified as a resource allocator, extending the idea from Chapter 5.

Promotion is not the only source of unsoundness in the session type system of Orchard et al., [103]. Some additional restrictions on the primitives which combine grading with linear channels are required, in order to prevent possible problems that can occur when two ends of the same channel reach inconsistent states or are used at different grades. We will discuss these problems in Chapter 7.

6.4 SEMANTICS

We define an operational model for our linear session types here, in order to make the connection between our type system and the underlying communication behaviours more concrete, and so that we can carefully prove that properties such as type and communication safety will continue to hold after our extensions with additional primitives in Sections 7.2.1, 7.2.2 and 7.2.3. We give a semantics to terms in the context of runtime "configurations" of processes and channels. This allows us to show that communication safety holds by proving that progress can always be made regardless of the current global state, and proving preservation of typing for the overall configuration.

6.4.1 Operational Model

The semantics is call-by-value and at its core is based on the standard small-step operational semantics for the linear λ-calculus and its

extension to graded modal types in a call-by-value setting given by Orchard et al. [103]. We extend this with the processes and channels.

PROCESSES A *process configuration* tracks the state of a number of processes, comprising ordinary terms, running in parallel:

$$\mathcal{P} ::= \cdot \mid (\mathcal{P} \mid t^{\mathcal{R}}) \qquad \text{(process configurations)}$$
$$\mathcal{R} ::= \circ \mid \bullet \qquad \text{(role)}$$

(the parentheses are typically omitted). The role $\circ$ marks a subordinate process and $\bullet$ marks the main process. Typing of process configurations later establishes that a well-typed process configuration has exactly one main process (Section 6.4.2).

Configurations are subject to an equivalence relation capturing commutativity and associativity of the parallel composition in process configurations:

$$\mathcal{P}_1 \mid \mathcal{P}_2 \equiv \mathcal{P}_2 \mid \mathcal{P}_1 \qquad (\mathcal{P}_1 \mid \mathcal{P}_2) \mid \mathcal{P}_3 \equiv \mathcal{P}_1 \mid (\mathcal{P}_2 \mid \mathcal{P}_3)$$

In subsequent relations (e.g., reduction) we work up-to process equivalence, i.e., we can freely apply the equivalence relation to rearrange process configurations. This relation is akin to *structural congruence* in process algebras.

CHANNELS Communication channels in the operational semantics are represented by pairs of dual *channel endpoints* to allow asynchronous duplex communication. Throughout, c ranges over channel endpoint names, which appear in 'positive' and 'negative' forms:

$$c ::= n^+ \mid n^- \qquad \text{(channel endpoint names)}$$

The *dual* of an endpoint name c is another endpoint name, denoted $\overline{c}$, where $\overline{n^+} = n^-$ and $\overline{n^-} = n^+$. Duality is an involution, i.e., $\overline{\overline{c}} \equiv c$.

Channel configurations record the contents of channel endpoints in the environment, associating names with a queue $\vec{v}$ of message values:

$$\mathcal{C} ::= \cdot \mid \mathcal{C}, c = \vec{v} \qquad \text{(channel configurations)}$$

We write $\langle\,\rangle$ for an empty queue, $\vec{v} : v$ for a queue with value v enqueued on the end, and $v : \vec{v}$ for a queue with v on the front (ready to be dequeued).

We will sometimes refer to c as simply a channel, and also as a reference to its contents (the queue) within a configuration.

REDUCTION Single-step global reductions are then defined over pairs of process and channel configurations (which we refer to as *global configurations*) with reductions of the form:

$$(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')$$

where $\mathcal{P}, \mathcal{P}'$ are (incoming and outgoing) process configurations and $\mathcal{C}, \mathcal{C}'$ are (incoming and outgoing) channel configurations respectively. The channel configuration may be updated if reduction involves applying a concurrency primitive, which could involve spawning two endpoints of a new channel or communicating a value from one end of a channel to the other.

Channels inside a channel configuration can either be *linear* or they can be *shared*, which means that rather than being used linearly they are used according to some grade r. We will discuss this in more detail when we describe primitives that can spawn graded channels, starting from Section 7.2.1.

The standard call-by-value small-step semantics for the linear λ-calculus embeds within this more complex global configuration semantics through the use of the following rule:

$$\frac{t \rightsquigarrow t'}{(\mathcal{P} \mid t^{\mathcal{R}};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid t'^{\mathcal{R}};\ \mathcal{C})}\ \textsc{reduce}$$

which states that if the term t can reduce to t' (without any need to interact with the global context) then we can make that same reduction within the global configuration, at any role, whilst threading all of the extra channel configuration information through so that it can come back into play once we reach a communication step. Going forwards, we elide the role in places where a reduction rule is agnostic to the role, assuming that the role of a process being reduced is preserved in its corresponding location in the output process configuration.

We will now briefly discuss how these standard reduction steps work in practice, to provide a foundation from which we can go on to explain the behaviour of our core session type primitives in this setting of process and channel configurations. We will explain some rules in detail and elide others which are very similar to those already introduced. All reduction rules for the entire semantics are collected together in d.1.

The core reduction rules for the (CBV) linear λ-calculus terms are of course β-reduction and function application, which are given below.

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_2}\ \rightsquigarrow_{\text{APP}} \qquad \frac{t \rightsquigarrow t'}{v\, t \rightsquigarrow v\, t'}\ \rightsquigarrow_{\text{prim}} \qquad \frac{}{(\lambda x.t)\, v \rightsquigarrow t[v/x]}\ \rightsquigarrow_{\beta}$$

Our semantics is call-by-value, and left-to-right: we fully reduce the left side of a function application to a value, then reduce on the right.

Perhaps more interesting are the rules for "boxed" terms which are those typed with a graded modality in our system; note that in Chapter 3 and many Granule-based papers with call-by-need semantics any

boxed term is a value by default, but here a boxed term is only a value if the term inside the box is itself a value. The reduction rules are:

$$\frac{}{\textbf{let } [x] \ = [v] \textbf{ in } t \rightsquigarrow t[v/x]} \ \rightsquigarrow\Box\beta \qquad \frac{t \rightsquigarrow t'}{[t] \rightsquigarrow [t']} \ \rightsquigarrow\Box\text{reduce}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let } [x] \ = t_1 \textbf{ in } t_2 \rightsquigarrow \textbf{let } [x] \ = t_1' \textbf{ in } t_2} \ \rightsquigarrow_{\text{LET}}\Box$$

Thus we can β-reduce an introduction-elimination pair for the graded modal values, eliminating the box modality through pattern matching. We reduce inside the box for terms that are not yet values, and we have a congruence rule for reducing the term in an elimination form. We will skip over the rules for tensor product and unit in this section, since they are very similar to the rules for boxed terms, with rules for reducing terms inside a pair and also substitution and elimination for products and unit as above.

It should be fairly clear to see that the semantics presented so far is coherent with the simple graded core fragment of the heap semantics presented in Chapter 5, since the rules for each construct are roughly equivalent. In other words, if $H(t) \rightsquigarrow H'(t')$ here, then $(H \vdash t) \rightsquigarrow (H' \vdash t')$ in Chapter 5's semantics.

The real action happens when we reach an instance of application where the value being applied on the left is some (potentially already partially applied) communication primitive and the term on the right is the final parameter for said primitive, as this is where some communication behaviour needs to occur and so we will have to consider the state of the channel configuration. We begin by considering the creation of a channel, which occurs through the use of the forkLinear primitive. For this we need to extend the syntax of terms just as we did for arrays and references in Chapter 5 to include *runtime values*, which in this setting are just channel endpoint names c:

$$t ::= \ldots \mid c \qquad \text{(Runtime terms)}$$

Furthermore, we also define the notion of an evaluation context, denoted *Ctxt*, in which reduction can occur at a single "hole" in the syntax tree:

Definition 9 (Evaluation contexts)**.** *Syntax trees with a single hole are defined:*

$$
\begin{array}{lll}
\mathit{Ctxt} & ::= & \qquad\qquad\qquad\qquad \mathit{Contexts} \\
& \mid & - \\
& \mid & \mathit{Ctxt}\, t \\
& \mid & v\, \mathit{Ctxt} \\
& \mid & [\mathit{Ctxt}] \\
& \mid & \mathbf{let}\, [x] = \mathit{Ctxt}\, \mathbf{in}\, t \\
& \mid & \mathbf{let}\, \mathbf{unit} = \mathit{Ctxt}\, \mathbf{in}\, t \\
& \mid & \mathbf{let}\, (x, y) = \mathit{Ctxt}\, \mathbf{in}\, t \\
& \mid & (\mathit{Ctxt}, t) \\
& \mid & (v, \mathit{Ctxt})
\end{array}
$$

where $-$ denotes a hole and we write $Ctxt\{t\}$ to denote the operation of filling the hole in $Ctxt$ with a term t.

The operational semantics of forkLinear, in some evaluation context, is then:

$$
\frac{\#c}{\begin{array}{l}(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkLinear}\, (\lambda x.t)\};\ \mathcal{C}) \\ \rightsquigarrow (\mathcal{P} \mid t[c/x]^{\circ} \mid \mathit{Ctxt}\{\overline{c}\};\ \mathcal{C}, \overline{c} = \langle\,\rangle, c = \langle\,\rangle)\end{array}}\ \textsc{forkLinear}
$$

In this case, we select a fresh name c for the new channel (denoted by $\#c$ in the premise) and add both dual endpoints of the channel (c and $\overline{c}$) to the channel configuration, each empty of values for now. As well as reducing the term by carrying out the expected substitution we also spawn a new process for communication on the fresh channel and add it to the process configuration, marked as a subordinate process.

In the forkLinear rule in particular, we need not concern ourselves with whether the fresh endpoint name c is a positive channel name n^+ or negative n^- since the actual polarity is immaterial; whatever fresh name we have, the channel configuration is extended with a queue associated to this endpoint name c and a queue associated to its dual endpoint name $\overline{c}$.

The next cases we consider are sending and receiving information on one end of a channel, where the dual end will have to receive or provide said information in turn. Sending is asynchronous, given by:

$$
\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{send}\, c\, v\};\ \mathcal{C}, \overline{c} = \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{c\};\ \mathcal{C}, \overline{c} = \vec{v} : v)}\ \textsc{send}
$$

The value v provided by the user is consumed, and the channel configuration is updated by appending v to the end of whatever queue of values is already present inside the *dual* end of the channel they sent on, where they will later be necessarily received by virtue of the session type guarantees. The process configuration contains the original end of the channel in context, which the sender will continue

to use according to the next step of its protocol given by the session type. Conversely, receiving operates as follows:

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{receive}\, c\};\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{(v, c)\};\ \mathcal{C}, c = \vec{v})}\ \textsc{recv}$$

In this case, the first value is popped from the queue of values contained in the end of the channel held by the receiver, and this is given to the user inside the context in the process configuration; they can again continue to use their end of the channel according to its protocol. This may involve receiving more values or sending values to the dual end depending on the particular session type that has been specified.

Finally, channels are closed as follows:

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{close}\, c\};\ \mathcal{C}, c = \langle\,\rangle) \rightsquigarrow (\mathcal{P} \mid Ctxt\{\mathbf{unit}\};\ \mathcal{C})}\ \textsc{close}$$

Note this reduction closes the referred-to endpoint of the channel c, rather than its dual, akin to how RECV receives from the referred to endpoint of the channel. This allows for asynchronous behaviour since the dual endpoint $\bar{c}$ may still have values in its queue to be received.

6.4.2 Runtime Typing

In order to later consider type safety properties, we extend typing to process and channel configurations, considering a notion of *runtime typing* as we did for arrays and references earlier.

For runtime typing of terms, the same typing rules apply but where the context also contains channel names and their typing. We refer to a *runtime context* by the letter γ, defined:

$$\gamma = \emptyset \mid \gamma, c : \mathbf{Chan}\, P \qquad \text{(Runtime contexts)}$$
$$\Gamma = \ldots \mid \gamma \qquad \text{(Contexts (redefined))}$$

where $c : \mathbf{Chan}\, P$ types a linear channel. Contexts Γ are then extended to include γ, though it is sometimes useful to denote contexts that contain only runtime (channel) types separately using the γ notation.

Context addition extends to γ, treating the linear channel names just as linear variables. The variable rule of typing extends to channel names, i.e., we can derive runtime typing:

$$\frac{}{c : \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\ \textsc{var}$$

Process configurations $\mathcal{P}$ are typed by a judgment $\Gamma, \gamma \vdash \mathcal{P} : A$ where A is the type of the main process and Γ are the term variables and γ the channel names in scope of the processes in $\mathcal{P}$. Here, the type $\bot$

cannot be given to any term. It is used only for process configuration typing to denote a configuration without a main process.

$$\frac{}{\emptyset \vdash \cdot : \bot}\ \textsc{empty} \qquad \frac{\begin{array}{c}\Gamma_1, \gamma_1 \vdash \mathcal{P} : \bot \\ \Gamma_2, \gamma_2 \vdash t : A\end{array}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2 \vdash \mathcal{P} \mid t^{\bullet} : A}\ \textsc{main}$$

$$\frac{\begin{array}{c}\Gamma_1, \gamma_1 \vdash \mathcal{P} : A \\ \Gamma_2, \gamma_2 \vdash t : \mathsf{unit}\end{array}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2 \vdash \mathcal{P} \mid t^{\circ} : A}\ \textsc{sub}$$

Empty process configurations are given the type $\bot$. The MAIN rule puts a term t in parallel with a configuration $\mathcal{P}$ and marks it as the main process and thus the process configuration has the type of this process. The SUB rule puts a term t in parallel with a configuration $\mathcal{P}$, but this term is marked as a subordinate process with $\circ$ (of which there may be many) and thus the return type of t must be unit. The process configuration's type is that of $\mathcal{P}$. Note that this type can be $\bot$, and so a process configuration with type $\bot$ is not necessarily empty.

Channel configurations $\mathcal{C}$ are typed by a judgment $\gamma \vdash \mathcal{C}$, defined:

$$\frac{}{\emptyset \vdash \cdot}\ \textsc{nil}$$

$$\frac{\gamma \vdash \mathcal{C}}{\gamma, c : \mathbf{Chan}\, P \vdash \mathcal{C}, c = \langle\rangle}\ \textsc{empty} \qquad \frac{\gamma \vdash \mathcal{C}}{\gamma, c : \mathbf{Chan}\, \mathbf{End} \vdash \mathcal{C}, \overline{c} = \vec{v}}\ \textsc{end}$$

$$\frac{\gamma_1 \vdash v : A \qquad \gamma_1, \gamma_2, c : \mathbf{Chan}\, P \vdash \mathcal{C}, \overline{c} = \vec{v} : v}{\gamma_1, \gamma_2, c : \mathbf{Chan}\, (\mathbf{Send}\ A\ P) \vdash \mathcal{C}, \overline{c} = \vec{v}}\ \textsc{send}$$

$$\frac{\gamma_1 \vdash v : A \qquad \gamma_1, \gamma_2, c : \mathbf{Chan}\, P \vdash \mathcal{C}, c = \vec{v}}{\gamma_1, \gamma_2, c : \mathbf{Chan}\, (\mathbf{Recv}\ A\ P) \vdash \mathcal{C}, c = v : \vec{v}}\ \textsc{recv}$$

The EMPTY rule says that a channel endpoint that is empty can have any type (it is in a 'waiting state'). Meanwhile, the END rule says a channel can be typed at **Chan End** while its dual endpoint may contain any number of values. Reading the SEND rule bottom-up, the conclusion types a channel on which a value v of type A can be sent and thus we can type that channel with the continuation behaviour P in the premise when that value has been enqueued on the channel. Note that channel value is dual to the typing, since sending is actioned by enqueueing on the dual endpoint. Reading the RECV rule bottom-up, the conclusion types a channel on which a value v of type A can now be received (from the head of the channel) with the continuation behaviour P in the premise after the value has been received (and hence removed) from the front of the channel.

6.4.2.1 Summary

Thus far we have extended our core calculus from Chapter 3 with linear channels and session types, and established just the operational semantics for the core calculus with linear channels. This resolves the soundness issues with the original implementation of session types in Granule due to Orchard et al. [103], and we are now in a position to establish the key properties of well-typed programs containing session types, but we delay this to Section 7.3, since the material in the next chapter will build directly on the calculus described here and so it will be easier to prove the theorems together for the overall system.

In the next chapter, we will introduce new communication primitives for non-linear behaviours and expand both the type system and the operational semantics to include their model and associated definitions, which also includes expanding the definition of processes to include some intermediate runtime processes. In Section 7.3, we will consider the syntactic type safety properties for the full calculus with the non-linear extensions included.

7 REPLICATE, REUSE, REPEAT
NON-LINEAR COMMUNICATION

7.1 INTRODUCTION

As we saw in Chapter 6, session types provide guarantees about concurrent behaviour and can be understood through a correspondence with linear logic, with propositions as sessions and proofs as processes.

Modern substructural type systems, however, allow us to go beyond the notion of linearity, classifying usage into more than just linear or non-linear. As explained in Chapter 2, this idea originates from Bounded Linear Logic (BLL), which generalises the ! modality to a family of modalities indexed by a polynomial expressing an upper bound on usage [51]. *Graded modal types* [103] extend this generalisation by building on the notion of *coeffects*, and as we have seen throughout this thesis they are an expressive system allowing for the specification and verification of many behavioural properties of programs.

Strict linearity alone is limiting since there exist useful communication patterns that rely on non-linear behaviours. For example, shared channels can repeatedly spawn a process with binary communication along a fresh linear channel. In this chapter, we demonstrate that combining session types with graded modal types (particularly for capturing coeffects) allows for various well-known non-linear behaviours of concurrent programs to be reintroduced in a linear setting in a controlled and precise way via the facilities of graded modalities.

We show three common patterns and how they can be described by combining linear session types and graded modal types: *reusable channels* (Section 7.2.1), *replicated servers* (Section 7.2.2), and *multicast communication* (Section 7.2.3). All of these patterns are a variation on the "fork" primitive introduced in Chapter 6, with some amount of substructural behaviour introduced via graded modal types, and sometimes restrictions to protocols via *type predicates*.

7.2 GRADING FOR NON-LINEAR COMMUNICATION

7.2.1 Reusable Channels

A reusable or *non-linear* channel is one which can be shared and thus used repeatedly in a sound way. This contrasts with the notion seen in Section 6.3 of promoting a fresh linear channel to being non-linear in a call-by-value setting, which we disallowed since it was unsound; a linear channel used in a shared way can easily lead to a deadlock

where one user of the channel leaves it in a state which is then blocking for another user of the channel. The key problem with sharing a channel is ending up with inconsistent states across different shared usages. This can be avoided if the protocol allowed on the channel is restricted such that only a single 'action' (send, receive, choice, or offer) is allowed, and thus if the channel is used multiple times it can never be left in an inconsistent state across shared uses. This is captured by the idea that $(P \mid P) \neq P$ in general in process calculi (like CCS or the π-calculus) and so a replicated channel which has more than a single action cannot be split off into many parallel uses, e.g., $(ab)^* \mid (ab)^* \not\equiv (ab)^*$. However, if there is only a single action then multiple repeated parallel uses are consistent, e.g., $a^* \mid a^* \equiv a^*$.

We capture this idea via the following variant of the fork primitive with arbitrarily graded channels:

```
forkNonLinear : ∀ {p : Protocol, s : Semiring, r : s}
              . {SingleAction p, ExactSemiring s}
             ⇒ ((LChan p) [r] → ()) → (LChan (Dual p)) [r]
```

where `ExactSemiring : Semiring → Predicate` restricts acceptable semirings to only those that do not permit any approximation; or, put another way, if one grade approximates to another in an exact semiring then the two grades must be equal. As far as semirings mentioned in this thesis are concerned, the grade `r` is allowed to be a natural number, representing exact usage, but importantly is *not* allowed to be an interval, representing a range of possible usages.

`SingleAction : Protocol → Predicate` is a type constraint that characterises only those protocols which comprise a single send, receive, choice, or offer and rejects any with multiple steps, i.e.,

```
SingleAction End

SingleAction (Send a End)
SingleAction (Recv a End)
SingleAction (Offer End End)
SingleAction (Select End End)
```

As an example, consider a channel in a graded modality which says it can be used exactly `n` times. The following uses this channel to send every element of a vector of size `n`:

```
sendVec : ∀ {n : Nat, a : Type}
        . (LChan (Send a End)) [n] → Vec n a → ()
sendVec [c] Nil = ();
sendVec [c] (Cons x xs) =
      let c'  = send c x;
          ()  = close c'
      in sendVec [c] xs
```

This code shows the powerful interaction between grading, linearity, and indexed types in Granule. Note that the above code is typeable without any of the new primitives described in this section, but would be unusable without promoting a use of `forkLinear`, which is disallowed

by the type system. However, we can complete this example with a dual process (`recvVec`) that is then connected to `sendVec` via `forkNonLinear` (on line 14 below):

```
recvVec : ∀ {n : Nat, a : Type}
          . N n → (LChan (Recv a End)) [n] → Vec n a
recvVec Z [c]     = Nil;
recvVec (S n) [c] = let (x, c') = recv c;
                        () = close c'
                    in Cons x (recvVec n [c])

length' : ∀ {n : Nat, a : Type} . Vec n a → (N n, Vec n a)
length' Nil = (Z, Nil);
length' (Cons x xs) = let (n, xs) = length' xs in (S n, Cons x xs)

example : ∀ {n : Nat, a : Type} . Vec n a → Vec n a
example xs = let (n, list) = length' xs
             in recvVec n (forkNonLinear (λc → sendVec c list))

main : Vec 5 Int
main = example (Cons 1 (Cons 1 (Cons 2 (Cons 3 (Cons 5 Nil)))))
```

Since `forkNonLinear` gives us a pair of channels whose semiring-usage is exact, we must consume all values produced by `sendVec` on the dual endpoint. Therefore, the receiver `recvVec` needs to know how many elements to receive. This information is passed separately (via an indexed natural number `N n`) so that `recvVec` can induct on the natural number and leave no message unreceived. A system with dependent session types [132, 133] could avoid this by first sending the length, but this is not (yet) possible in Granule; the Gerty prototype language provides full dependent types and graded modal types which would be a good starting point [95].

7.2.1.1 Semantics

In the notation of the core calculus, the `forkNonLinear` primitive has its typing rendered as:

$$\frac{\mathsf{SingleAction}(P) \qquad \mathsf{ExactSemiring}(S) \qquad r : S}{\emptyset \vdash \mathsf{forkNonLinear} : (\Box_r \mathbf{Chan}\, P \multimap \mathsf{unit}) \multimap \Box_r (\mathbf{Chan}\, \overline{P})}\ \textsc{forkNonLin}$$

This primitive now introduces the notion of a non-linear, or *shared* channel, which we reflect into the run-time typing and channel configuration typing that was set up in Section 6.4.

As an example (partial) derivation here, consider the following term which uses forkNonLinear with a protocol **Send Int End**, and the natural numbers semiring with grade 2 to spawn a process reusing the channel twice to send two integer terms (3 then 5), yielding the dual channel which can be used twice to receive two integers:

$$\dfrac{\dfrac{\dfrac{y : [\mathbf{Chan}\,(\mathbf{Send}\;\mathbf{Int}\;\mathbf{End})]_2 \vdash \mathbf{let\ unit} = \mathsf{close}\,(\mathsf{send}\,y\,3)\ \mathbf{in}\ \mathsf{close}\,(\mathsf{send}\,y\,5) : \mathsf{unit}}{x : \square_2(\mathbf{Chan}\,(\mathbf{Send}\;\mathbf{Int}\;\mathbf{End}))\vdash \mathbf{let}\,[y] = x\ \mathbf{in\ let\ unit} = \mathsf{close}\,(\mathsf{send}\,y\,3)\ \mathbf{in}\ \mathsf{close}\,(\mathsf{send}\,y\,5) : \mathsf{unit}}\ \text{ELIM}}{\emptyset \vdash \lambda x.\mathbf{let}\,[y] = x\ \mathbf{in\ let\ unit} = \mathsf{close}\,(\mathsf{send}\,y\,3)\ \mathbf{in}\ \mathsf{close}\,(\mathsf{send}\,y\,5) : \mathsf{unit}}\ \multimap_i}{\emptyset \vdash \mathsf{forkNonLinear}\,(\lambda x.\mathbf{let}\,[y] = x\ \mathbf{in\ let\ unit} = \mathsf{close}\,(\mathsf{send}\,y\,3)\ \mathbf{in}\ \mathsf{close}\,(\mathsf{send}\,y\,5)) : \square_2(\mathbf{Chan}\,(\mathbf{Recv}\;\mathbf{Int}\;\mathbf{End}))}\ \text{FORKNL}$$

SHARED CHANNEL TYPING We extend runtime-typing contexts γ with a shared channel typing:

$$\gamma = \cdot \mid \gamma, c : \mathbf{Chan}\,P \mid \gamma, c :^* \mathbf{Chan}\,P \qquad \text{(Runtime contexts)}$$

where $c :^* \mathbf{Chan}\,P$ types a shared channel.

The predicate $[\Gamma]$ classifying graded contexts then also extends to include the rule that $[\Gamma] \implies [\Gamma, c :^* \mathbf{Chan}\,P]$, i.e., we treat shared channels akin to graded variables, as graded variables may be used non-linearly. Runtime context addition then accounts for sharing:

Definition 10 (Runtime context addition)**.** *Runtime context addition is akin to context addition, where channels can be marked as linear or shared and addition is only defined for contexts that are disjoint in their linear channels and where shared channels may appear in either context:*

$$\begin{aligned}
\emptyset + \gamma_2 &= \gamma_2 \\
\gamma_1 + \emptyset &= \gamma_1 \\
(\gamma_1, c : \mathbf{Chan}\,P) + \gamma_2 &= (\gamma_1 + \gamma_2), c : \mathbf{Chan}\,P \\
&\quad \textit{iff}\ \ c \notin \mathsf{dom}(\gamma_2) \\
\gamma_1 + (\gamma_2, c : \mathbf{Chan}\,P) &= (\gamma_1 + \gamma_2), c : \mathbf{Chan}\,P \\
&\quad \textit{iff}\ \ c \notin \mathsf{dom}(\gamma_1) \\
(\gamma_1, c :^* \mathbf{Chan}\,P_1) + (\gamma_2, c :^* \mathbf{Chan}\,P_2) &= (\gamma_1 + \gamma_2), c :^* \mathbf{Chan}\,P \\
&\quad \textit{where}\ \ P = P_1 \bowtie P_2
\end{aligned}$$

where $P_1 \bowtie P_2$ defines compatibility of shared channel session types, *a symmetric partial operation defined:*

$$\begin{aligned}
P \bowtie P &= P \quad \textit{if}\ \mathsf{SingleAction}(P) \\
P \bowtie \mathbf{End} &= P \quad \textit{if}\ \mathsf{SingleAction}(P) \\
\mathbf{End} \bowtie P &= P \quad \textit{if}\ \mathsf{SingleAction}(P)
\end{aligned}$$

where SingleAction *is a built-in predicate on protocols:*

$$\mathsf{SingleAction}(\mathbf{End}) \qquad \mathsf{SingleAction}(\mathbf{Recv}\ A\ \mathbf{End})$$

$$\mathsf{SingleAction}(\mathbf{Send}\ A\ \mathbf{End})$$

Thus, two shared channels are compatible when they have the same protocol, or when one of the shared channels has been used and is in

the **End** state. For example, consider three non-linear uses of a shared channel c whose protocol is a single action **Send** A **End**; after one communication is performed, one of the uses of c is typed as **End**, but this is still compatible with the two remaining non-linear uses of c typed at **Send** A **End**.

Thus, $+$ combines contexts which are disjoint in their linear channels, and allows shared channels to be included in both contexts as long as their session types are compatible, computing via $\bowtie$ their compatible shared type.

Scalar multiplication of contexts extends to scalar multiplication of runtime contexts where $r \cdot (\gamma, c :^* \mathbf{Chan}\, P) = (r \cdot \gamma), c :^* \mathbf{Chan}\, P$.

The variable rule extends to shared channel names:

$$\frac{}{c :^* \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\ \textsc{varChanS}$$

The weakening rule also extends to shared channel names:

$$\frac{\Gamma \vdash t : B}{\Gamma, 0 \cdot \Gamma', c :^* \mathbf{Chan}\, P \vdash t : B}\ \textsc{weakChanS}$$

CHANNEL CONFIGURATIONS AND REDUCTION In channel configurations, we explicitly denote shared channels with assignments written $c^* = \vec{v}$. If we need to distinguish a linear channel in a channel configuration, we write $c^1 = \vec{v}$ though if the linear or shared nature is immaterial we write $c = \vec{v}$ as before. The typing of channel configurations behaves the same for shared channels as linear channels. All of the rules are collected in Appendix d.

Send and receive reduction rules apply to shared channels exactly as linear channels. However, the close primitive has a different reduction semantics for shared channels:

$$\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{close}\, c\};\ \mathcal{C}, c^* = \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{\mathbf{unit}\};\ \mathcal{C}, c^* = \vec{v})}\ \textsc{sharedClose}$$

Note that when a shared channel is closed, we preserve it inside the channel configuration rather than removing it. Furthermore, we do not even require that it is empty since the channel may be shared and messages may have arrived for another interaction, e.g., particularly in the case of forkNonLinear.

An important point here is that although the channel is preserved, unlike in the linear close rule, this still does not permit orphan messages to remain inside any channel when it is closed. This is because of the `ExactSemiring` constraint on the forkNonLinear primitive, which ensures that there are always exactly the same number of senders as receivers, and the `SingleAction` constraint, which restricts each participant to only send or receive a single message. Indeed, this is exactly why both of these restrictions are necessary.

REDUCTION SEMANTICS FOR FORKNONLINEAR Now that we have extended the runtime typing notions to shared channels we can consider the operational semantics of forkNonLinear.

The operational model of forkNonLinear follows the typing. Its reduction starts with an application of this primitive to a function value forkNonLinear $(\lambda x.t)$ in some evaluation context within a process reducing by creating a fresh shared channel endpoint c (and its dual endpoint $\overline{c}$). It spawns a new process t with x replaced by the graded modal lifting of the channel $[c]$ and returning the dual endpoint also inside a graded modal box:

$$\frac{\#c}{\begin{array}{ll} & (\mathcal{P} \mid Ctxt\{\mathsf{forkNonLinear}\,(\lambda x.t)\};\ \mathcal{C}) \\ \rightsquigarrow & (\mathcal{P} \mid Ctxt\{[\overline{c}]\} \mid t[[c]/x]^{\circ};\ \mathcal{C}, \overline{c}^{*} = \langle\,\rangle, c^{*} = \langle\,\rangle) \end{array}}\ \textsc{forkNonLinear}$$

The spawned process, based on the function body t (with the promoted channel substituted in), is marked as being a subordinate process with $\circ$. The newly created channels in the context are marked as being shared channels c^{*} and $\overline{c}^{*}$.

7.2.2 Replicated Servers

The π-calculus provides replication of a process P as the process $!P$, which session-typed π-calculus variants have refined into the more controlled idea of having a replicated 'server'. The spawning of replicated instances is controlled via a special receive [18, 123, 136], e.g., written like $*c(x).P$ meaning receive an x on channel c and then continue as P, whilst still providing the original process. From an operational semantics point of view this looks like $*c(x).P \mid \overline{c}\langle d\rangle.Q \rightarrow *c(x).P \mid P[d/x] \mid Q$ where $\overline{c}\langle d\rangle.Q$ sends the message d to the server which 'spawns' a fresh copy of the server process P, preserving the original server process for further clients to interact with.

This important pattern is not achievable with the reusable channels of Section 7.2.1 but the non-linearity inherent to replicated servers can be captured and explained again via the combination of graded and indexed types. We provide the replicated server functionality here as the fork variant `forkReplicate`:

```
forkReplicate : ∀ {p : Protocol, n : Nat} . {ReceivePrefix p}
              ⇒ (LChan p → ()) [0..n] → N n
              → Vec n ((LChan (Dual p)) [0..1])
```

Here the grading is less general than in Section 7.2.1; we focus on a particular grading which says that given a server process `(LChan p → ())` that can be used 0 to n times, we get a vector of size n of dual channels which we can use to interact with the server. The predicate `ReceivePrefix p` classifies those protocols which start with a receive:

```
ReceivePrefix (Recv a p)        ReceivePrefix (Offer p1 p2)
```

Each returned client channel can itself be discarded due to the graded modality `... [0..1]`. Thus, we can choose not to use any/all of the client channels, reflected in the dual side where the server can be used at most n times. This introduces flexibility in the amount of usage.

The following example demonstrates `forkReplicate` to define a simple server that can receive requests to receive two further integers which it adds and sends back, or requests to compare a received integer for equality to 0:

```
addServer : LChan (Offer (Recv Int (Recv Int (Send Int End)))
                         (Recv Int (Send Bool End))) → ()
addServer c =
 offer
       (λc → let (x, c) = recv c;
                 (y, c) = recv c;
             in close (send c (x + y))

       (λc → let (x, c) = recv c;
             in close (send c (x == 0))) c
```

We then define two clients `client1` and `client2` that can interact with this server, exercising both parts of the offered choice:

```
client1 : ∀ {p : Protocol}
        . LChan (Select (Send Int (Send Int (Recv Int End))) p)
        → Int
client1 c = let (x, c) = recv (send (send (selectLeft c) 10) 20);
                () = close c
            in x

client2 : ∀ {p : Protocol}
        . LChan (Select p (Send Int (Recv Bool End)))
        → Bool
client2 c = let (b, c) = recv (send (selectRight c) 42);
                () = close c
            in b
```

We then compose the server and clients together using `forkReplicate` to get a vector of channels which we pattern match on immediately to get the channels to pass to each client (which are composed in parallel using `par` defined in Chapter 6):

```
import Parallel
-- Provides the ‘par‘ combinator derived from ‘forkLinear‘
main : (Int, Bool)
main =
  let (Cons [c1] (Cons [c2] Nil))
    = forkReplicate [addServer] (S (S Z))
  in par (λ () → client1 c1) (λ () → client2 c2)
```

In the intuitionistic linear logical propositions of Caires and Pfenning [31], this same idea of a replicating fork is captured by a non-linear channel (in Dual Intuitionistic Linear Logical style) which yields a linear version when interacted with; this is akin to the idea of *shared names* [154]. Our approach codifies the same principle using graded

modalities, but only allows finite replication. Section 7.2.2.2 shows a variation of this primitive which allows unbounded replication.

7.2.2.1 Semantics

In the notation of the core calculus, the `forkReplicate` primitive has its typing rendered as:

$$\frac{\mathsf{ReceivePrefix}(P)}{\varnothing \vdash \mathsf{forkReplicate} : \Box_{0..n}(\mathbf{Chan}\, P \multimap \mathsf{unit}) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\Box_{0..1}(\mathbf{Chan}\, \overline{P}))}\ \textsc{forkRep}$$

We thus also extend the core calculus with indexed types for natural numbers and vectors, which previously were derived inside Granule using the syntax of GADTs:

$$A ::= \ldots \mid \mathbf{N}\ n \mid \mathbf{Vec}\, n\, A$$

These indexed types are internally definable in Granule as GADTs, but for simplicity we add them as primitive to the core calculus, with standard introduction rules:

$$\frac{}{\varnothing \vdash \mathbf{Z} : \mathbf{N}\ 0}\ \mathrm{Z} \qquad \frac{\Gamma \vdash t : \mathbf{N}\ n}{\Gamma \vdash \mathbf{S}\, t : \mathbf{N}\ (n+1)}\ \mathrm{S}$$

$$\frac{}{\varnothing \vdash \langle\rangle : \mathbf{Vec}\, 0\, A}\ \textsc{nil} \qquad \frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : \mathbf{Vec}\, n\, A}{\Gamma_1 + \Gamma_2 \vdash t_1 :: t_2 : \mathbf{Vec}\, (n+1)\, A}\ \textsc{cons}$$

We elide their eliminators here for brevity, but their elimination is standard following usual eliminators for indexed types [109] (see also Granule's type system [103], whose indexed type eliminators also cause specialisation in grades, allowing type indices and grades to interact). As a shorthand, we write a vector with n elements as $\langle t_1, .., t_n \rangle$.

As an example (partial) derivation, the following uses forkReplicate to spawn a server that can be used at most three times to receive an integer and sends it back incremented:

$$\frac{\dfrac{\varnothing \vdash \lambda x.\mathbf{let}\ (y, x) = \mathsf{receive}\, x\ \mathbf{in}\ \mathsf{close}\, (\mathsf{send}\, x\, (y+1)) : \mathbf{Chan}\, (\mathbf{Recv}\ \mathbf{Int}\ (\mathbf{Send}\ \mathbf{Int}\ \mathbf{End})) \multimap \mathsf{unit}}{\varnothing \vdash [\lambda x.\mathbf{let}\ (y, x) = \mathsf{receive}\, x\ \mathbf{in}\ \mathsf{close}\, (\mathsf{send}\, x\, (y+1))] : \Box_{0..3}(\mathbf{Chan}\, (\mathbf{Recv}\ \mathbf{Int}\ (\mathbf{Send}\ \mathbf{Int}\ \mathbf{End})) \multimap \mathsf{unit})}\ \textsc{pr}}{\varnothing \vdash \mathsf{forkReplicate}\, [\lambda x.\mathbf{let}\ (y, x) = \mathsf{receive}\, x\ \mathbf{in}\ \mathsf{close}\, (\mathsf{send}\, x\, (y+1))] : \mathbf{N}\ 3 \multimap \mathbf{Vec}\, 3\, (\Box_{0..1}(\mathbf{Chan}\, (\mathbf{Send}\ \mathbf{Int}\ (\mathbf{Recv}\ \mathbf{Int}\ \mathbf{End}))))}\ \textsc{forkRep}$$

The operational model of forkReplicate splits into two rules. The first follows the typing and corresponds to a β-reduction of forkReplicate: its reduct has an application of this primitive to a promoted function application and a natural number term $\mathsf{forkReplicate}\, [\lambda x.t]\, n$. The result is to spawn off a number of intermediate processes called "watchers" and to return to the originating process a vector of fresh channel endpoints (each promoted), of type $\mathbf{Vec}\, n\, (\Box_{0..1}(\mathbf{Chan}\, \overline{P}))$:

$$\frac{\#c_1 \ldots \#c_n}{\begin{array}{l}(\mathcal{P} \mid Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\};\ \mathcal{C}) \\ \rightsquigarrow\ (\mathcal{P} \mid Ctxt\{\langle [\overline{c_1}], .., [\overline{c_n}]\rangle\} \mid \mathsf{Wch}\,(c_1 \mapsto \lambda x.t) \mid .. \mid \mathsf{Wch}\,(c_n \mapsto \lambda x.t) \\ \quad ;\ \mathcal{C}, {c_1}^* = \langle\rangle, .., {c_n}^* = \langle\rangle, \overline{c_1}^* = \langle\rangle, .., \overline{c_n}^* = \langle\rangle)\end{array}}\ \textsc{forkRep}$$

Thus, n pairs of fresh channel endpoints are created and the reduct forkReplicate, in some evaluation context $Ctxt$, is replaced by a vector of the dual endpoints (promoted to the graded modality). In parallel, n "watcher" processes are created, where $\mathsf{Wch}\,(c \mapsto \lambda x.t)$ is a runtime process waiting to receive a value on channel c and containing the continuation of the replicated process body: the suspended computation of process t, abstracted over the channel x.

Watcher processes (which are not ordinary terms) have the following typing, under the process typing judgment:

$$\frac{\Gamma_1, \gamma_1 \vdash \mathcal{P} : A \qquad \Gamma_2, x : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P), \gamma_2 \vdash t : \mathsf{unit}}{\Gamma_1 + (0..1 \cdot \Gamma_2), \gamma_1 + \gamma_2, c : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P) \vdash \mathcal{P} \mid \mathsf{Wch}\,(c \mapsto \lambda x.t) : A}\ \textsc{watch}$$

Thus, given a t which has a free x which is a channel on which we can receive, then t can be placed into a watcher process, linked to the channel c on which a receive can occur. Note that in the conclusion here we scale the requirement of t by $0..1$ since t may never be evaluated as forkReplicate only provides an upper bound on possible replications.

The second part of the operational semantics for forkReplicate then involves reducing a watcher process into a regular term-based process upon receiving an initial request:

$$\frac{}{(\mathcal{P} \mid \mathsf{Wch}\,(c \mapsto \lambda x.t);\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^\circ;\ \mathcal{C}, c = v : \vec{v})}\ \textsc{watcher}$$

For a watcher on channel c, if there is a value v at the end head of the channel v then a new subordinate process is formed, substituting c for x. Note that the channel is left unchanged so that the spawned process can receive the value; the presence of the value v in the channel simply triggers the spawning of the process from the watcher.

7.2.2.2 *Variations*

In `forkReplicate`, each returned channel (in a vector) can be discarded due to the graded modality `... [0..1]`: we can choose not to use any/all of the client channels, which allows for some flexibility in the amount of usage. A more strict variant is given as:

```
forkReplicateExactly : ∀ {p : Protocol, n : Nat} . {ReceivePrefix p}
    ⇒ (LChan p → ()) [n] → N n → Vec n (LChan (Dual p))
```

meaning we have exactly n clients that *must* spawn the server n times.

A more powerful version of replication allows an unknown number of clients (possibly infinite), with an interface similar to `forkReplicate` but returning a lazy stream of client channels which can be affinely used:

```
data Stream a = Next a (Stream a)

forkReplicateForever : ∀ {p : Protocol} . {ReceivePrefix p}
    ⇒ (LChan p → ()) [0..∞] → (Stream (LChan (Dual p))) [0..1]
```

The result is a stream wrapped in the graded modality with grade `0..1`. Due to the typing of pattern matching in Granule, one can pattern match on the stream inside the graded modality, such that its components inherit the grade `0..1`; e.g., `example [Next c rest] = ...` binds the stream `rest` in the scope of the function body with the same grade `0..1` so it can be used once or not at all.

This last variation of replication captures more closely the replicated server concept found in session-typed π-calculi. The above two primitives are provided in the implementation but are not considered in the core calculus and metatheory here.

7.2.3 Multicast Sending

The final primitive we introduce here provides the notion of *multicast* (or *broadcast*) communication where messages on a channel can be received by multiple processes by copying the messages. This non-linearity is thus on the payload values being sent, with a graded modal type deployed to explain that the amount of non-linearity in the messages matches the number of receiving processes (clients). The `forkMulticast` primitive provides this behaviour, typed in Granule:

```
forkMulticast : ∀ {p : Protocol, n : Nat}
              . {Sends p}
             ⇒ (Chan (Graded n p) → ()) → N n
             → Vec n (Chan (Dual p))
```

where `Graded : Nat → Protocol → Protocol` is a built-in type function that adds a graded modality to payload types, defined:

```
Graded n End           = End
Graded n (Send a p)    = Send (a [n]) (Graded n p)
Graded n (Recv a p)    = Recv (a [n]) (Graded n p)
Graded n (Select p1 p2) = Select (Graded n p1) (Graded n p2)
Graded n (Offer p1 p2)  = Offer (Graded n p1) (Graded n p2)
```

and `Sends : Protocol → Predicate` is a built-in predicate defined:

$$\frac{}{\texttt{Sends End}} \qquad \frac{\texttt{Sends p}}{\texttt{Sends (Send a p)}} \qquad \frac{\texttt{Sends p1} \quad \texttt{Sends p2}}{\texttt{Sends (Select p1 p2)}}$$

Thus, as long as we are sending values that are wrapped in the graded modality such that they can be used n times, we can then broadcast these to n client processes via the `Vec n (Chan (Dual p))` channels returned by `forkMulticast`.

For example, in the following we have a `broadcaster` that takes a channel which expects an integer to be sent which can be used 3 times.

```
broadcaster : LChan (Send (Int [3]) End) → ()
broadcaster c = close (send c [42])
```

We can then broadcast these results with `forkMulticast broadcaster (S (S (S Z)))` producing a 3-vector of channels. Below we aggregate the results from these three receiver channels by applying `aggregateRecv` to the vector, giving us the result `126`, i.e. `3 * 42`.

```
aggregateRecv : ∀ {n : Nat} . Vec n (LChan (Recv Int End)) → Int
aggregateRecv Nil         = 0;
aggregateRecv (Cons c cs) = let (x, c) = recv c;
                                () = close c
                            in x + aggregateRecv cs

main : Int -- Evaluates to 126
main = aggregateRecv (forkMulticast broadcaster (S (S (S Z))))
```

7.2.3.1 Semantics

In the notation of the core calculus, the `forkMulticast` primitive has its typing rendered as:

$$\frac{\mathsf{Sends}(P)}{\varnothing \vdash \mathsf{forkMulticast} : (\mathbf{Chan}\,(\mathsf{Graded}_n P) \multimap \mathsf{unit}) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})}\ \textsc{forkMulti}$$

Similarly to forkReplicate, the operational model of forkMulticast comprises two rules; a β-reduction that generate some intermediate processes, and a rule actioning those processes in response to communication. The β-reduction rule follows the typing: its reduct has an application of the primitive to a function value and a natural number term: $\mathsf{forkMulticast}\,(\lambda x.t)\,n$. The result is to spawn the process derived from t as well as a special "forwarding" process (explained below):

$$\frac{\#c \qquad \#c_1 \ldots \#c_n}{\begin{array}{ll} & (\mathcal{P} \mid \mathcal{C}txt\{\mathsf{forkMulticast}\,(\lambda x.t)\,n\};\ \mathcal{C}) \\ \rightsquigarrow & (\mathcal{P} \mid \mathcal{C}txt\{\langle \overline{c_1}, .., \overline{c_n}\rangle\} \mid t[c/x]^\circ \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n) \\ & \quad ;\ \mathcal{C}, c = \langle\,\rangle, \overline{c} = \langle\,\rangle, c_1 = \langle\,\rangle, .., c_n = \langle\,\rangle, \overline{c_1} = \langle\,\rangle, .., \overline{c_n} = \langle\,\rangle) \end{array}}\ \textsc{forkMulticast}$$

Thus, n pairs of fresh channel endpoints are created, $c_1 \ldots c_n$, and the forkMulticast reduct is replaced by a vector of the dual endpoints $\langle \overline{c_1}, .., \overline{c_n}\rangle$. In parallel, the function body t from the reduct has c substituted for x to form a subordinate process $t[c/x]^\circ$, and a "forwarding" process is created. The forwarding process $\mathsf{Fwd}\,(c \mapsto c_1, .., c_n)$ is a runtime process that receives a graded modal value from c, unboxes the value and forwards copies of this payload on the (dual end of) channels c_1 to c_n.

Forwarder processes are typed by the process typing judgment as:

$$\frac{\mathsf{Sends}(P) \qquad \Gamma, \gamma \vdash \mathcal{P} : A}{\Gamma, \gamma, c : \mathbf{Chan}\,(\mathsf{Graded}_n \overline{P}), c_i : \mathbf{Chan}\,P \vdash \mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n) : A}\ \textsc{fwd}$$

where c_i : **Chan** P is short hand for the typing c_1 : **Chan** $P, .., c_n$: **Chan** P. For example, if $P =$ **Send** $B\ P'$ (satisfying the Sends predicate) then c : **Chan** (**Recv** $(\Box_n B)$ $(\mathsf{Graded}_n \overline{P'})$) in the above.

The reduction of the forwarding process then occurs by receiving on c an $n-$graded value, which gets sent along n further channels:

$$\frac{}{(\mathcal{P} \mid \mathsf{Fwd}\ (c \mapsto c_1, .., c_n);\ \mathcal{C}, c = [v] : \vec{v}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n) \rightsquigarrow (\mathcal{P} \mid \mathsf{Fwd}\ (c \mapsto c_1, .., c_n);\ \mathcal{C}, c = \vec{v}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v)}\ \textsc{Forwarder}$$

Note that there is no "garbage collection" step for the forwarding process: this can be easily added as an implementational detail.

7.3 PROPERTIES

We consider standard probes on the language design: whether typing guarantees progress of reduction (we do not get stuck within a single step) and preservation of types (reduction does not conflict with the specifications of types and maintains its meaning throughout). Proofs of all statements that follow are provided in Appendix d.

We are now in a position to establish the key properties of well-typed programs containing graded modal session types. Our semantics defined over global configurations that was presented in Section 6.4 allows us to demonstrate that communication safety holds by proving two main theorems: progress and preservation.

Lemma 11. *Value lemma*
Given $\Gamma \vdash v : A$ *then, from its type, the shape of* v *can be inferred:*

- $A = A' \multimap B$ *then* $v = \lambda x.v'$ *or a partially applied primitive term* p.
- $A = \Box_r A'$ *then* $v = [v']$.
- $A = A' \otimes B$ *then* $v = (v_1, v_2)$.
- $A =$ unit *then* $v =$ **unit**.
- $A =$ **Chan** P *then* $v = c$.

Note that, as in other session-typed systems, typability of an expression is not sufficient to guarantee that it can immediately make progress at runtime [48]. For example, a channel with a type such as **Chan** (**Recv** $A\ P$) can only make progress if the dual channel has already sent the value that this channel needs to receive (as described by the Recv rule). Our progress theorem applies to well-typed configurations that also satisfy the following property.

Definition 11. *Given* $\Delta \vdash t : A \wedge \gamma \vdash \mathcal{C}$ *then we say that* $\mathcal{C}$ *is* buffered *with respect to* Δ, *written* $\mathsf{buffered}(\mathcal{C}, \Delta)$, *if for any channel* c *in* $\mathcal{C}$ *with type* **Chan** (***Recv*** $A\ P$), *we have that* c *contains at least one value* v.

The full inductive definition of the buffered *predicate is as follows, where* $c :^r \ldots$ *in the runtime contexts and* $c =^r \ldots$ *in the process configurations abstracts over both linear and shared channels.*

$$\frac{}{\mathsf{buffered}(\cdot, \gamma)} \qquad \frac{\mathsf{buffered}(\mathcal{C}, \gamma)}{\mathsf{buffered}((\mathcal{C}, c =^r \vec{v}), (\gamma, c :^r \mathbf{Chan}\, \textit{End}))}$$

$$\frac{\mathsf{buffered}(\mathcal{C}, \gamma)}{\mathsf{buffered}((\mathcal{C}, c =^r \vec{v}), (\gamma, c :^r \mathbf{Chan}\, (\textit{Send}\ A\ P)))}$$

$$\frac{\mathsf{buffered}(\mathcal{C}, \gamma)}{\mathsf{buffered}((\mathcal{C}, c =^r v : \vec{v}), (\gamma, c :^r \mathbf{Chan}\, (\textit{Recv}\ A\ P)))}$$

Theorem 11 (Progress terms)**.**

$$\begin{aligned} & \gamma \vdash t : A \\ \wedge\ & \gamma \vdash \mathcal{C} \\ \wedge\ & \mathsf{buffered}(\mathcal{C}, \Delta) \\ \wedge\ & \gamma \vdash \mathcal{P} : B \\ \Longrightarrow\ & \mathsf{value}(t) \vee \exists t', \mathcal{C}', \mathcal{P}'.\ (\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t';\ \mathcal{C}') \end{aligned}$$

A process configuration $\mathcal{P}$ contains all values, defined by the predicate $\mathsf{values}(\mathcal{P})$, if every process is either a value v or is a forwarder or watcher process.

Theorem 12 (Progress global config)**.**

$$\begin{aligned} & \gamma \vdash \mathcal{P} : A \ \wedge\ \gamma \vdash \mathcal{C} \ \wedge\ \mathsf{buffered}(\mathcal{C}, \Delta) \\ \Longrightarrow\ & \mathsf{values}(\mathcal{P}) \vee \exists \mathcal{C}', \mathcal{P}'.(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}') \end{aligned}$$

Proof. (Highlights) See d.3.1 for the full proof.

Note that there are two separate progress theorems here; the first is akin to the standard progress theorem on terms, though some cases require additional information from the process and channel configurations and the overall result is that either the term t is a value or a global step can be taken which can reduce either or both of the two configurations along with the term. The second global progress theorem completes the communication safety result by considering cases where we must reduce a process configuration without being able to focus on a term, for example if we need to reduce based on watcher and forwarder processes which only exist at runtime.

Most cases in the first progress theorem proceed exactly as they would in a standard proof of progress for a type system; in these cases the Reduce rule from the global semantics allows us to say that if a term can be reduced in the standard way, then we can take a global step which does not affect the remainder of the process configuration or any of the channels which exist at runtime, simply reducing that term in place.

The interesting cases are the ones where a communication primitive is applied to a value, which by the value lemma (Lemma 11) must be a channel. In these cases we cannot reduce by only considering the term; we must use the global semantic rules for the communication primitives, which take into account the channels that are present in the channel configuration and in some cases (such as ForkReplicate and ForkMulticast) also spawn additional processes.

The second proof is not too involved, and almost follows as a corollary of the first, but does require a little extra work for the cases of watcher and forwarder processes. Note in particular that it is possible for a forwarder process to still be present and for no reduction to be possible, but this is only in cases where all other terms are values, so the theorem does still hold here because $\mathsf{values}(\mathcal{P})$ holds for a process configuration as long as all terms have been fully reduced. □

Theorem 13 (Preservation)**.**

$$\begin{aligned} & \Gamma, \gamma \vdash \mathcal{P} : A \ \wedge \ \gamma \vdash \mathcal{C} \ \wedge \ (\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}') \\ \Longrightarrow \ & \exists \gamma'.\ \Gamma, \gamma' \vdash \mathcal{P}' : A \ \wedge \ \gamma' \vdash \mathcal{C}' \end{aligned}$$

Proof. (Highlights) See d.3.2 for the full proof.

The proof proceeds by induction on the structure of process configuration typing, in concert with channel configuration typing and the inductive definition of global reductions possible at each typing.

Preservation for reduction of a single term comes from the previous result for the graded core calculus as described in Chapter 3.

For the concurrent primitives (both the communication and forking primitives introduced here), their reductions involve terms in a reduction context. A lemma provides *factorisation* for a well-typed term in context: if $Ctxt\{t\}$ is well-typed then $Ctxt\{x\}$ is well-typed and t is well-typed separately (Lemma 22, p. 326). This lemma tells us that either x is a linear variable, i.e., the term t is substituted into a linear hole position, or x is a graded variable, i.e., the term t is substituted into a graded hole position, which for $\Gamma \vdash Ctxt\{t\} : B$ then gives:

$$\begin{aligned} (\exists \Gamma_1, \Gamma_2, A, r. \quad & \neg \mathsf{resourceAllocator}(t) \\ \wedge \quad & \Gamma_1, x : [A]_r \vdash Ctxt\{x\} : B \\ \wedge \quad & [\Gamma_2] \vdash t : A \\ \wedge \quad & \Gamma = \Gamma_1 + r \cdot \Gamma_2) \end{aligned}$$

Critical here is that we know that this factorisation tells us that t is not a resource allocator. Given a global reduction which leads to a new term t', we then have the evidence to show that $\neg \mathsf{resourceAllocator}(t')$ in these cases. *Context application* lemmas then give the dual result allowing us to reconstruct a well-typed term in context, i.e., $Ctxt\{t'\}$ is well-typed (with preserved types).

Note that communication primitives (send, receive, close) are not resource allocators and can occur in graded contexts, in which case

the factorisation lemma also states that their channels must therefore be shared, since $[\Gamma_2]$ classifies (runtime) contexts which are all graded or shared channels. In these cases, preservation is straightforward.

For the forking primitives, and auxiliary processes like forwarding and watching, the factorisation lemma tells us that these must necessarily be within a linear context as they are resource allocators. Thus, given $\Gamma \vdash Ctxt\{t\} : B$ we can factorise the typing into $\Gamma_1, x : A \vdash Ctxt\{x\} : B$ and $\Gamma_2 \vdash t : A$ where $\Gamma = \Gamma_1 + \Gamma_2$. This shows the effect of restricting terms to non-resource allocators in ensuring preservation. Type preservation follows by constructing typing of the subject produced by reduction, which satisfies the theorem statement.

For the forkReplicate case, the $\mathsf{ReceivePrefix}(P)$ predicate is essential to type preservation: the watcher processes that forkReplicate reduces into rely on having receiving channels in order to continue reducing. Furthermore, the typing of a forkReplicate reduct has its first argument of type $\square_{0..n}(\mathbf{Chan}\, P \multimap \mathsf{unit})$. Since this argument is a value in the reduct, by inversion (the value lemma 11), it is necessarily of the form $[\lambda x.t]$ and we have $\Gamma_1, x : \mathbf{Chan}\, P, \gamma_1 \vdash t : \mathsf{unit}$ and $0..n \cdot (\Gamma_1, \gamma_1) \vdash [\lambda x.t] : (\mathbf{Chan}\, P \multimap \mathsf{unit})$ where Γ_1, γ_1 is a graded context. Reduction yields n watchers, each of which holds a copy of t that by the typing of watchers promotes their resources by $0..1$ to $0..1 \cdot (\Gamma_1, \gamma_1)$. Thus the n-wise parallel composition therefore (by distributivity of $*$ over $+$) results in preserving the $0..n \cdot (\Gamma_1, \gamma_1)$ resources.

For the forkMulticast case, the $\mathsf{Sends}(P)$ predicate is essential to type preservation also: the forwarding processes that forkMulticast reduce into rely on having a receiving channel endpoint and subsequent channels to send upon. Furthermore, the payload grading given by $\mathsf{Graded}_n P$ ensures that the copying of the value to n channel endpoints is well-resourced. □

7.4 RELATED WORK

SHARED CHANNELS Session type theories often provide non-linearity via *shared names* which provide repeated behaviours such as a replicable 'server' [154] akin to our `forkReplicate` concept.

Particularly relevant in our context of linear types, Caires and Pfenning give a session-types meaning to intuitionistic linear logic in which propositions denote the types of (linear) channels and the exponential modality ! types the communication of shared servers which can be repeatedly spawned [31]; that is, $!A$ denotes the type of a non-linear shared channel which is used by a 'server' process to spawn new sessions conforming to the communication behaviour of A. The production of a new 'client' channel involves giving a semantic meaning to *dereliction* where a non-linear (shared) channel has a linear copy 'sharded' off. In their formulation, $\Gamma; \Delta \vdash P :: T$ types a process P

providing behaviour T along a channel, and using non-linear channels Γ and linear channels Δ. Access to the replicable server then involves the following process and typing:

$$\frac{\Gamma, u : A; \Delta, y : A \vdash P :: T}{\Gamma, u : A; \Delta \vdash (\nu y) u\langle y \rangle.P :: T}$$

i.e., access to the shared server via shared channel $u : A$ (essentially a $!A$ linear channel) is provided to P by the new linear channel $y : A$, amounting to contraction of u and dereliction of u into y. This is coherent with a call-by-name semantics for linear logic where a value of type $!A$ is essentially an infinite number of copies of an A value, whereas in the call-by-value semantics $!A$ is just a single A value that can be used non-linearly. Therefore in our CBV approach, `forkReplicate` provides a vector of client channels to represent channel copies.

In the GV calculus, Lindley and Morris add replicated channels by first extending GV with an S4-style $\Box$ modality (akin to the exponential modality) to provide "unlimited types" [80]. They derive a replicate primitive of the following type (where in GV, the type $!A.S$ denotes a channel on which we first send an A then behave as S, see Section 6.2):

$$\mathsf{replicate} : (!(\Box\overline{S}).\mathsf{end}_!) \otimes (\Box(S \multimap \mathsf{end}_!)) \multimap \mathsf{end}_!$$
$$\mathsf{replicate}(x, f) = \mathsf{send}\ ([\mathbf{let}\ [g]\ = f\ \mathbf{in}\ \mathsf{fork}\ f], x)$$

(we have used our notation for modal introduction and elimination to make the derivation more easily understood here).

Thus, replication takes a pair of (1) a channel x on which is sent a $\Box\overline{S}$ value, i.e., a non-linear channel which can behave as $\overline{S}$ and (2) a non-linear function f which expects a channel behaving as S as input). The function unboxes the non-linear function f into an instance g, and forks this, but underneath modal promotion, yielding a 'boxed' channel which is then sent on the input channel x. Whilst GV is call-by-value like Granule, the derivation above is only sound if $\Box$ behaves in a call-by-name way, i.e., we do not reduce under $\Box$ but instead capture the enclosing fork such that uses of it on the dual endpoint of x get a freshly forked function g each time. Thus, the semantics of $\Box$ in GV in this extension differs to Granule which is 'pervasively' call-by-value, evaluating inside of modalities.

For more detail, d.4 gives a rendering of replicate in Granule, but using its call-by-name language extension for soundness.

NON-LINEAR COMMUNICATION Possibly the closest ideas to the ones in this chapter existing in the literature appear in the type system of Zalakain and Dardha, who use leftover typing to define a resource-aware variant of the π-calculus that can represent shared channels (which can be freely discarded or reused), graded channels (which must be used precisely n times for some n) and linear channels [155].

Applying their notion of 'graded channel' allows them to capture roughly the same behaviour as our `forkNonLinear` primitive, but their

work does not consider more general forms of grading (though their mechanisation allows for user-defined resource algebras). We capture some of the same ideas inside our unified graded approach, showing how different amounts of sharing can be characterised precisely with the interaction of linear, indexed, and graded types; our setting is also more general, since we work in a pre-existing programming language rather than developing a calculus focused on communication.

Another paper which describes some similar non-linear communication patterns to our work is the generalisation of session types via adjoint logic presented by Pruiksma and Pfenning [115]. Here, they offer the capacity for both multicast communication and replicated servers, corresponding to our `forkMulticast` and `forkReplicate` primitives respectively. However, their system does not include any notion of grading, and so while they offer non-linear communication this is without the precise quantification allowed by grading in our system.

CLASSICAL SESSION TYPES As well as the session-based interpretation of intuitionistic linear logic mentioned above [31], it is also possible to understand the correspondence between communication and logic by relying on *classical* linear logic, as was first demonstrated by Wadler [147]. The differences between these two systems go beyond the superficial; it has been shown that there exists a class of processes which are typable following the classical interpretation but not via the intuitionistic interpretation, and that these are the processes which break the so-called *locality principle* for shared channels [59].

Recent work by Kokke, Morris and Wadler extends this interpretation through an approach which draws on notions from bounded linear logic [73]. While their notion of a shared channel corresponds closely to the one that we develop in our work (which may be unsurprising, since graded modal types are also built upon the foundation of bounded linear logic), their focus is more on accounting for nondeterminism and races in a logical setting while preserving deadlock freedom. Their paper contrasts the system they develop with other related work on *manifest sharing* [13]; said work uses modal operators to describe the interaction between linear and shared processes, and has greater expressivity when it comes to non-linear communication, but this comes at the cost of reintroducing some deadlocks.

GRADED SESSIONS IN LINEAR HASKELL We explored combining grading with linear session types in Granule, but this could be extended to other settings. The linear types extension to Haskell is based on a calculus involving graded function arrows [19], and adding additional multiplicities to this system is a possibility. The Priority Sesh library provides a convenient embedding of the GV linear session calculus in Haskell [72], and extending this to make use of more precise information about channel usage could be valuable. This would also

allow for experimenting with graded channels in a setting where grading can be implicit, rather than one where (like Granule) all grades are explicitly encoded via modalities.

7.5 DISCUSSION AND CONCLUSION

The key idea here is that graded types capture various standard non-linear forms of communication pattern atop the usual linear session types presentation. The demands of call-by-value required a different approach to past work on the interaction between graded types and session types, including the syntactic restriction discussed in Section 6.3 and vectors to capture multiplicity of channels. This connects back to similar ideas explored throughout Chapters 4 and 5.

We have also proven type safety and various essential communication properties given the additional restriction on promoting channels and the other features introduced here, showing in particular that we can still enforce communication safety in the presence of some forms of non-linear communication.

8 FURTHER WORK

Thus far, we have made significant progress on our primary goal: demonstrating the feasibility of using graded type systems to model and verify global program properties such as memory and communication safety. However, there are certainly plenty of remaining avenues for future work, expanding on the ideas we have developed up until this point and branching off in new directions. This chapter aims to provide seeds for some of these possibilities and offer some inspiration for further research.

8.1 GUARANTEES

Information flow control aims to track the flow of information through a program when it is executed, to make sure that the program handles that information in a secure way [121]. Secure information flow (discussed in the literature since the 1970s [37, 41]) encompasses multiple aspects, with two of the most essential being *confidentiality* and *integrity*. Pfleeger's textbook *Security in Computing* [110] describes these two properties as follows. Confidentiality "ensures that assets are accessed only by authorised parties", or in other words that private information is never accessed by a program which only has public clearance. Meanwhile, integrity "means that assets can be modified only by authorised parties or only in authorised ways", such that a trusted program never depends on information from an untrusted source. The strictest desirable property in both cases is *noninterference*, which only holds if public or trusted outputs may *never* depend on private or untrusted inputs respectively [52], though this is often considered difficult to achieve in practical systems.

Much of the prior work on using graded type systems for information flow control aims to track and restrict the outputs that can be produced by a given program [1, 58, 113]. Implementations of such ideas also exist for more widely-used functional languages such as Haskell [120]. More recently, graded type systems have been designed which enforce properties based on *coeffects*, where the inputs that can be passed into a program are the focus. Such systems (with the type system underlying Granule whose core we described in Chapter 3 being the one we will focus on here [103]) often make use of information flow security as a case study in how annotating types with grades can allow more properties of a program to be verified [2, 32, 47, 95]. However, these tend to only focus on confidentiality properties,

which omits a host of additional properties that could be guaranteed if it were also possible to enforce integrity. Consider the following definition of a data type in Granule.

```
data Patient where
  Patient
    (Int    [Hi]) -- Patient ID
    (String [Hi]) -- Patient name
    (Int    [Lo])  -- Patient age
```

Here, the patient's ID and name are private (`Hi` security), while their age is public (`Lo` security). Now, consider a function with type:

```
meanAge : List Patient → Int [Lo]
```

that calculates the mean age of a database (here simplified to a list) of patients and returns the result at the public security level. Since ages are also `Lo`, this is permitted by the type system–but if we made a mistake while writing the function and used private information such as IDs (the first component of the data structure) instead, this would be rejected due to the security annotations, with an error stating that `Lo is not approximatable by Hi`. Thus, no information can be leaked. Similar properties also apply when storing secret data such as passwords or credit card numbers.

However, imagine a case where we are constructing a patient to add to the database, as follows:

```
addPatient : List Patient → String [Hi]
          → Int [Lo] → List Patient
```

Here, we have security level grades restricting who will be able to view various details once the database has been updated, but we have no way to stop some compromised code from passing in an attempt at SQL injection using a string such as `"Alice'); DROP TABLE patients;-"`, for example. If this input is treated as trustworthy, we might encounter problems later in our program's execution[1].

In order to avoid this, we would want some kind of grade that carries information about the *provenance* of our data [77], so we could declare that we can only safely add patient information into our database if the string verifiably comes from a trusted location. This would also be useful if, for instance, we were encrypting private data and wanted a way to ensure that our random numbers used for encryption were reliable and had not been tainted by an untrusted source.

It is well understood that this kind of integrity property is dual in some sense to the confidentiality properties which Granule can already express [23] (though it is also known that this duality is not sufficient to cover every facet of the concept of integrity, with more complex mechanisms than a lattice model being required for some applications [34, 99]). It turns out that this duality is closely comparable to a similar duality between *linear* types (forming the

1 https://xkcd.com/327/

basis of Granule's graded type system) and *uniqueness* types, which has been elucidated in Chapter 4.

The crucial insight here is that in order to combine confidentiality (public and private) and integrity (trusted and untrusted) in a single system, we can treat "public" and "untrusted" as the *same state*, both represented by the Lo grade; these both carry the same information, telling us that there is no restriction on how the data may be used in the future (it is not restricted to private usage) and we have no guarantee about how it was used in the past (it did not necessarily come from a trusted source). The Hi grade behaves exactly as described in other works on Granule [79, 103], and in Chapter 3 of this thesis.

We introduce a new $*$ modality (whose syntax is borrowed from the corresponding modality for uniqueness types [91], as mentioned above) to carry the Trusted grade, with the important rules for this modality's behaviour given below.

$$\frac{\Gamma \vdash t : *_{\mathsf{Trusted}}\ A}{\Gamma \vdash \mathsf{reveal}\ t : \Box_{\mathsf{Lo}}\ A}\ \textsc{Reveal} \qquad \frac{\varnothing \vdash t : A}{[\Gamma] \vdash *t : *_{\mathsf{Trusted}}\ A}\ \textsc{Nec}$$

$$\frac{\Gamma_1 \vdash t_1 : \Box_{\mathsf{Lo}}\ A \quad \Gamma_2, x : *_{\mathsf{Trusted}}\ A \vdash t_2 : \Box_{\mathsf{Lo}}\ B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{endorse}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : \Box_{\mathsf{Lo}}\ B}\ \textsc{Endorse}$$

The Reveal rule maps a trusted value to a public (untrusted) one, allowing the information flow for integrity to behave as expected. The Endorse rule allows a public value to be temporarily used as trusted within the context of a larger computation; this mimics the common integrity pattern of *endorsement*, where a value is examined and declared to be trusted to whatever extent is necessary for a particular usage [100]. The output, however, is required to be public, so that we can not leverage our temporary integrity to 'smuggle out' a trusted value outside the context of the endorsement. These rules are accompanied by a necessitation rule, allowing values to be trusted by default if they have no dependencies.

Note that the naming and pattern of the rules suggests a structural relationship between the two modalities. The $\Box$ modality previously acted as a *graded comonad*, but now when graded by Lo also acts as a *relative monad* [9] over the $*$ modality graded by Trusted, with Reveal acting as the 'return' of the monad and Endorse acting as the 'bind'; the various required axioms for a relative monad hold also. This is very much analogous to the relative monad structure discussed in Chapter 4; the rules for the uniqueness modality from that chapter are presented below for reference and comparison.

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathsf{share}\ t : !A}\ \textsc{share} \qquad \frac{\varnothing \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \textsc{clone}$$

The implementation follows much the same pattern. Granule already possesses a *semiring graded necessity modality*, where for a preordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$, there is a family of types $\{\Box_r A\}_{r \in \mathcal{R}}$. Granule's confidentiality tracking is based on the semiring $\{\mathsf{Lo}, \mathsf{Hi}\}$, where $0 = \mathsf{Hi}$, $1 = \mathsf{Lo}$, $* = \vee$ and $+ = \wedge$. The implementation provides an additional $*$ modality obeying the above rules along with syntactic sugar for applying them. This allows us to repair the earlier example:

```
addPatient : List Patient → String *{Trusted}
           → Int [Lo] → List Patient
```

Note that we use the syntax `*{Trusted}` to represent the Trusted modality in Granule code.

Now, only a string from a trusted source will be accepted as the name of a patient added to our database. Note that we do not need to change the definition of our data type, since the string can be converted to Hi through a combination of the Reveal rule and the *approximation* rule that already exists in Granule's type system, which in this case allows Lo security information to be treated as Hi since $\mathsf{Hi} \sqsubseteq \mathsf{Lo}$ is given by the ordering of the semiring.

We have shown that simple integrity properties can be expressed in Granule alongside confidentiality, leveraging a similar duality to one that exists between linearity and uniqueness. This points towards a larger body of work that involves uncovering a general algebraic structure underlying *global guarantees* about a program, which are best represented using a graded comonad that also acts as a relative monad over some functor. This work is ongoing.

8.2 LINEAR HASKELL

Granule's linear basis and assortment of modalities allows for a particularly natural embedding of uniqueness and ownership properties, but this does not preclude the theory described in this thesis from being applied in other contexts. One particularly valuable setting to consider would be Haskell, which as of GHC 9 already has linear types based on an underlying graded system called $\lambda^q_{\to}$.

Haskell's graded representation of linearity involves function types `(a %r -> b)` which have a multiplicity annotation `r`. At present, this can be either `'One` (linear) or `'Many` (unrestricted). But $\lambda^q_{\to}$ is designed to be extensible, and the possibility of introducing additional multiplicities is welcomed [19, 129].

The original paper on linear Haskell [19] mentions that "linear types are conceptually simpler than uniqueness type systems, giving a clearer path to implementation in GHC", and also that "functional languages have more use for fusion than in-place update". Our clarification of the relationship between linearity and uniqueness demonstrates that not only are uniqueness types no more complex

conceptually than linear ones, they can comfortably sit alongside one another in a single calculus; our evaluation demonstrates that while linearity is certainly useful, there are still further practical benefits to be gained from introducing uniqueness into a language with linear types. Perhaps these contributions will begin to forge a path towards a future for Haskell where linear types, uniqueness types, ownership and borrowing can all be leveraged for their respective strengths.

8.3 ADJOINT MODELS

Benton's linear/non-linear (LNL) logic [17] consists of two fragments: intuitionistic (non-linear) logic $\Phi \vdash_{\mathcal{I}} X$ and a mixed fragment of intuitionistic linear logic with non-linear hypotheses $\Phi, \Gamma \vdash_{\mathcal{L}} A$. These fragments are connected by modalities $\mathrm{Lin}(X)$ and $\mathrm{Mny}(A)$, which form an adjunction; the ! modality can be recovered by $!A = \mathrm{Lin}(\mathrm{Mny}(A))$.

Breaking the ! modality into two and allowing linear logic to be mixed with non-linear logic has been a valuable endeavour, and this idea has even been adapted and built upon for modelling graded logics [137], so a natural question is whether it is possible to build an LNL-style adjoint model for uniqueness. It seems plausible that building an adjoint model for just uniqueness logic would not be too difficult; this would be very similar to the LNL model but with the adjunction moving in the opposite direction, and the monadic modality $\circ$ from uniqueness logic could be represented in much the same way that the comonadic ! can be recovered in LNL.

An adjoint model for the full LCU calculus presented in Chapter 4 would be more interesting. This would most likely involve three fragments, two of which would be symmetric monoidal categories (for unique and linear values) and one of which would be a Cartesian closed category (for unrestricted values), with two adjunctions allowing values to flow from unique to unrestricted to linear as desired.

8.4 ORDERED AND DEPENDENT TYPES

As expressive as Granule's type system may be, there are opportunities for enforcing stronger properties on programs elsewhere in the landscape of type theories. One possibility is that in addition to contraction and weakening, it is also possible to restrict *exchange*, giving ordered type theories which correspond to noncommutative logic.

Such systems can be used to model stack-based memory allocation (as opposed to heap-based), since without exchange an object may only be used when it is at the top of the stack [30, 148]. But much like linearity, these systems restrict the use of exchange in the *future*; is there an equivalent of uniqueness for ordered types which guarantees

that exchange has never been applied in the *past*, and could this be useful for tracking references on the stack?

Another possibility is to bring uniqueness into the realm of dependent types. Recent work on graded modal dependent type theory (GrTT) [95] allows for capturing requirements on variable usage at both the type and computation levels; grades come in pairs, where the first component is the computation-level grading and the second component is the type-level grading. Strictly linear usage in types is rare – is there value in being able to represent uniqueness here?

8.5 PERFORMANCE IMPROVEMENTS

Chapter 5 generalised uniqueness and linearity in order to develop a unified framework for reasoning about ownership and grading. While we have an implementation of this built upon the Granule compiler, the practical benefits it can offer for resourceful programming have not been fully explored. We hope to evaluate *performance improvements* that introducing precise ownership tracking into a functional language can offer through obviating garbage collection for a subset of a program, by collecting examples involving ownership and borrowing and benchmarking them against equivalent functional code.

8.6 COUNTING PERMISSIONS

Fractional permissions and the general notion of dividing a single mutable borrow into many immutable borrows is not the only model for representing borrowing that we considered. One interesting path for future research would be to explore an alternative strategy for graded uniqueness that is less symmetrical but instead privileges the original owner, allowing for a more exact count of other extant references. This model, based on *counting permissions* [25], involves having many references with only read permissions, and a designated owner that keeps a count of how many such references exist, so that uniqueness can still be recovered.

8.7 NON-LEXICAL LIFETIMES

The model of ownership developed in Chapter 5 was *not* intended to be a complete model of every aspect of Rust's ownership system, instead aiming to extract essential features and develop a framework where they coexist with more traditional linear and graded types. It would still be of interest to pursue this further and capture Rust's more advanced features—in particular, incorporating notions such as

non-lexical lifetimes would increase the power of our system. This could eventually allow for an encoding study between Granule's calculus and a more faithful representation of Rust's type system such as λ_{Rust} [69]. Moreover, it would be useful to uncover general principles for extending our type system with further resource interfaces, beyond the particular examples introduced in this work.

8.8 RECURSION AND OTHER COMBINATORS

A notable omission from the core session types calculus of Chapter 6 is the ability to define *recursive protocols*. However, some of the power of recursive session types is provided in Chapter 7; reusable channels (Section 7.2.1) are equivalent to linearly recursive session types (e.g., $\mu x.P.x$), although the expected usage has to be precisely specified in the grading since for example allowing the grade `r` to be instantiated to `0..`∞ to capture an arbitrary amount of use would break our guarantee of deadlock freedom. Further work is to integrate standard ideas on recursive session types and to explore their interaction with grading.

Note that combinations of the ideas here should also be possible; for example, combining multicast sending with reuse to get channels which we can repeatedly use to broadcast upon.

8.9 APPLICATIONS OF GRADED SESSIONS

As was discussed briefly in Chapter 6, by using the linear channels already present in Granule it is possible to represent functions that are *sequentially realizable* [90]; recall that a sequentially realizable function is one which has outwardly pure behaviour, but relies on a notion of local side effects which are contained entirely within the body of the function. Looking into which such behaviours may be more easily expressed by introducing non-linearity via graded channels would be an interesting avenue for future work.

More recently, the GV session type calculus has been extended to allow for *multiparty* session types [66], allowing for concurrent communication between several participants at a time. It has also been shown that this multiparty system is strictly more general than the original binary GV system. It would be interesting to explore an implementation of this in Granule, particularly with a view to introducing grading into this setting to see if it can increase expressivity further. Other recent work has begun to introduce certain flavours of non-linear communication into multiparty settings [28].

8.10 OTHER RESOURCE ALLOCATORS

Chapters 4 and 5 describe how to use a resource allocator restriction on promotion to treat the creation of mutable arrays and polymorphic references soundly under a call-by-value semantics, and Chapters 6 and 7 illustrate how to treat the forking of linear and graded channel endpoints using an analogous restriction.

These are most likely not the only resource allocators for which such a restriction applies. Future work is to develop a better understanding of the general case here, and to further explore the best way to embed interfaces for allocating resources in a graded type system such as Granule's which can express multiple different evaluation strategies.

8.11 UNIQUENESS AND GRADED SESSIONS

Uniqueness in a concurrent setting was considered in various works by de Vries et al., but in particular we draw attention to their paper on resource management via unique and affine channels [139]. They allow for a notion of a channel which we can guarantee unique access to after i communication steps where i is some natural number; this is a form of grading, though quite different from the ideas we have discussed. It would be interesting to explore how we can represent this in Granule which already has both uniqueness and graded session types, and whether we gain further expressivity from doing so, perhaps even through extending this to the fractional permissions discussed in Chapter 5. This idea closely relates to recent work by Rocha and Caires on introducing mutable state into the classical linear logic interpretation of session types [119], through typing rules inspired by differential linear logic.

8.12 DEADLOCK FREEDOM

Whilst the proofs of progress and preservation in Chapter 7 go some way to ensuring that our new communication primitives do not undermine the guarantees of the language, a further useful and important property to prove would be *deadlock freedom*. Whilst progress and preservation suggest that we don't become locally stuck, they do not rule out the possibility of some subset of processes becoming indefinitely blocked due to a mutual dependency. In our system these properties also rely on the *buffered* predicate, which may be stronger than necessary and therefore could forbid some programs which encapsulate valid communication. Proving deadlock freedom without such side conditions is future work, though we sketch some details here based on other approaches in the literature.

There have been various proofs of deadlock freedom for session-typed calculi with asynchronous communication. These approaches generalise duality to *compatibility*, which can be seen for example in the work of Fowler [44] and Bocchi et al. [24]. In essence, compatibility needs to take into account that asynchronous protocols may be 'out of sync' as one process sends to another, filling a channel's buffer without those messages being received yet. We can tie together the types of two channel endpoints P_1, P_2 with the contents of the associated channel values (buffers) $\vec{v}_1, \vec{v}_2$ via a relation:

$$\gamma \vdash c_1 : (P_1 - \vec{v}_1) \bowtie c_2 : (P_2 - \vec{v}_2)$$

meaning that P_1 is compatible with P_2 once the channel buffer values $\vec{v}_1$ and $\vec{v}_2$ have been 'stripped off' the head of the protocol. For example, a receiving protocol $\mathbf{Recv}\ A\ P$ for channel endpoint c is compatible with another protocol P' for the dual channel endpoint $\bar{c}$, if the channel has a value v of type A in its head, and then P and P' are compatible with that v removed:

$$\frac{\gamma \vdash c : (P - \vec{v}_1) \bowtie \bar{c} : (P' - \vec{v}_2) \qquad \gamma \vdash v : A}{\gamma \vdash c : ((\mathbf{Recv}\ A\ P) - v : \vec{v}_1) \bowtie \bar{c} : (P' - \vec{v}_2)}\ \bowtie_{\mathsf{RECV}}$$

Following Fowler [44], we then need to adapt process configuration typing into a typing of the entire configuration (process and channels), with parallel composition enforcing that between two processes there is exactly one pair of compatible endpoints connecting them, i.e.:

$$\frac{\begin{array}{c}\Gamma_1, \gamma_1, c : \mathbf{Chan}\, P \vdash t : A \qquad \Gamma_2, \gamma_2, \bar{c} : \mathbf{Chan}\, P' \vdash \mathcal{P} : \bot; \mathcal{C} \\ \gamma \vdash c : (P - \vec{v}_1) \bowtie \bar{c} : (P' - \vec{v}_2)\end{array}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2 + \gamma, c : \mathbf{Chan}\, P, \bar{c} : \mathbf{Chan}\, P' \vdash \mathcal{P} \mid t^\bullet : A; \mathcal{C}, c = \vec{v}_1, \bar{c} = \vec{v}_2}$$

From this 'global' configuration typing, we can then prove that there is a limit on the number of 'connection' points between processes:

Definition 12 (Connection). *For processes $\mathcal{P}_1$ and $\mathcal{P}_2$ then we define those channels endpoints that connect them via the function:*

$$\mathsf{connected}(\mathcal{P}_1, \mathcal{P}_2) = \bigcup \{\{c, \bar{c}\} \mid c \in \mathsf{fn}(\mathcal{P}_1), \bar{c} \in \mathsf{fn}(\mathcal{P}_2)\}$$

Lemma 12 (Limited connectivity). *If $\Gamma, \gamma \vdash \mathcal{P} : A; \mathcal{C}$ and $\mathcal{P} = \mathcal{P}' \mid \mathcal{P}_1 \mid \mathcal{P}_2$ then $\exists c.\mathsf{connections}(\mathcal{P}_1, \mathcal{P}_2) \subseteq \{c, \bar{c}\}$*

We note however that, for a full proof for our system, this lemma needs extending to handle forwarding processes which have a number of channels connecting processes, but where only sending is allowed.

Deadlock can then be characterised in terms of mutually dependent blocked processes, which are ruled out by the typing, similar to the formulation of Lindley and Morris for GV [80]. That is, a term t blocked on a channel c is a term which is equivalent to some evaluation context in which a receive is applied to c:

$$\mathsf{blocked}(t, c) \triangleq \exists Ctxt.\ t \equiv Ctxt\{\mathsf{receive}\, c\} \vee t \equiv Ctxt\{\mathsf{close}\, c\}$$

Thus, the term t can only reduce further in composition with another a global configuration. A dependency can then be characterised as a term which is:

$$\begin{aligned}\mathsf{depends}(t, c, x) &\triangleq \mathsf{blocked}(t, c) \ \wedge \ x \in \mathsf{fv}(Ctxt\{t\}) \\ \mathsf{depends}(\mathcal{P}, c, x) &\triangleq (\exists t, \mathcal{P}'.\ \mathcal{P} \equiv (\mathcal{P}' \mid t^{\mathcal{R}}) \ \wedge \ \mathsf{depends}(t, c, x)) \\ &\quad \vee (\exists \mathcal{P}_1, \mathcal{P}_2.\ \mathcal{P} \equiv (\mathcal{P}_1 \mid \mathcal{P}_2) \\ &\quad \wedge \ \exists d.\, \mathsf{depends}(\mathcal{P}_1, c, d) \wedge \mathsf{depends}(\mathcal{P}_2, d, x))\end{aligned}$$

The main definition of a deadlocked configuration with cyclic dependencies can then be characterised as:

$$\begin{aligned}\mathsf{deadlocked}(\mathcal{P}) &\triangleq \exists \mathcal{P}', \mathcal{P}_1, \mathcal{P}_2.\ \mathcal{P} \equiv (\mathcal{P}' \mid \mathcal{P}_1 \mid \mathcal{P}_2) \\ &\quad \wedge \ \exists c, d.\mathsf{depends}(\mathcal{P}_1, c, d) \wedge \mathsf{depends}(\mathcal{P}_2, d, c)\end{aligned}$$

From Lemma 12, we conjecture that this then enables a proof of the deadlock freedom theorem:

Theorem conjecture 1. *If* $\Gamma, \gamma \vdash \mathcal{P} : A; \mathcal{C}$ *then* $\neg\mathsf{deadlocked}(\mathcal{P})$.

Further work remains to flesh out the details of Lemma 12 for forwarding, and to integrate the global configuration typing (processes and channels together, with compatibility) into the progress and preservation theorems here, rather than their current formulation with process and channel configuration typing separate.

9 CONCLUSIONS

9.1 SUMMARY

In this dissertation, we have demonstrated how precise forms of grading can be used to track crucial program properties, allowing for additional information to be encoded at the type level alongside values which should be treated as resources, and applied this notion to two paradigms that are crucial to the safety of the complex programs we write in the modern day: memory and communication safety.

In the thesis statement presented in Chapter 1, we posited that graded type systems can not only be used to track local properties of values inside programs, but can also be used to verify global properties pertaining to an entire program. We have illustrated this throughout the thesis by providing frameworks which allow for several such examples of global properties to be encoded entirely in types via the use of grades, and via the implementation in Granule we have also demonstrated some of the benefits to practical programming.

To summarise, this thesis has made the following key contributions:

- In Chapter 4, a type system incorporating both the substructural notions of linearity and uniqueness, showing that these concepts can be combined in a single calculus and that the resulting system is practical and useful for reasoning about programs.
- In Chapter 4, a heap-based call-by-name operational semantics for the above type system, allowing the guarantees it offers with regards to both type and memory safety to be verified.
- In Chapter 5, an extension of the above type system through a new form of grading based on fractional permissions, allowing for the tracking of mutable and immutable borrowing in a manner similar to Rust's ownership model.
- In Chapter 5, a call-by-value operational semantics for this extended system, both showing that the guarantees continue to hold given the new level of flexibility and also that it is still possible to verify these guarantees in a more practical setting.
- In Chapter 6, a formalisation of linear session types in a graded setting, using a similar operational model to show that safety properties can be enforced which prevent processes from getting stuck in many communication scenarios.

- In Chapter 7, an extension of the above system to allow for non-linear communication, through the addition of several new primitives and corresponding semantic constructs.

There are still many possibilities for future work in this area, as we have described in Chapter 8. We hope that practitioners in both programming language research but also those working on the implementation of practical languages in industry will find value in the frameworks we have presented here.

9.2 DISCUSSION

Many important properties that apply to programs are global, in the sense that verifying their correctness requires information about the state of the whole program; two examples of such properties that have been widely studied in the literature are memory and communication safety. We hope that we have been able to present a new perspective on these properties, with a theory based on graded types that is relatively simple compared to other models and yet captures the essential core of these ideas.

We believe that from the perspective of the programmer, reasoning about global properties that hold across an entire program's state is often difficult, particularly in the presence of concurrency and mutable state. Hence, there is great value in being able to reason locally about individual pieces of data through their grades, and then having it be possible to reassemble that local information into an overarching global guarantee. We have demonstrated that this is possible for some key memory and communication safety properties, and we aim to generalise this concept further in the future.

The work accomplished here also serves as an illustration of the power and utility that graded types and graded modalities can offer, both as a theoretical underpinning for programming language research but also as a practical tool for ensuring safety and correctness in an elegant and extensible way. We are excited to see how the framework of graded types is applied in the future, towards the verification of any and all kinds of properties; we invite the curious reader to explore how grades could be used to track whatever information they might be interested in.

BIBLIOGRAPHY

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. "A Core Calculus of Dependency." In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 147–160. ISBN: 1581130953. DOI: 10.1145/292540.292555.

[2] Andreas Abel and Jean-Philippe Bernardy. "A Unified View of Modalities in Type Systems." In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 90:1–90:28. DOI: 10.1145/3408972.

[3] Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. "A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized." In: *Proc. ACM Program. Lang.* 7.ICFP (2023), pp. 920–954. DOI: 10.1145/3607862.

[4] Samson Abramsky. "Computational interpretations of linear logic." In: *Theoretical Computer Science* 111.1 (1993), pp. 3–57. ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90181-R.

[5] Benedikt Ahrens and Régis Spadotti. "Terminal Semantics for Codata Types in Intensional Martin-Löf Type Theory." English. In: *20th International Conference on Types for Proofs and Programs (TYPES 2014)*. Ed. by Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau. Vol. 39. Leibniz International Proceedings in Informatics (LIPIcs). Germany: Schloss Dagstuhl, Oct. 2015, pp. 1–26. ISBN: 978-3-939897-88-0. DOI: 10.4230/LIPIcs.TYPES.2014.1.

[6] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. "Alias Annotations for Program Understanding." In: *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '02. Seattle, Washington, USA: Association for Computing Machinery, 2002, pp. 311–330. ISBN: 1581134711. DOI: 10.1145/582419.582448.

[7] Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. *Polymorphic Context-free Session Types*. 2021. arXiv: 2106.06658 [cs.PL].

[8] Paulo Sérgio Almeida. "Balloon Types: Controlling Sharing of State in Data Types." In: *ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11*. Springer. 1997, pp. 32–59. DOI: 10.1007/BFb0053373.

[9] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. "Monads Need Not be Endofunctors." In: *Logical Methods in Computer Science* 11.1 (Mar. 2015). Ed. by LukeEditor Ong. ISSN: 1860-5974. DOI: 10.2168/lmcs-11(1:3)2015.

[10] Robert Atkey. "Syntax and Semantics of Quantitative Type Theory." In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 2018, pp. 56–65. DOI: 10.1145/3209108.3209189.

[11] Henry G. Baker. "Lively Linear Lisp: "Look Ma, No Garbage!"" In: *SIGPLAN Not.* 27.8 (Aug. 1992), pp. 89–98. ISSN: 0362-1340. DOI: 10.1145/142137.142162.

[12] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. "The Design and Formalization of Mezzo, a Permission-Based Programming Language." In: *ACM Trans. Program. Lang. Syst.* 38.4 (Aug. 2016). ISSN: 0164-0925. DOI: 10.1145/2837022.

[13] Stephanie Balzer and Frank Pfenning. "Manifest Sharing with Session Types." In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017). DOI: 10.1145/3110281.

[14] Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. "Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs." In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485516.

[15] Erik Barendsen and Sjaak Smetsers. "Conventional and uniqueness typing in graph rewrite systems." In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 1993, pp. 41–51.

[16] Erik Barendsen and Sjaak Smetsers. "Uniqueness Typing for Functional Languages with Graph Rewriting Semantics." In: *Mathematical Structures in Computer Science* 6.6 (1996), pp. 579–612. DOI: 10.1017/S0960129500070109.

[17] P Nick Benton. "A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models." In: *Computer Science Logic: 8th Workshop, CSL'94 Kazimierz, Poland, September 25–30, 1994 Selected Papers 8*. Springer. 1995, pp. 121–135. DOI: 10.1007/BFb0022251.

[18] Martin Berger, Kohei Honda, and Nobuko Yoshida. "Sequentiality and the π-calculus." In: *International Conference on Typed Lambda Calculi and Applications*. Springer. 2001, pp. 29–45. DOI: 10.1007/3-540-45413-6_7.

[19] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. "Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language." In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–29. DOI: 10.1145/3158093.

[20] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. "Multi-Graded Featherweight Java." In: *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 3:1–3:27. DOI: 10.4230/LIPIcs.ECOOP.2023.3.

[21] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. "Resource-Aware Soundness for Big-Step Semantics." In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622843.

[22] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, and Marco Servetto. "Coeffects for Sharing and Mutation." In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022). DOI: 10.1145/3563319.

[23] Kenneth J Biba. *Integrity Considerations for Secure Computer Systems*. Tech. rep. MTR-3153. The Mitre Corporation, 1975.

[24] Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. "Asynchronous Timed Session Types - From Duality to Time-Sensitive Processes." In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Luís Caires. Vol. 11423. Lecture Notes in Computer Science. Springer, 2019, pp. 583–610. DOI: 10.1007/978-3-030-17184-1_21.

[25] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. "Permission Accounting in Separation Logic." In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, California, USA: Association for Computing Machinery, 2005, pp. 259–270. ISBN: 158113830X. DOI: 10.1145/1040305.1040327.

[26] John Boyland. "Checking Interference with Fractional Permissions." In: *Static Analysis: 10th International Symposium, SAS 2003 San Diego, CA, USA, June 11–13, 2003 Proceedings*. Springer. 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4.

[27] Edwin C. Brady. *Idris 2: Quantitative Type Theory in Practice*. Ed. by Anders Møller and Manu Sridharan. 2021. DOI: 10.4230/LIPIcs.ECOOP.2021.9.

[28] Matthew Alan Le Brun, Simon Fowler, and Ornela Dardha. *Multiparty Session Types with a Bang!* 2025. arXiv: 2501.14702 [cs.PL].

[29] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. "A Core Quantitative Coeffect Calculus." In: *ESOP*. Vol. 8410. Springer. 2014, pp. 351–370. DOI: 10.1007/978-3-642-54833-8_19.

[30] Aubrey B. Bryant and Harley D. Eades. "The Graded Lambek Calculus." In: *2020 Joint Workshop on Linearity & TLLA: The 6th Workshop on Linearity and the 4th Workshop on Trends in Linear Logic and Applications*. 2020. URL: https://www.cs.unibo.it/~dallago/TLLALINEARITY2020/The_Graded_Lambek_Calculus.pdf.

[31] Luís Caires and Frank Pfenning. "Session Types as Intuitionistic Linear Propositions." In: *Proceedings of the 21st International Conference on Concurrency Theory*. CONCUR'10. Paris, France: Springer-Verlag, 2010, pp. 222–236. ISBN: 3642153747.

[32] Pritam Choudhury, Harley Eades III, Richard A Eisenberg, and Stephanie Weirich. "A Graded Dependent Type System with a Usage-Aware Semantics." In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–32. DOI: 10.1145/3434331.

[33] Vikraman Choudhury and Neel Krishnaswami. "Recovering purity with comonads and capabilities." In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 111:1–111:28. DOI: 10.1145/3408993.

[34] David D. Clark and David R. Wilson. "A Comparison of Commercial and Military Computer Security Policies." In: *1987 IEEE Symposium on Security and Privacy*. 1987, p. 184. DOI: 10.1109/SP.1987.10001.

[35] Dave Clarke and Tobias Wrigstad. "External Uniqueness is Unique Enough." In: *ECOOP 2003–Object-Oriented Programming: 17th European Conference, Darmstadt, Germany, July 21-25, 2003. Proceedings 17*. Springer. 2003, pp. 176–200. DOI: 10.1007/978-3-540-45070-2_9.

[36] David G. Clarke, John M. Potter, and James Noble. "Ownership Types for Flexible Alias Protection." In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '98. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1998, pp. 48–64. ISBN: 1581130058. DOI: 10.1145/286936.286947.

[37] Ellis S Cohen. "Information Transmission in Sequential Programs." In: *Foundations of Secure Computation* (1978), pp. 297–335.

[38] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. "Stream Fusion: From Lists to Streams to Nothing at All." In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP '07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 315–326. ISBN: 9781595938152. DOI: 10.1145/1291151.1291199.

[39] Loris D'Antoni, Marco Gaboardi, Emilio Jesús Gallego Arias, Andreas Haeberlen, and Benjamin C. Pierce. "Sensitivity analysis using type-based constraints." In: *Proceedings of the 1st Annual Workshop on Functional Programming Concepts in Domain-Specific Languages, FPCDSL@ICFP 2013, Boston, Massachusetts, USA, September 22, 2013*. 2013, pp. 43–50. DOI: 10.1145/2505351.2505353.

[40] Ugo de'Liguoro and Luca Padovani. "Mailbox Types for Unordered Interactions." In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Ed. by Todd Millstein. Vol. 109. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 15:1–15:28. ISBN: 978-3-95977-079-8. DOI: 10.4230/LIPIcs.ECOOP.2018.15.

[41] Dorothy E. Denning and Peter J. Denning. "Certification of Programs for Secure Information Flow." In: *Commun. ACM* 20.7 (July 1977), pp. 504–513. ISSN: 0001-0782. DOI: 10.1145/359636.359712.

[42] Matthew Fluet, Greg Morrisett, and Amal Ahmed. "Linear Regions Are All You Need." In: *Programming Languages and Systems*. Ed. by Peter Sestoft. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 7–21. ISBN: 978-3-540-33096-7. DOI: 10.1007/11693024_2.

[43] Simon Fowler, Duncan Paul Attard, Danielle Marshall, Simon J. Gay, and Phil Trinder. *Special Delivery: Programming with Mailbox Types (Extended Version)*. 2025. arXiv: 2306.12935 [cs.PL].

[44] Simon John Fowler. "Typed concurrent functional programming with channels, actors and sessions." PhD thesis. University of Edinburgh, 2019. URL: https://era.ed.ac.uk/handle/1842/35873.

[45] Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. "Towards a Formal Theory of Graded Monads." In: *Foundations of Software Science and Computation Structures*. Ed. by Bart Jacobs and Christof Löding. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 513–530. ISBN: 978-3-662-49630-5. DOI: 10.1007/978-3-662-49630-5_30.

[46] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. "Linear dependent types for differential privacy." In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. 2013, pp. 357–370. DOI: 10.1145/2429069.2429113.

[47] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. "Combining Effects and Coeffects via Grading." In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 476–489. ISBN: 9781450342193. DOI: 10.1145/2951913.2951939.

[48] Simon J. Gay and Vasco Thudichum Vasconcelos. "Linear type theory for asynchronous session types." In: *J. Funct. Program.* 20.1 (2010), pp. 19–50. DOI: 10.1017/S0956796809990268.

[49] Dan R. Ghica and Alex I. Smith. "Bounded Linear Types in a Resource Semiring." In: *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 331–350. ISBN: 9783642548321. DOI: 10.1007/978-3-642-54833-8_18.

[50] Jean-Yves Girard. "Linear Logic." In: *Theoretical Computer Science* 50.1 (1987), pp. 1–101. DOI: 10.1016/0304-3975(87)90045-4.

[51] Jean-Yves Girard, Andre Scedrov, and Philip J Scott. "Bounded Linear Logic: A Modular Approach to Polynomial-Time Computability." In: *Theoretical Computer Science* 97.1 (1992), pp. 1–66. DOI: 10.1016/0304-3975(92)90386-T.

[52] J. A. Goguen and J. Meseguer. "Security Policies and Security Models." In: *1982 IEEE Symposium on Security and Privacy*. 1982, p. 11. DOI: 10.1109/SP.1982.10014.

[53] Georges Gonthier et al. "Formal proof–the four-color theorem." In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393.

[54] Colin S. Gordon. "Designing with Static Capabilities and Effects: Use, Mention, and Invariants (Pearl)." In: *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 10:1–10:25. ISBN: 978-3-95977-154-2. DOI: 10.4230/LIPIcs.ECOOP.2020.10.

[55] J.C. Guzman and P. Hudak. "Single-threaded polymorphic lambda calculus." In: *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*. 1990, pp. 333–343. DOI: 10.1109/LICS.1990.113759.

[56] Philipp Haller and Martin Odersky. "Capabilities for Uniqueness and Borrowing." In: *ECOOP 2010 – Object-Oriented Programming*. Ed. by Theo D'Hondt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 354–378. ISBN: 978-3-642-14107-2.

[57] Dana Harrington. "Uniqueness Logic." In: *Theoretical Computer Science* 354.1 (2006), pp. 24–41. DOI: 10.1016/j.tcs.2005.11.006.

[58] Nevin Heintze and Jon G. Riecke. "The SLam Calculus: Programming with Secrecy and Integrity." In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. San Diego, California, USA: Association for Computing Machinery, 1998, pp. 365–377. ISBN: 0897919793. DOI: 10.1145/268946.268976. URL: https://doi.org/10.1145/268946.268976.

[59] Bas van den Heuvel and Jorge A Pérez. "Session Type Systems based on Linear Logic: Classical versus Intuitionistic." In: *PLACES@ ETAPS* 314 (2020), pp. 1–11.

[60] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. "Experience with Safe Manual Memory-Management in Cyclone." In: *Proceedings of the 4th International Symposium on Memory Management*. ISMM '04. Vancouver, BC, Canada: Association for Computing Machinery, 2004, pp. 73–84. ISBN: 1581139454. DOI: 10.1145/1029873.1029883.

[61] John Hogg. "Islands: Aliasing Protection in Object-Oriented Languages." In: *SIGPLAN Not.* 26.11 (Nov. 1991), pp. 271–285. ISSN: 0362-1340. DOI: 10.1145/118014.117975.

[62] Kohei Honda. "Types for dyadic interaction." In: *CONCUR'93*. Ed. by Eike Best. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 509–523. ISBN: 978-3-540-47968-0. DOI: 10.1007/3-540-57208-2_35.

[63] Jack Hughes, Danielle Marshall, James Wood, and Dominic Orchard. "Linear Exponentials as Graded Modal Types." In: *5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021)*. Rome (virtual), Italy, June 2021. URL: https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271465.

[64] Jack Hughes and Dominic Orchard. "Program Synthesis from Graded Types." In: *Programming Languages and Systems*. Ed. by Stephanie Weirich. Cham: Springer Nature Switzerland, 2024, pp. 83–112. ISBN: 978-3-031-57262-3.

[65] Jack Hughes, Michael Vollmer, and Dominic Orchard. "Deriving Distributive Laws for Graded Linear Types." In: *Electronic Proceedings in Theoretical Computer Science* 353 (Dec. 2021), pp. 109–131. DOI: 10.4204/eptcs.353.6.

[66] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. "Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom." In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547638.

[67] Pierre Jouvelot and David Gifford. "Algebraic Reconstruction of Types and Effects." In: *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '91. Orlando, Florida, USA: Association for Computing Machinery, 1991, pp. 303–310. ISBN: 0897914198. DOI: 10.1145/99583.99623.

[68] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. "Stacked Borrows: An Aliasing Model for Rust." In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–32. DOI: 10.1145/3371109.

[69] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language." In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–34. DOI: 10.1145/3158154.

[70] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. "Regular, Shape-Polymorphic, Parallel Arrays in Haskell." In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: Association for Computing Machinery, 2010, pp. 261–272. ISBN: 9781605587943. DOI: 10.1145/1863543.1863582.

[71] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. "seL4: Formal verification of an OS kernel." In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 2009, pp. 207–220.

[72] Wen Kokke and Ornela Dardha. "Deadlock-Free Session Types in Linear Haskell." In: *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*. Haskell 2021. Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 1–13. ISBN: 9781450386159. DOI: 10.1145/3471874.3472979.

[73] Wen Kokke, J Garrett Morris, and Philip Wadler. "Towards Races in Linear Logic." In: *Coordination Models and Languages: 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings*. Springer. 2019, pp. 37–53.

[74] Herb Krasner. "The cost of poor software quality in the US: A 2022 report." In: *Proc. Consortium Inf. Softw. QualityTM (CISQTM)* (2023), pp. 1–61. URL: https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf.

[75] Xavier Leroy. "Formal verification of a realistic compiler." In: *Communications of the ACM* 52.7 (2009), pp. 107–115.

[76] Nancy G Leveson and Clark S Turner. "An investigation of the Therac-25 accidents." In: *Computer* 26.7 (1993), pp. 18–41.

[77] Peng Li and Steve Zdancewic. "Unifying Confidentiality and Integrity in Downgrading Policies." In: *In Proc. of the LICS'05 Affiliated Workshop on Foundations of Computer Security (FCS)*. Citeseer. 2005.

[78] Vilem-Benjamin Liepelt, Danielle Marshall, and Dominic Orchard. "Same Coeffect, Different Base: Connecting Two Dominant Approaches to Graded Types." under review. 2026.

[79] Vilem-Benjamin Liepelt, Danielle Marshall, Dominic Orchard, Vineet Rajani, and Michael Vollmer. "On Graded Coeffect Types for Information-Flow Control." In: *Languages, Compilers, Analysis - From Beautiful Theory to Useful Practice: Essays Dedicated to Alan Mycroft on the Occasion of His Retirement*. Ed. by Dominic Orchard, Tomas Petricek, and Jeremy Singer. Cham: Springer Nature Switzerland, 2026, pp. 114–148. ISBN: 978-3-032-08187-2. DOI: 10.1007/978-3-032-08187-2_7.

[80] Sam Lindley and J Garrett Morris. "A semantics for propositions as sessions." In: *European Symposium on Programming Languages and Systems*. Springer. 2015, pp. 560–584. DOI: 10.1007/978-3-662-46669-8_23.

[81] Sam Lindley and J Garrett Morris. "Lightweight functional session types." In: *Behavioural Types: from Theory to Tools. River Publishers* (2017), pp. 265–286.

[82] John Longley. "When is a functional program not a functional program?" In: *ACM SIGPLAN Notices*. Vol. 34. 9. ACM. 1999, pp. 1–7.

[83] John Longley. "The sequentially realizable functionals." In: *Ann. Pure Appl. Log.* 117.1-3 (2002), pp. 1–93. DOI: 10.1016/S0168-0072(01)00110-5.

[84] Anton Lorenzen, Daan Leijen, and Wouter Swierstra. "FP^2: Fully in-Place Functional Programming." In: *Proc. ACM Program. Lang.* 7.ICFP (Aug. 2023). DOI: 10.1145/3607840.

[85] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. "Oxidizing OCaml with Modal Memory Management." In: *Proc. ACM Program. Lang.* 8.ICFP (Aug. 2024). DOI: 10.1145/3674642.

[86] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics." In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 329–339. ISBN: 9781595939586. DOI: 10.1145/1346281.1346323.

[87] J. M. Lucassen and D. K. Gifford. "Polymorphic Effect Systems." In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 47–57. ISBN: 0897912527. DOI: 10.1145/73560.73564.

[88] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. "Exploiting Vector Instructions with Generalized Stream Fusion." In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 37–48. ISBN: 9781450323260. DOI: 10.1145/2500365.2500601.

[89] Dhruv C Makwana and Neel Krishnaswami. "NumLin: Linear Types for Linear Algebra." In: (2019). DOI: 10.4230/LIPICS.ECOOP.2019.14.

[90] Danielle Marshall and Dominic Orchard. "How to Take the Inverse of a Type." In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Ed. by Karim Ali and Jan Vitek. Vol. 222. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 5:1–5:27. ISBN: 978-3-95977-225-9. DOI: 10.4230/LIPIcs.ECOOP.2022.5.

[91] Danielle Marshall, Michael Vollmer, and Dominic Orchard. "Linearity and Uniqueness: An Entente Cordiale." In: *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings*. Springer International Publishing Cham. 2022, pp. 346–375. DOI: 10.1007/978-3-030-99336-8_13.

[92] Nicholas D Matsakis and Felix S Klock II. "The Rust language." In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.

[93] Conor McBride. "The Derivative of a Regular Type is its Type of One-Hole Contexts." In: *Unpublished manuscript* (2001), pp. 74–88. URL: http://strictlypositive.org/diff.pdf.

[94] Conor McBride. "I Got Plenty o' Nuttin'." In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday* (2016), pp. 207–233. DOI: 10.1007/978-3-319-30936-1_12.

[95] Benjamin Moon, Harley Eades III, and Dominic Orchard. "Graded Modal Dependent Type Theory." In: *Programming Languages and Systems* 12648 (2021), p. 462. DOI: 10.1007/978-3-030-72019-3_17.

[96] J. Garrett Morris. "The Best of Both Worlds: Linear Functional Programming without Compromise." In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 448–461. ISBN: 9781450342193. DOI: 10.1145/2951913.2951925.

[97] Greg Morrisett, Amal Ahmed, and Matthew Fluet. "L^3: A Linear Language with Locations." In: *Typed Lambda Calculi and Applications*. Ed. by Paweł Urzyczyn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 293–307. ISBN: 978-3-540-32014-2.

[98] Alan Mycroft and Janina Voigt. "Notions of Aliasing and Ownership." In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Ed. by Dave Clarke, James Noble, and Tobias Wrigstad. Vol. 7850. Lecture Notes in Computer Science. Springer, 2013, pp. 59–83. DOI: 10.1007/978-3-642-36946-9_4.

[99] Andrew C. Myers and Barbara Liskov. "Protecting Privacy Using the Decentralized Label Model." In: *ACM Trans. Softw. Eng. Methodol.* 9.4 (Oct. 2000), pp. 410–442. ISSN: 1049-331X. DOI: 10.1145/363516.363526.

[100] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. "Enforcing Robust Declassification and Qualified Robustness." In: *Journal of Computer Security* 14.2 (2006), pp. 157–196. DOI: 10.3233/JCS-2006-14203.

[101] Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. "Cogent: Uniqueness Types and Certified Compilation." In: *J. Funct. Program.* (2021). DOI: 10.1017/S095679682100023X.

[102] Office of the National Cyber Director. *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. Tech. rep. The White House, Feb. 2024. URL: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.

[103] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. "Quantitative Program Reasoning with Graded Modal Types." In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–30. DOI: 10.1145/3342537.

[104] Dominic Orchard and Alan Mycroft. "Categorical programming for data types with restricted parametricity." In: *Unpublished note.* (2012). URL: https://www.cs.kent.ac.uk/people/staff/dao7/drafts/tfp-structures-orchard12.pdf.

[105] Dominic Orchard, Tomas Petricek, and Alan Mycroft. *The Semantic Marriage of Monads and Effects*. 2014. arXiv: 1401.5391 [cs.PL].

[106] Jennifer Paykin and Steve Zdancewic. "The Linearity Monad." In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. Haskell 2017. Oxford, UK: Association for Computing Machinery, 2017, pp. 117–132. ISBN: 9781450351829. DOI: 10.1145/3122955.3122965.

[107] David J Pearce. "A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43.1 (2021), pp. 1–73. DOI: 10.1145/3443420.

[108] Tomas Petricek, Dominic Orchard, and Alan Mycroft. "Coeffects: A Calculus of Context-Dependent Computation." In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 123–135. ISBN: 9781450328739. DOI: 10.1145/2628136.2628160.

[109] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. "Simple unification-based type inference for GADTs." In: *ACM SIGPLAN Notices*. Vol. 41. 9. ACM. 2006, pp. 50–61.

[110] Charles P. Pfleeger. *Security in Computing*. Prentice-Hall, Inc., 1988. ISBN: 0137989431.

[111] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.

[112] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. "Clean Version 2.2 Language Report." In: (2011). URL: https://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf.

[113] François Pottier and Vincent Simonet. "Information Flow Inference for ML." In: *ACM Trans. Program. Lang. Syst.* 25.1 (Jan. 2003), pp. 117–158. ISSN: 0164-0925. DOI: 10.1145/596980.596983. URL: https://doi.org/10.1145/596980.596983.

[114] Dag Prawitz. "Natural Deduction: A proof-theoretical study." In: *Stockholm Studies in Philosophy. Almqvist & Wiksell, Stockholm* 3 (1965).

[115] Klaas Pruiksma and Frank Pfenning. "A Message-Passing Interpretation of Adjoint Logic." In: *Electronic Proceedings in Theoretical Computer Science* 291 (Apr. 2019), pp. 60–79. ISSN: 2075-2180. DOI: 10.4204/eptcs.291.6.

[116] Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. "Kindly Bent to Free Us." In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3408985.

[117] Alex Rebert and Christoph Kern. "Secure by design: Google's perspective on memory safety." In: *Technical report, Google Security Engineering* (2024).

[118] Greg Restall. *An introduction to substructural logics*. Routledge, 2002.

[119] Pedro Rocha and Luís Caires. "Propositions-as-Types and Shared State." In: *Proc. ACM Program. Lang.* 5.ICFP (Aug. 2021). DOI: 10.1145/3473584. URL: https://doi.org/10.1145/3473584.

[120] Alejandro Russo. "Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell." In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 280–288. ISBN: 9781450336697. DOI: 10.1145/2784731.2784756.

[121] A. Sabelfeld and A.C. Myers. "Language-Based Information-Flow Security." In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121.

[122] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types." In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 158–174. DOI: 10.1145/3453483.3454036.

[123] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.

[124] Sven-Bodo Scholz. "Single Assignment C – Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen." Shaker Verlag, Aachen, 1997. PhD thesis. Kiel, Germany: Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, 1996. URL: https://www.sac-home.org/_media/publications:pdf:sac-design-sbs-phd-96.pdf.

[125] Peter Selinger and Benoît Valiron. "Quantum Lambda Calculus." In: *Semantic Techniques in Quantum Computation*. Ed. by Simon Gay and Ian Mackie. Cambridge University Press, 2009, pp. 135–172. DOI: 10.1017/CBO9781139193313.005.

[126] Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. "Guaranteeing Safe Destructive Updates Through a Type System with Uniqueness Information for Graphs." In: *Graph Transformations in Computer Science*. Ed. by Hans Jürgen Schneider and Hartmut Ehrig. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 358–379. ISBN: 978-3-540-48333-5. DOI: 10.1007/3-540-57787-4_23.

[127] AL Smirnov. "Graded monads and rings of polynomials." In: *Journal of Mathematical Sciences* 151.3 (2008), pp. 3032–3051.

[128] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. "The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language." In: *The Journal of Logic Programming* 29.1 (1996). High-Performance Implementations of Logic Programming Systems, pp. 17–64. ISSN: 0743-1066. DOI: 10.1016/S0743-1066(96)00068-4.

[129] Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. "Linearly Qualified Types: Generic Inference for Capabilities and Uniqueness." In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547626.

[130] K. Terui. "Light Affine Lambda Calculus and Polytime Strong Normalization." In: LICS '01. IEEE Computer Society, 2001, pp. 209–220.

[131] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. "A Retrospective on Region-Based Memory Management." In: *Higher-Order and Symbolic Computation* 17.3 (Sept. 2004), pp. 245–265. ISSN: 1573-0557. DOI: 10.1023/B:LISP.0000029446.78563.a4.

[132] Bernardo Toninho, Luís Caires, and Frank Pfenning. "A Decade of Dependent Session Types." In: *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*. Ed. by Niccolò Veltri, Nick Benton, and Silvia Ghilezan. ACM, 2021, 3:1–3:3. DOI: 10.1145/3479394.3479398.

[133] Bernardo Toninho and Nobuko Yoshida. "Depending on Session-Typed Processes." In: *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Christel Baier and Ugo Dal Lago. Vol. 10803. Lecture Notes in Computer Science. Springer, 2018, pp. 128–145. DOI: 10.1007/978-3-319-89366-2_7.

[134] Jesse A. Tov and Riccardo Pucella. "Practical Affine Types." In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin,

Texas, USA: Association for Computing Machinery, 2011, pp. 447–458. ISBN: 9781450304900. DOI: 10.1145/1926385.1926436.

[135] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. "Understanding Real-World Concurrency Bugs in Go." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 865–878. ISBN: 9781450362405. DOI: 10.1145/3297858.3304069.

[136] Vasco T. Vasconcelos. "Fundamentals of Session Types." In: *Information and Computation* 217 (2012), pp. 52–70. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2012.05.002. URL: https://www.sciencedirect.com/science/article/pii/S0890540112001022.

[137] Victoria Vollmer, Danielle Marshall, Harley Eades III, and Dominic Orchard. "A Mixed Linear and Graded Logic: Proofs, Terms, and Models." In: *33rd EACSL Annual Conference on Computer Science Logic (CSL 2025)*. Ed. by Jörg Endrullis and Sylvain Schmitz. Vol. 326. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 32:1–32:21. ISBN: 978-3-95977-362-1. DOI: 10.4230/LIPIcs.CSL.2025.32.

[138] Edsko de Vries. "Modelling Unique and Affine Typing Using Polymorphism." In: *Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday on The Beauty of Functional Code - Volume 8106*. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 181–192. ISBN: 9783642403545. DOI: 10.1007/978-3-642-40355-2{_}13.

[139] Edsko de Vries, Adrian Francalanza, and Matthew Hennessy. "Uniqueness typing for resource management in message-passing concurrency." In: *Journal of Logic and Computation* 24.3 (June 2012), pp. 531–556. ISSN: 0955-792X. DOI: 10.1093/logcom/exs022. eprint: https://academic.oup.com/logcom/article-pdf/24/3/531/2782308/exs022.pdf.

[140] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. "Uniqueness Typing Redefined." In: *Proceedings of the 18th International Conference on Implementation and Application of Functional Languages*. IFL'06. Budapest, Hungary: Springer-Verlag, 2006, pp. 181–198. ISBN: 9783540741299.

[141] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. "Uniqueness Typing Simplified." In: *Implementation and Application of Functional Languages*. Ed. by Olaf Chitil, Zoltán Horváth, and Viktória Zsók. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 201–218. ISBN: 978-3-540-85373-2. DOI: 10.1007/978-3-540-85373-2_12.

[142] Philip Wadler. "Linear Types can Change the World!" In: *Programming Concepts and Methods*. Vol. 3. 4. Citeseer. 1990, p. 5. URL: https://homepages.inf.ed.ac.uk/wadler/papers/linear/linear.dvi.

[143] Philip Wadler. "Is There a Use for Linear Logic?" In: *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. PEPM '91. New Haven, Connecticut, USA: Association for Computing Machinery, 1991, pp. 255–273. ISBN: 0897914333. DOI: 10.1145/115865.115894.

[144] Philip Wadler. "There's no substitute for linear logic." In: *8th International Workshop on the Mathematical Foundations of Programming Semantics*. 1992.

[145] Philip Wadler. "A Syntax for Linear Logic." In: *Mathematical Foundations of Programming Semantics, 9th International Conference, New Orleans, LA, USA, April 7-10, 1993, Proceedings*. 1993, pp. 513–529. DOI: 10.1007/3-540-58027-1_24.

[146] Philip Wadler. "A taste of linear logic." In: *Mathematical Foundations of Computer Science 1993*. Ed. by Andrzej M. Borzyszkowski and Stefan Sokołowski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 185–210. ISBN: 978-3-540-47927-7.

[147] Philip Wadler. "Propositions as sessions." In: *Journal of Functional Programming* 24.2-3 (2014), pp. 384–418. DOI: 10.1017/S095679681400001X.

[148] David Walker. "Substructural Type Systems." In: *Advanced Topics in Types and Programming Languages* (2005), pp. 3–44.

[149] David Walker, Karl Crary, and Greg Morrisett. "Typed Memory Management via Static Capabilities." In: *ACM Trans. Program. Lang. Syst.* 22.4 (July 2000), pp. 701–771. ISSN: 0164-0925. DOI: 10.1145/363911.363923.

[150] Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. "Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs." In: *Proc. ACM Program. Lang.* 8.POPL (Jan. 2024). DOI: 10.1145/3632856.

[151] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. *Oxide: The Essence of Rust*. 2021. arXiv: 1903.00982 [cs.PL].

[152] James Wood and Robert Atkey. "A Linear Algebra Approach to Linear Metatheory." In: *Proceedings Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity&TLLA@IJCAR-FSCD 2020, Online, 29-30 June 2020*. Ed. by Ugo Dal Lago and Valeria de Paiva. Vol. 353. EPTCS. 2020, pp. 195–212. DOI: 10.4204/EPTCS.353.10.

[153] James Wood and Robert Atkey. "A Framework for Substructural Type Systems." In: *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*. Ed. by Ilya Sergey. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 376–402. DOI: 10.1007/978-3-030-99336-8_14.

[154] Nobuko Yoshida and Vasco T Vasconcelos. "Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication." In: *Electronic Notes in Theoretical Computer Science* 171.4 (2007), pp. 73–93. DOI: 10.1016/j.entcs.2007.02.056.

[155] Uma Zalakain and Ornela Dardha. "π with Leftovers: A Mechanisation in Agda." In: *Formal Techniques for Distributed Objects, Components, and Systems*. Ed. by Kirstin Peters and Tim A. C. Willemse. Cham: Springer International Publishing, 2021, pp. 157–174. ISBN: 978-3-030-78089-0. DOI: 10.1007/978-3-030-78089-0_9.

[156] Dengping Zhu and Hongwei Xi. "Safe Programming with Pointers Through Stateful Views." In: *Practical Aspects of Declarative Languages*. Ed. by Manuel V. Hermenegildo and Daniel Cabeza. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 83–97. ISBN: 978-3-540-30557-6. DOI: 10.1007/978-3-540-30557-6_8.

APPENDICES

A APPENDIX FOR CHAPTER 3

A.1 COLLECTED RULES

A.1.1 Typing

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

$$\frac{\Gamma \vdash t : A}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C}\ \otimes_E$$

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : B}\ 1_E$$

A.1.2 Reduction rules for heap semantics

$$\frac{}{(\lambda x.t_2)\, t_1 \rightsquigarrow t_2[t_1/x]}\ \beta_{\mathrm{L}} \qquad \frac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_2}\ \textsc{congApp}_{\mathrm{L}}$$

$$\frac{}{\mathbf{let}\,[x] = [t_1]\,\mathbf{in}\,t_2 \rightsquigarrow t_2[t_1/x]}\ \Box\beta_{\mathrm{L}} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \rightsquigarrow \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2}\ \textsc{congLet}_{\mathrm{L}}$$

$$\frac{}{\mathbf{let}\,(x,y) = t_1 * t_2\,\mathbf{in}\,t_3 \rightsquigarrow t_3[t_2/y][t_1/x]}\ \otimes\beta_{\mathrm{L}}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\,(x,y) = t_1\,\mathbf{in}\,t_2 \rightsquigarrow \mathbf{let}\,(x,y) = t_1'\,\mathbf{in}\,t_2}\ \textsc{congProd}_{\mathrm{L}}$$

$$\frac{}{\mathbf{let}\,() = ()\,\mathbf{in}\,t_2 \rightsquigarrow t_2}\ \textsc{Unit}\beta_{\mathrm{L}} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 \rightsquigarrow \mathbf{let}\,() = t_1'\,\mathbf{in}\,t_2}\ \textsc{congUnit}_{\mathrm{L}}$$

A.2 SUBSTITUTION PROOFS

Lemma 1 (Admissibility of weakening)**.**

$$\Gamma \vdash t : A \implies (0 \cdot \Gamma'), \Gamma \vdash t : A$$

Proof. By induction on the structure of typing and since all 'leaf' nodes of a typing derivation permit weakening (necessarily the var rule here). □

Lemma 2 (Linear substitution)**.** *Given* $\Gamma' \vdash t' : S$ *and* $\Gamma, y : S \vdash t : T$ *then* $\Gamma' + \Gamma \vdash t[t'/y] : T$.

Proof. By induction on the typing derivation of t_2.

- (var)

  $$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var}$$

  Here, $y = t = x$ necessarily because this is the only linear variable in the scope ($y \notin \mathsf{dom}([\Gamma])$), and thus $S = T$. We need to show that $\Gamma' + [\Gamma] \vdash y[t'/y] : S$.

  Then since $y[t'/y] = t'$, and we have $\Gamma' \vdash t' : S$, we can apply the lemma of admissibility of weakening (Lemma 1) to derive the goal $\Gamma' + [\Gamma] \vdash y[t'/y] : S$.

- (abs)

  $$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

By induction, we have $\Gamma' + \Gamma, x : A \vdash t[t'/y] : B$. We need to show that $\Gamma' + \Gamma \vdash (\lambda x.t)[t'/y] : A \multimap B$.

Applying the ABS rule to the induction hypothesis we get $\Gamma' + \Gamma \vdash \lambda x.t[t'/y] : A \multimap B$.

Then by the definition of substitution we have $\Gamma' + \Gamma \vdash (\lambda x.t)[t'/y] : A \multimap B$ as required.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

We need to show that $\Gamma' + \Gamma_1 + \Gamma_2 \vdash (t_1\, t_2)[t'/y] : B$.

By the definition of context addition and the fact that y is linear, we either have $y \in \Gamma_1$ or $y \in \Gamma_2$ but not both. So there are two cases to consider.

  - If $y \in \Gamma_1$, then by induction on the first premise, we have $\Gamma' + \Gamma_1 \vdash t_1[t'/y] : A \multimap B$. Applying the APP rule to this and the second premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash t_1[t'/y]\, t_2 : B$.

    Since $y \notin \Gamma_2$, $t_1[t'/y]\, t_2[t'/y]$ (which we need) is equivalent to $t_1[t'/y]\, t_2$ (which we have).
  - If $y \in \Gamma_2$, then by induction on the second premise, we have $\Gamma' + \Gamma_2 \vdash t_2[t'/y] : A$. Applying the APP rule to this and the first premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash t_1\, t_2[t'/y] : B$.

    Since $y \notin \Gamma_1$, $t_1[t'/y]\, t_2[t'/y]$ (which we need) is equivalent to $t_1\, t_2[t'/y]$ (which we have).

- (pr)

$$\frac{\Gamma \vdash t : A}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

This case is trivial, since by the premise $r \cdot \Gamma$ contains only graded assumptions, so cannot contain any linear variables.

- (elim)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

We need to show that $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2[t'/y] : B$.

By the definition of context addition and the fact that y is linear, we either have $y \in \Gamma_1$ or $y \in \Gamma_2$ but not both. So there are two cases to consider:

- If $y \in \Gamma_1$, then by induction on the first premise, we have $\Gamma' + \Gamma_1 \vdash t_1[t'/y] : \Box_r A$. Applying the ELIM rule to this and the second premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1[t'/y]\,\mathbf{in}\,t_2 : B$.

  Since $y \notin \Gamma_2$, $\mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2[t'/y]$ (which we need) is equivalent to $\mathbf{let}\,[x] = t_1[t'/y]\,\mathbf{in}\,t_2$ (which we have).

- If $y \in \Gamma_2$, then by induction on the second premise, we have $\Gamma' + \Gamma_2, x : \Box_r A \vdash t_2[t'/y] : B$. Applying the ELIM rule to this and the first premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2[t'/y] : B$.

  Since $y \notin \Gamma_1$, $\mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2[t'/y]$ (which we need) is equivalent to $\mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2[t'/y]$ (which we have).

- (approx)

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx}$$

  We know that $x \neq y$ as x here is graded whilst y is a linear assumption, thus we assume $\Gamma = \Gamma'', y : S$.

  By induction, we have $\Gamma''' + (\Gamma'' + \Gamma'), x : \Box_r A \vdash t[t'/y] : B$. We need to show that $(\Gamma''' + \Gamma'' + \Gamma'), x : \Box_r A \vdash t[t'/y] : B$.

  Applying the APPROX rule to the induction hypothesis we get $\Gamma''' + (\Gamma'' + \Gamma'), x : \Box_r A \vdash t[t'/y] : B$ whose context, if it is defined, is then equal to $(\Gamma''' + \Gamma'' + \Gamma'), x : \Box_r A$ and we are done.

- (der)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

  We know that $x \neq y$ as x here is graded whilst y is a linear assumption, thus we assume $\Gamma = \Gamma'', y : S$

  By induction, we have $\Gamma' + (\Gamma'', x : A) \vdash t[t'/y] : B$. We need to show that $(\Gamma' + \Gamma''), x : \Box_r A \vdash t[t'/y] : B$.

  Applying the DER rule to the induction hypothesis we get $\Gamma' + \Gamma'', x : \Box_r A \vdash t[t'/y] : B$, whose context, if it is defined, is then equal to $(\Gamma' + \Gamma''), x : \Box_r A$ and we are done.

- (pairIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

  We need to show that $\Gamma' + \Gamma_1 + \Gamma_2 \vdash (t_1, t_2)[t'/y] : A \otimes B$.

By the definition of context addition and the fact that y is linear, we either have $y \in \Gamma_1$ or $y \in \Gamma_2$ but not both. So there are two cases to consider:

- If $y \in \Gamma_1$, then by induction on the first premise, we have $\Gamma' + \Gamma_1 \vdash t_1[t'/y] : A$. Applying the $\otimes_I$ rule to this and the second premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash (t_1[t'/y], t_2) : A \otimes B$.

  Since $y \notin \Gamma_2$, $(t_1, t_2)[t'/y]$ (which we need) is equivalent to $(t_1[t'/y], t_2)$ (which we have).
- If $y \in \Gamma_2$, then by induction on the second premise, we have $\Gamma' + \Gamma_2, x : [A] \vdash t_2[t'/y] : B$. Applying the $\otimes_I$ rule to this and the first premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash (t_1, t_2[t'/y]) : A \otimes B$.

  Since $y \notin \Gamma_1$, $(t_1, t_2)[t'/y]$ (which we need) is equivalent to $(t_1, t_2[t'/y])$ (which we have).

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C}\ \otimes_E$$

We need to show that $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2[t'/y] : C$.

By the definition of context addition and the fact that y is linear, we either have $y \in \Gamma_1$ or $y \in \Gamma_2$ but not both. So there are two cases to consider:

- If $y \in \Gamma_1$, then by induction on the first premise, we have $\Gamma' + \Gamma_1 \vdash t_1[t'/y] : A \otimes B$. Applying the $\otimes_E$ rule to this and the second premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1[t'/y]\,\mathbf{in}\,t_2 : C$.

  Since $y \notin \Gamma_2$, $\mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2[t'/y]$ (which we need) is equivalent to $\mathbf{let}\,(x, y) = t_1[t'/y]\,\mathbf{in}\,t_2$ (which we have).
- If $y \in \Gamma_2$, then by induction on the second premise, we have $\Gamma' + \Gamma_2, x : A, y : B \vdash t_2[t'/y] : C$. Applying the $\otimes_E$ rule to this and the first premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2[t'/y] : C$.

  Since $y \notin \Gamma_1$, $\mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2[t'/y]$ (which we need) is equivalent to $\mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2[t'/y]$ (which we have).

- (unitIntro)

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I$$

Holds trivially since there is no linear assumption in the context, so the premise is false.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ () = t_1\ \mathbf{in}\ t_2 : B}\ 1_E$$

We need to show that $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ () = t_1\ \mathbf{in}\ t_2[t'/y] : B$.

By the definition of context addition and the fact that y is linear, we either have $y \in \Gamma_1$ or $y \in \Gamma_2$ but not both. So there are two cases to consider:

- If $y \in \Gamma_1$, then by induction on the first premise, we have $\Gamma' + \Gamma_1 \vdash t_1[t'/y] : \mathsf{unit}$. Applying the 1_E rule to this and the second premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ () = t_1[t'/y]\ \mathbf{in}\ t_2 : B$.

  Since $y \notin \Gamma_2$, $\mathbf{let}\ () = t_1\ \mathbf{in}\ t_2[t'/y]$ (which we need) is equivalent to $\mathbf{let}\ () = t_1[t'/y]\ \mathbf{in}\ t_2$ (which we have).

- If $y \in \Gamma_2$, then by induction on the second premise, we have $\Gamma' + \Gamma_2 \vdash t_2[t'/y] : B$. Applying the 1_E rule to this and the first premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ () = t_1\ \mathbf{in}\ t_2[t'/y] : B$.

  Since $y \notin \Gamma_1$, $\mathbf{let}\ () = t_1\ \mathbf{in}\ t_2[t'/y]$ (which we need) is equivalent to $\mathbf{let}\ () = t_1\ \mathbf{in}\ t_2[t'/y]$ (which we have).

□

Lemma 3 (Graded substitution)**.** *Given* $[\Gamma'] \vdash t' : S$ *and* $\Gamma, y : [S]_r \vdash t : T$ *(where* $[\Gamma']$ *represents a context* Γ' *containing only graded assumptions) then* $\Gamma + r \cdot \Gamma' \vdash t[t'/y] : T$.

Proof. By induction on the typing derivation of t_2.

- (var)

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var}$$

Two cases:

1. $x = y$. Not possible as the typing doesn't match; y is a graded variable.
2. $x \neq y$ therefore $x \in \mathsf{dom}(\Gamma)$, i.e. the typing is:

$$(0 \cdot \Gamma''), y : [S]_0, x : A \vdash x : A\ \textsc{var}$$

Thus since $x[t'/y] = x$ we get the goal here by re-deriving the variable typing as follows (since it include arbitrary weakening in its definition):

$$0 \cdot (\Gamma + \Gamma'), x : A \vdash x : A\ \textsc{var}$$

satisfying the goal here.

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

By induction, we have $(r \cdot \Gamma') + \Gamma, x : A \vdash t[t'/y] : B$. We need to show that $(r \cdot \Gamma') + \Gamma \vdash \lambda x.t[t'/y] : A \multimap B$.

Applying the ABS rule to the induction hypothesis we get $(r \cdot \Gamma') + \Gamma \vdash \lambda x.t[t'/y] : A \multimap B$.

Then by the definition of substitution we have $(r \cdot \Gamma') + \Gamma \vdash \lambda x.t[t'/y] : A \multimap B$.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

We need to show that $(r \cdot \Gamma') + \Gamma_1 + \Gamma_2 \vdash t_1[t'/y]\, t_2 : B$.

By the definition of context addition we have either $y \in \Gamma_1$ or $y \in \Gamma_2$ or both, so there are three cases to consider.

1. If $y \in \Gamma_1$, then by induction on the first premise, we have $(r \cdot \Gamma') + \Gamma_1 \vdash t_1[t'/y] : A \multimap B$. Applying the APP rule to this and the second premise gives $(r \cdot \Gamma') + \Gamma_1 + \Gamma_2 \vdash t_1[t'/y]\, t_2 : B$.

   Since $y \notin \Gamma_2$, $t_1[t'/y]\, t_2[t'/y]$ (which we need) is equivalent to $t_1[t'/y]\, t_2$ (which we have).

2. If $y \in \Gamma_2$, then by induction on the second premise, we have $(r \cdot \Gamma') + \Gamma_2 \vdash t_2[t'/y] : A$. Applying the APP rule to this and the first premise gives $(r \cdot \Gamma') + \Gamma_1 + \Gamma_2 \vdash t_1\, (t_2[t'/y]) : B$.

   Since $y \notin \Gamma_1$, $t_1[t'/y]\, t_2[t'/y]$ (which we need) is equivalent to $t_1\, t_2[t'/y]$ (which we have).

3. If $y \in \Gamma_2$ and $y \in \Gamma_1$ then by induction on both premises, we have $(r \cdot \Gamma') + \Gamma_2 \vdash t_2[t'/y] : A$ and $(r \cdot \Gamma') + \Gamma_1 \vdash t_1[t'/y] : A \multimap B$. Applying the APP rule to this and the first premise gives

   $(r \cdot \Gamma') + (r \cdot \Gamma') + \Gamma_1 + \Gamma_2 \vdash (t_1[t'/y])\, (t_2[t'/y]) : B$.

   By the definition of context addition $(r \cdot \Gamma') + (r \cdot \Gamma') = (r \cdot \Gamma')$ so we have the goal here.

- (pr)

$$\frac{\Gamma \vdash t : A}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

Let $\Gamma = \Gamma_1, y : S$ thus we can induct on the premise yielding $(r \cdot \Gamma') + (r \cdot \Gamma_1) \vdash t[t'/y] : A$ upon which we can apply promotion again yielding:

$$\frac{(r \cdot \Gamma') + (r \cdot \Gamma_1) \vdash t[t'/y] : A}{(r \cdot \Gamma') + (r \cdot \Gamma_1) \vdash [(t[t'/y])] : \Box_r A}\ \textsc{pr}$$

which satisfies the goal since syntactic substitution is defined $[t][t'/y] = [(t[t'/y])]$.

- (elim)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

We need to show that $(r \cdot \Gamma') + \Gamma_1 + \Gamma_2 \vdash (\mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2)[t'/y] : B$.

The reasoning is almost identical to that of the application case where (app) where we inductively apply the substitution and we rely on the property that $(r \cdot \Gamma') + (r \cdot \Gamma') = (r \cdot \Gamma')$ in the case that y appears in both subterms.

- (approx)

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx}$$

There are two cases to consider depending on the location of y:

1. $x \equiv y$ Therefore the goal is $(r \cdot \Gamma'') + \Gamma + \Gamma' \vdash t[t'/y] : B$ which follows by induction on the premise.
2. $x \not\equiv y$, i.e., $x \in \mathsf{dom}(\Gamma)$ i.e., $\Gamma = \Gamma'', y : \Box_r S$ then we induct on the premise to get $((r \cdot \Gamma''') + \Gamma'' + \Gamma'), x : \Box_r A \vdash t[t'/y] : B$ on which we apply approximation again to get the goal.

- (der)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

There are two cases to consider depending on the location of y:

1. $x \equiv y$ Therefore the goal is $(r \cdot \Gamma') + \Gamma \vdash t[t'/y] : B$ which follows by applying Lemma 2 on the premise.
2. $x \not\equiv y$, i.e., $x \in \mathsf{dom}(\Gamma)$ i.e., $\Gamma = \Gamma'', y : \Box_r S$ then we induct on the premise to get $((r \cdot \Gamma') + \Gamma''), x : A \vdash t[t'/y] : B$ on which we apply dereliction again to get the goal.

- (pairIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B} \otimes_I$$

  As above for (elim) and (app), same proof pattern.

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C} \otimes_E$$

  Same dual induction as for (app), (elim), (pairIntro).

- (unitIntro)

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I$$

  Let $(r \cdot \Gamma) = (r \cdot \Gamma_1), y : \Box_r S, (r \cdot \Gamma_2)$ then since $()[t'/y] = ()$ then we can get the goal $\Gamma' + (r \cdot \Gamma) \vdash ()[t'/y] : \mathsf{unit}$ simply by reapplying this typing rule with the $\Gamma' + (r \cdot \Gamma)$ context.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : B}\ 1_E$$

  Same proof pattern as in (app), (pairIntro) etc.

□

A.3 TYPE SAFETY

A.3.1 Value lemma

Lemma 4 (Value lemma)**.** *Given $\Gamma \vdash v : A$ then depending on the type, the shape of v can be inferred:*

- $A = A' \to B$ *then* $v = \lambda x.t$
- $A = \Box_r A'$ *then* $v = [t]$
- $A = A' \otimes B'$ *then* $v = (t_1, t_2)$
- $A = 1$ *then* $v = ()$

Proof. Recall that the sub-grammar of value terms is defined as follows:

$$v ::= \lambda x.t \mid [t] \mid (t_1, t_2) \mid () \qquad \text{(value terms sub-grammar)}$$

We proceed by case analysis on the type A to match the structure of the lemma. In each case we must consider what possible values can be assigned the type A and by which rules.

In all cases, there exists additional derivations based on dereliction and approximation, e.g., for the case where $A = A' \to B$:

$$\frac{\Gamma, x : A'' \vdash t : A' \to B}{\Gamma, x : [A'']_1 \vdash t : A' \to B}\ \text{DER}$$

$$\frac{\Gamma, y : [A'']_r, \Gamma' \vdash t : A' \to B \quad r \sqsubseteq s}{\Gamma, y : [A'']_s, \Gamma' \vdash t : A' \to B}\ \text{APPROX}$$

In all of these cases we can apply induction on the premise to get the result since the term is preserved between the premise and the conclusion.

We elide handling this separately each time in the cases that follow as the reasoning through dereliction and approximation is the same each time.

- $A = A' \to B$ then the only possible non-dereliction/non-approximation typing is:
$$\frac{\Gamma, x : A' \vdash t : B}{\Gamma \vdash \lambda x.t : A' \multimap B}\ \text{ABS}$$
thus $v = \lambda x.t$ as in the lemma statement.

- $A = \Box_r A'$ then the only possible non-dereliction/non-approximation typing is:
$$\frac{\Gamma \vdash v' : A'}{r \cdot \Gamma \vdash [v'] : \Box_r A'}\ \text{PR}$$
thus $v = [v']$ as in the lemma statement.

- $A = A' \otimes B'$ then the only possible non-dereliction/non-approximation typing is:
$$\frac{\Gamma_1 \vdash v_1 : A' \quad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : A' \otimes B}\ \otimes_I$$
thus $v = (v_1, v_2)$ as in the lemma statement.

- $A = 1$ then the only possible non-dereliction/non-approximation typing is:
$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I$$
thus $v = ()$ as in the lemma statement.

□

A.3.2 Progress proof

Theorem 1 (Progress)**.** *Values in our model are given above.*

Given $\Gamma \vdash t : A$, then t is either some value v, or there exists a term t' such that $t \rightsquigarrow t'$.

Proof. By induction on typing.

- (var)

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var}$$

  Trivial since the variable x cannot be reduced and so is necessarily a value.

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

  Trivial since $(\lambda x.t)$ is already a value.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

  By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'.

  In the first case, by the value lemma $t_1 = \lambda x.t_1'$. Here, there are two possibilities: either $t_2 = v$ for some value v or $t_2 \rightsquigarrow t_2'$ for some t_2'.

  In the first case, we can reduce by

$$\frac{}{(\lambda x.t_2)\, t_1 \rightsquigarrow t_2[t_1/x]}\ \beta_{\mathrm{L}}$$

  .

  In the second case, we can reduce by

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_2}\ \textsc{congApp}_{\mathrm{L}}$$

  .

- (pr)

$$\frac{\Gamma \vdash t : A}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

  Trivial since $[t]$ is already a value.

- (elim)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'.

In the first case, by the value lemma $t_1 = [v']$ for some value v'. This refines the typing as follows:

$$\dfrac{\dfrac{\varnothing \vdash v' : A}{\varnothing \vdash [v'] : \Box_r A}\ \textsc{pr} \qquad x : [A]_r \vdash t_2 : B}{\varnothing \vdash \mathbf{let}\,[x] = [v']\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

Therefore, we can reduce by

$$\dfrac{}{\mathbf{let}\,[x] = [t_1]\,\mathbf{in}\,t_2 \rightsquigarrow t_2[t_1/x]}\ \Box\beta_{\mathrm{L}}$$

In the second case, by induction on the premise we can reduce by

$$\dfrac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \rightsquigarrow \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2}\ \textsc{congLet}_{\mathrm{L}}$$

- (approx)

$$\dfrac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx}$$

Goal achieved immediately by induction on the premise.

- (der)

$$\dfrac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

Goal achieved immediately by induction on the premise.

- (pairIntro)

$$\dfrac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

Trivial since (t_1, t_2) is already a value.

- (pairElim)

$$\dfrac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C}\ \otimes_E$$

By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'.

In the first case, by the value lemma $t_1 = (v_1, v_2)$ for some pair of values (v_1, v_2). This refines the typing as follows:

$$\dfrac{\dfrac{\varnothing \vdash v_1 : A \quad \varnothing \vdash v_2 : A'}{\varnothing \vdash (v_1, v_2) : A \otimes A'}\ \otimes_I \qquad x : A, y : A' \vdash t_2 : B}{\varnothing \vdash \mathbf{let}\,(x, y) = (v_1, v_2)\,\mathbf{in}\,t_2 : B}\ \otimes_E$$

Therefore, we can reduce by

$$\frac{}{\textbf{let}\,(x,y) = t_1 * t_2\,\textbf{in}\,t_3 \rightsquigarrow t_3[t_2/y][t_1/x]}\ \otimes\beta_{\mathsf{L}}$$

In the second case, by induction on the premise we can reduce by

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let}\,(x,y) = t_1\,\textbf{in}\,t_2 \rightsquigarrow \textbf{let}\,(x,y) = t_1'\,\textbf{in}\,t_2}\ \textsc{congProd}_{\mathsf{L}}$$

- (unitIntro)

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I$$

Trivial since $()$ is already a value.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\,() = t_1\,\textbf{in}\,t_2 : B}\ 1_E$$

By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'.

In the first case, by the value lemma $t_1 = ()$ and so we can reduce by

$$\frac{}{\textbf{let}\,() = ()\,\textbf{in}\,t_2 \rightsquigarrow t_2}\ \textsc{Unit}\beta_{\mathsf{L}}$$

In the second case, by induction on the premise we can reduce by

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let}\,() = t_1\,\textbf{in}\,t_2 \rightsquigarrow \textbf{let}\,() = t_1'\,\textbf{in}\,t_2}\ \textsc{congUnit}_{\mathsf{L}}$$

□

A.3.3 Preservation proof

Theorem 2 (Type preservation)**.** *Given $\Gamma \vdash t : A$ and a single-step reduction $t \rightsquigarrow t'$, then we can construct the typing derivation $\Gamma \vdash t' : A$.*

Proof. By induction on the structure of typing and reductions.

- (var)

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var}$$

Trivial since the variable x cannot be reduced so the second requirement of the theorem cannot be fulfilled.

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

Trivial since $(\lambda x.t)$ is a value so the second requirement of the theorem cannot be fulfilled.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

Here, the reduction is necessarily:

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_2}\ \textsc{congApp}_{\mathrm{L}}$$

By induction we have $\Gamma_1 \vdash t_1' : A \multimap B$ from which the required typing follows immediately by application.

- (pr)

$$\frac{\Gamma \vdash t : A}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

Trivial since $[t]$ is a value so the second requirement of the theorem cannot be fulfilled.

- (elim)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

Here, the reduction is necessarily:

$$\frac{}{\mathbf{let}\,[x] = [t_1]\,\mathbf{in}\,t_2 \rightsquigarrow t_2[t_1/x]}\ \Box\beta_{\mathrm{L}}$$

The goal follows from the second premise by applying the substitution lemma.

- (approx)

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx}$$

Straightforward induction since the term t is unchanged.

- (der)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

Straightforward induction since the term t is unchanged.

- (pairIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

Trivial since (t_1, t_2) is a value so the second requirement of the theorem cannot be fulfilled.

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C}\ \otimes_E$$

  Here, the reduction is necessarily:

$$\frac{}{\mathbf{let}\,(x, y) = t_1 * t_2\,\mathbf{in}\,t_3 \rightsquigarrow t_3[t_2/y][t_1/x]}\ \otimes\beta_{\mathsf{L}}$$

  The goal follows from the second premise by applying the substitution lemma (twice).

- (unitIntro)

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I$$

  Trivial since $()$ is a value so the second requirement of the theorem cannot be fulfilled.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : B}\ 1_E$$

  Here, the reduction is necessarily:

$$\frac{}{\mathbf{let}\,() = ()\,\mathbf{in}\,t_2 \rightsquigarrow t_2}\ \textsc{Unit}\beta_{\mathsf{L}}$$

  The goal follows directly from the second premise.

□

b APPENDIX FOR CHAPTER 4

B.1 COLLECTED RULES

B.1.1 Typing

Definition 13 (All non-linear assumptions). *A context Γ is denoted as containing only non-linear assumptions by writing $[\Gamma]$ in proofs, where $[\emptyset]$ and $[\Gamma] \implies [\Gamma], x : [A]$.*

Definition 14 (Context addition). *The partial operation $+$ on contexts is the union of two contexts as long as they are disjoint in their linear assumptions and any variables occurring in both contexts are both non-linear assumptions, i.e.*

$$\Gamma_1 + \Gamma_2 = \Gamma_1 \cup \Gamma_2 \quad \textit{iff} \quad \forall x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) \implies \exists A.\Gamma_1(x) = \Gamma_2(x) = [A]$$

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs} \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I \qquad \frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,(x, y) = t_1\,\mathsf{in}\,t_2 : C}\ \otimes_E$$

$$\frac{}{[\Gamma] \vdash \mathsf{unit} : 1}\ 1_I \qquad \frac{\Gamma_1 \vdash t_1 : 1 \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let\,unit} = t_1\,\mathsf{in}\,t_2 : B}\ 1_E$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A] \vdash t : B}\ \textsc{der} \qquad \frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash\, !t : !A}\ !_I \qquad \frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,!x = t_1\,\mathsf{in}\,t_2 : B}\ !_E$$

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathsf{share}\,t : !A}\ \textsc{share} \qquad \frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\,t_1\,\mathsf{as}\,x\,\mathsf{in}\,t_2 : !B}\ \textsc{clone} \qquad \frac{\emptyset \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

PRIMITIVES

$$\frac{}{[\Gamma] \vdash \mathsf{newArray} : \mathbb{N} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{new}$$

$$\frac{}{[\Gamma] \vdash \mathsf{readArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{read}$$

$$\frac{}{[\Gamma] \vdash \mathsf{writeArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{write}$$

$$\frac{}{[\Gamma] \vdash \mathsf{deleteArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap 1}\ \mathsf{del}$$

B.1.2 Reduction rules for heap semantics

RUNTIME TYPING In order to prove type preservation and progress, conservation, and uniqueness we need to weave in typing into the heap model so we also need a way of typing runtime array primitives (which extend the term syntax with an atom a). Contexts are extend to include their typing as:

$$\Gamma ::= \ldots \mid \Gamma, a : \mathsf{Array}\ A$$

and with typing rules:

$$\frac{}{[\Gamma], a : \mathsf{Array}\ A \vdash a : \mathsf{Array}\ A}\ \mathsf{arr} \qquad \frac{}{[\Gamma], a : \mathsf{Array}\ A \vdash *a : *(\mathsf{Array}\ A)}\ *\mathsf{arr}$$

Context operations are therefore also extended:

Definition b.1.1 (All non-linear assumptions (runtime typing))**.** *A context Γ is denoted as containing only non-linear assumptions by writing $[\Gamma]$ in the typing rules, which is defined inductively as*

$$\frac{}{[\varnothing]} \qquad \frac{[\Gamma]}{[\Gamma], x : [A]} \qquad \frac{[\Gamma]}{[\Gamma], a : \mathsf{Array}\ A}$$

ADDITIONAL DEFINITIONS

Definition b.1.2 (Reference counts in a heap)**.** *Given a heap H then $[H]_r$ converts all array references to have reference count r, defined:*

$$\begin{aligned} [\varnothing]_r &= \varnothing \\ [(H, a \mapsto_{r'} \mathbf{arr})]_r &= ([H]_r), a \mapsto_r \mathbf{arr} \\ [(H, x \mapsto_{r'} t)]_r &= ([H]_r), x \mapsto_{r'} t \end{aligned}$$

Note that variable assignments in the heap are ignored (although in the heap semantics this operation is applied only to heaps with just array references).

This can also be thought of as a predicate on the image of the function.

Definition b.1.3 (Array references in a term). *By induction:*

$$
\begin{aligned}
&\mathsf{arrRefs}(x) && = \emptyset \\
&\mathsf{arrRefs}(\lambda x.t) && = \mathsf{arrRefs}(t) \\
&\mathsf{arrRefs}(t_1\, t_2) && = \mathsf{arrRefs}(t_1) \cup \mathsf{arrRefs}(t_2) \\
&\mathsf{arrRefs}(!t) && = \mathsf{arrRefs}(t) \\
&\mathsf{arrRefs}(\mathsf{let}\, !x = t_1\, \mathsf{in}\, t_2) && = \mathsf{arrRefs}(t_1) \cup \mathsf{arrRefs}(t_2) \\
&\mathsf{arrRefs}((t_1, t_2)) && = \mathsf{arrRefs}(t_1) \cup \mathsf{arrRefs}(t_2) \\
&\mathsf{arrRefs}(\mathsf{let}\, (x, y) = t_1\, \mathsf{in}\, t_2) && = \mathsf{arrRefs}(t_1) \cup \mathsf{arrRefs}(t_2) \\
&\mathsf{arrRefs}(\mathsf{share}\, t) && = \mathsf{arrRefs}(t) \\
&\mathsf{arrRefs}(\mathsf{clone}\, t_1\, \mathsf{as}\, x\, \mathsf{in}\, t_2) && = \mathsf{arrRefs}(t_1) \cup \mathsf{arrRefs}(t_2) \\
&\mathsf{arrRefs}(\mathsf{unit}) && = \emptyset \\
&\mathsf{arrRefs}(a) && = \{a\} \\
&\mathsf{arrRefs}(*t) && = \mathsf{arrRefs}(t) \\
&\mathsf{arrRefs}(\mathsf{newArray}) && = \emptyset \\
&\mathsf{arrRefs}(\mathsf{readArray}) && = \emptyset \\
&\mathsf{arrRefs}(\mathsf{writeArray}) && = \emptyset \\
&\mathsf{arrRefs}(\mathsf{deleteArray}) && = \emptyset
\end{aligned}
$$

Definition b.1.4 (Heap copy). *The heap copy* $\mathsf{copy}(H)$ *copies all array references (ignoring variable assignments in the heap; this function is only ever called on sub-heaps with just array references) into fresh array references along with a substitution (return as a pair). Defined:*

$$
\begin{aligned}
\mathsf{copy}(\emptyset) &= (\emptyset, \emptyset) \\
\mathsf{copy}(H, x \mapsto_r t) &= \mathsf{copy}(H) \\
\mathsf{copy}(H, a \mapsto_r \mathbf{arr}) &= ((H', a \mapsto_r \mathbf{arr}, a' \mapsto_1 \mathbf{arr}),\ \theta \cup \{a \mapsto a'\}) \\
&\qquad \textit{where}\ (H', \theta) = \mathsf{copy}(H) \quad \wedge \quad a' \,\#\, \mathsf{dom}(H)
\end{aligned}
$$

Definition 15 (Heap-context compatibility). *A heap H is compatible with a typing context Γ if H contains assignments for every variable in the context and the typing contexts of the terms in the heap are also compatible with the heap. The relation is defined inductively as:*

$$\frac{H \bowtie \Gamma}{H, a \mapsto_r \mathbf{arr} \bowtie \Gamma, a : \mathsf{Array}\ A}\ \textsc{Ref} \qquad \frac{}{\emptyset \bowtie \emptyset}\ \textsc{Empty}$$

$$\frac{H \bowtie (\Gamma_1 + \Gamma_2) \quad \Gamma_2 \vdash t : A \quad x \notin \mathsf{dom}(H)}{(H, x \mapsto_r (\Gamma_2 \vdash t : A)) \bowtie (\Gamma_1, x : A)}\ \textsc{Lin}$$

$$\frac{H \bowtie (\Gamma_1 + [\Gamma_2]) \quad [\Gamma_2] \vdash t : A \quad x \notin \mathsf{dom}(H)}{(H, x \mapsto_\omega ([\Gamma_2] \vdash t : A)) \bowtie (\Gamma_1, x : [A])}\ \omega$$

Definition 16 (Usage context extraction). *For a context Γ or heap H we can extract usage information denoted $\overline{\Gamma}$ or $\overline{H}$ defined as:*

$$\begin{aligned}
\overline{\emptyset} &= \emptyset \\
\overline{(\Gamma, x : [A])} &= \overline{\Gamma}, x : \omega \\
\overline{(\Gamma, a : A)} &= \overline{\Gamma} \\
\overline{(\Gamma, x : A)} &= \overline{\Gamma}, x : 1 \qquad \textit{(contexts)}
\end{aligned}$$

$$\begin{aligned}
\overline{\emptyset} &= \emptyset \\
\overline{(H, x \mapsto_r (\Gamma \vdash t : A))} &= \overline{H}, x : r \\
\overline{(H, a \mapsto_r t)} &= \overline{H} \qquad \textit{(heaps)}
\end{aligned}$$

Definition b.1.5 (Usage context approximation). *We say that a usage context Δ is approximated by another Δ' if $\Delta \sqsubseteq \Delta'$ which is defined as follows:*

$$\frac{}{\overline{\emptyset \sqsubseteq \emptyset}} \qquad \frac{\Delta \sqsubseteq \Delta' \quad r \leq r'}{(\Delta, x : r) \sqsubseteq (\Delta', x : r')}$$

where $1 \leq 1$ and $1 \leq \omega$ and $\omega \leq \omega$.

MULTI-REDUCTION RULES

$$\frac{}{H \vdash t \;\Rightarrow\; H \vdash t \mid \emptyset \mid \emptyset}\ \textsc{refl} \qquad \frac{\begin{array}{c} H \vdash t_1 \;\rightsquigarrow\; H' \vdash t_2 \mid \Gamma_1 \mid \Delta_1 \\ H' \vdash t_2 \;\Rightarrow\; H'' \vdash t_3 \mid \Gamma_2 \mid \Delta_2 \end{array}}{H \vdash t_1 \;\Rightarrow\; H'' \vdash t_3 \mid \Gamma_1, \Gamma_2 \mid \Delta_1 + \Delta_2}\ \textsc{ext}$$

SINGLE-REDUCTION CONGRUENCES

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma \mid \Delta}{H \vdash t_1\, t_2 \rightsquigarrow H' \vdash t_1'\, t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\text{APP}} \qquad \frac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash p\, t \rightsquigarrow H' \vdash p\, t' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\text{prim}}$$

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma \mid \Delta}{H \vdash \mathsf{let}\,(x,y) = t_1\,\mathsf{in}\,t_2 \rightsquigarrow H' \vdash \mathsf{let}\,(x,y) = t_1'\,\mathsf{in}\,t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\text{LET}\otimes}$$

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma \mid \Delta}{H \vdash \mathsf{let\,unit} = t_1\,\mathsf{in}\,t_2 \rightsquigarrow H' \vdash \mathsf{let\,unit} = t_1'\,\mathsf{in}\,t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\text{LETunit}}$$

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma \mid \Delta}{H \vdash \mathsf{let}\,!x = t_1\,\mathsf{in}\,t_2 \rightsquigarrow H' \vdash \mathsf{let}\,!x = t_1'\,\mathsf{in}\,t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\text{LET!}}$$

$$\frac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash \mathsf{share}\,t \rightsquigarrow H' \vdash \mathsf{share}\,t' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{share}}$$

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma \mid \Delta}{H \vdash \mathsf{clone}\,t_1\,\mathsf{as}\,x\,\mathsf{in}\,t_2 \rightsquigarrow H' \vdash \mathsf{clone}\,t_1'\,\mathsf{as}\,x\,\mathsf{in}\,t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{clone}}$$

$$\frac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash \mathsf{clone}\,!t\,\mathsf{as}\,x\,\mathsf{in}\,t_2 \rightsquigarrow H' \vdash \mathsf{clone}\,!t'\,\mathsf{as}\,x\,\mathsf{in}\,t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{clone!}}$$

$$\frac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash *t \rightsquigarrow H' \vdash *t' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{*}$$

SINGLE-REDUCTION β-RULES

$$\frac{}{H, x{\mapsto_\omega} t \vdash x \rightsquigarrow H, x{\mapsto_\omega} t \vdash t \mid \emptyset \mid x : 1}\ {\rightsquigarrow}_{\text{VAR}\omega} \qquad \frac{}{H, x{\mapsto_1} t \vdash x \rightsquigarrow H \vdash t \mid \emptyset \mid x : 1}\ {\rightsquigarrow}_{\text{VAR1}}$$

$$\frac{\Gamma \vdash t' : A}{H \vdash (\lambda x.t)\, t' \rightsquigarrow H, x{\mapsto_1}(\Gamma \vdash t' : A) \vdash t \mid x : A \mid \emptyset}\ {\rightsquigarrow}_{\beta}$$

$$\frac{\Gamma_1 \vdash t_1' : A \qquad \Gamma_2 \vdash t_1'' : B}{H \vdash \mathsf{let}\,(x,y) = (t_1', t_1'')\,\mathsf{in}\,t_3 \rightsquigarrow H, x{\mapsto_1}(\Gamma_1 \vdash t_1' : A), y{\mapsto_1}(\Gamma_2 \vdash t_1'' : B) \vdash t_3 \mid \emptyset, x : A, y : B \mid \emptyset}\ {\rightsquigarrow}_{\otimes\beta}$$

$$\frac{}{H \vdash \mathsf{let\,unit} = \mathsf{unit}\,\mathsf{in}\,t \rightsquigarrow H \vdash t \mid \emptyset \mid \emptyset}\ {\rightsquigarrow}_{\beta\mathsf{unit}}$$

$$\frac{[\Gamma] \vdash t_1 : A}{H \vdash \mathsf{let}\,!x = !t_1\,\mathsf{in}\,t_2 \rightsquigarrow H, x{\mapsto_\omega}([\Gamma] \vdash t_1 : A) \vdash t_2 \mid x : [A] \mid \emptyset}\ {\rightsquigarrow}_{!\beta}$$

$$\frac{\mathsf{dom}(H) \equiv \mathsf{arrRefs}(v)}{H, H' \vdash \mathsf{share}\,(*v) \rightsquigarrow ([H]_\omega), H' \vdash !v \mid \emptyset \mid \emptyset}\ {\rightsquigarrow}_{\beta\mathsf{share}}$$

$$\frac{\Gamma \vdash v : A \qquad \mathsf{dom}(H') \equiv \mathsf{arrRefs}(v) \qquad (H'', \theta) \equiv \mathsf{copy}(H')}{H, H' \vdash \mathsf{clone}\,!v\,\mathsf{as}\,x\,\mathsf{in}\,t_2 \rightsquigarrow H, H', H'', x{\mapsto_1}(\Gamma \vdash *(\theta(v)) : *A) \vdash t_2 \mid x : *A \mid \emptyset}\ {\rightsquigarrow}_{\mathsf{clone}\beta}$$

$$\frac{a \# H}{H \vdash \mathsf{newArray}\,n \rightsquigarrow H, a{\mapsto_1}\mathbf{arr} \vdash *a \mid \emptyset \mid \emptyset}\ {\rightsquigarrow}_{\mathsf{newArray}}$$

$$\frac{}{H, a{\mapsto_r}(\mathbf{arr}[i] = v) \vdash \mathsf{readArray}\,(*a)\,i \rightsquigarrow H, a{\mapsto_r}(\mathbf{arr}[i] = v) \vdash (v, *a) \mid \emptyset \mid \emptyset}\ {\rightsquigarrow}_{\mathsf{readArray}}$$

$$\frac{}{H, a{\mapsto_r}\mathbf{arr} \vdash \mathsf{writeArray}\,(*a)\,i\,v \rightsquigarrow H, a{\mapsto_r}(\mathbf{arr}[i] = v) \vdash *a \mid \emptyset \mid \emptyset}\ {\rightsquigarrow}_{\mathsf{writeArray}}$$

$$\frac{}{H, a{\mapsto_r}\mathbf{arr} \vdash \mathsf{deleteArray}\,(*a) \rightsquigarrow H \vdash \mathsf{unit} \mid \emptyset \mid \emptyset}\ {\rightsquigarrow}_{\mathsf{deleteArray}}$$

B.2 SUBSTITUTION AND WEAKENING PROOFS

Lemma 5 (Linear substitution). *Given* $\Gamma' \vdash t' : S$ *and* $\Gamma, s : S \vdash t : T$ *then* $\Gamma' + \Gamma \vdash [t'/s]t : T$.

Proof. By induction on the structure of $\Gamma, s : S \vdash t : T$.

- (var)

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \text{VAR}$$

Here, $s = t = x$ necessarily because this is the only linear variable in the scope ($s \notin \mathsf{dom}([\Gamma])$), and thus $S = T$. We need to show that $\Gamma' + [\Gamma] \vdash [t'/s]s : S$.

Then since $[t'/s]s = t'$, and we have $\Gamma' \vdash t' : S$, we can apply the lemma of admissibility of weakening (Lemma 7) to derive the goal $\Gamma' + [\Gamma] \vdash [t'/s]s : S$.

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

By induction, we have $\Gamma' + \Gamma, x : A \vdash [t'/s]t : B$. We need to show that $\Gamma' + \Gamma \vdash [t'/s](\lambda x.t) : A \multimap B$.

Applying the ABS rule to the induction hypothesis we get $\Gamma' + \Gamma \vdash \lambda x.[t'/s]t : A \multimap B$.

Then by the definition of substitution we have $\Gamma' + \Gamma \vdash [t'/s](\lambda x.t) : A \multimap B$ as required.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

We need to show that $\Gamma' + \Gamma_1 + \Gamma_2 \vdash [t'/s](t_1\, t_2) : B$.

By the definition of context addition and the fact that s is linear, we either have $s \in \Gamma_1$ or $s \in \Gamma_2$ but not both. So there are two cases to consider.

  - If $s \in \Gamma_1$, then by induction on the first premise, we have $\Gamma' + \Gamma_1 \vdash [t'/s]t_1 : A \multimap B$. Applying the APP rule to this and the second premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash [t'/s]t_1\, t_2 : B$.

    Since $s \notin \Gamma_2$, $[t'/s]t_1\, [t'/s]t_2$ (which we need) is equivalent to $[t'/s]t_1\, t_2$ (which we have).
  - If $s \in \Gamma_2$, then by induction on the second premise, we have $\Gamma' + \Gamma_2 \vdash [t'/s]t_2 : A$. Applying the APP rule to this and the first premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash t_1\, [t'/s]t_2 : B$.

    Since $s \notin \Gamma_1$, $[t'/s]t_1\, [t'/s]t_2$ (which we need) is equivalent to $t_1\, [t'/s]t_2$ (which we have).

- (dereliction)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A] \vdash t : B}\ \textsc{der}$$

We know that $x \neq s$ as x here is non-linear whilst s is a linear assumption, thus we assume $\Gamma = \Gamma'', s : S$

By induction, we have $\Gamma' + (\Gamma'', x : A) \vdash [t'/s]t : B$. We need to show that $(\Gamma' + \Gamma''), x : [A] \vdash [t'/s]t : B$.

Applying the DER rule to the induction hypothesis we get $\Gamma' + \Gamma'', x : [A] \vdash [t'/s]t : B$, whose context, if it is defined is then equal to $(\Gamma' + \Gamma''), x : [A]$ and we are done.

- (bangIntro)

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash !t : !A}\ !_I$$

This case is trivial, since by the premise Γ contains only non-linear assumptions, so cannot contain any linear variables.

- (bangElim)

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\, !x = t_1 \,\mathsf{in}\, t_2 : B}\ !_E$$

We need to show that $\Gamma' + \Gamma_1 + \Gamma_2 \vdash [t'/s]\mathsf{let}\, !x = t_1 \,\mathsf{in}\, t_2 : B$.

By the definition of context addition and the fact that s is linear, we either have $s \in \Gamma_1$ or $s \in \Gamma_2$ but not both. So there are two cases to consider:

  - If $s \in \Gamma_1$, then by induction on the first premise, we have $\Gamma' + \Gamma_1 \vdash [t'/s]t_1 : !A$. Applying the $!_E$ rule to this and the second premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathsf{let}\, !x = [t'/s]t_1 \,\mathsf{in}\, t_2 : B$.

    Since $s \notin \Gamma_2$, $[t'/s]\mathsf{let}\, !x = t_1 \,\mathsf{in}\, t_2$ (which we need) is equivalent to $\mathsf{let}\, !x = [t'/s]t_1 \,\mathsf{in}\, t_2$ (which we have).
  - If $s \in \Gamma_2$, then by induction on the second premise, we have $\Gamma' + \Gamma_2, x : [A] \vdash [t'/s]t_2 : B$. Applying the $!_E$ rule to this and the first premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathsf{let}\, !x = t_1 \,\mathsf{in}\, [t'/s]t_2 : B$.

    Since $s \notin \Gamma_1$, $[t'/s]\mathsf{let}\, !x = t_1 \,\mathsf{in}\, t_2$ (which we need) is equivalent to $\mathsf{let}\, !x = t_1 \,\mathsf{in}\, [t'/s]t_2$ (which we have).

- (pairIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

We need to show that $\Gamma' + \Gamma_1 + \Gamma_2 \vdash [t'/s](t_1, t_2) : A \otimes B$.

By the definition of context addition and the fact that s is linear, we either have $s \in \Gamma_1$ or $s \in \Gamma_2$ but not both. So there are two cases to consider:

  - If $s \in \Gamma_1$, then by induction on the first premise, we have $\Gamma' + \Gamma_1 \vdash [t'/s]t_1 : A$. Applying the $\otimes_I$ rule to this and the second premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash ([t'/s]t_1, t_2) : A \otimes B$.

    Since $s \notin \Gamma_2$, $[t'/s](t_1, t_2)$ (which we need) is equivalent to $([t'/s]t_1, t_2)$ (which we have).

– If $s \in \Gamma_2$, then by induction on the second premise, we have $\Gamma' + \Gamma_2, x : [A] \vdash [t'/s]t_2 : B$. Applying the $\otimes_I$ rule to this and the first premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash (t_1, [t'/s]t_2) : A \otimes B$.

Since $s \notin \Gamma_1$, $[t'/s](t_1, t_2)$ (which we need) is equivalent to $(t_1, [t'/s]t_2)$ (which we have).

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,(x, y) = t_1 \,\mathsf{in}\, t_2 : C}\ \otimes_E$$

We need to show that $\Gamma' + \Gamma_1 + \Gamma_2 \vdash [t'/s]\mathsf{let}\,(x, y) = t_1 \,\mathsf{in}\, t_2 : C$.

By the definition of context addition and the fact that s is linear, we either have $s \in \Gamma_1$ or $s \in \Gamma_2$ but not both. So there are two cases to consider:

– If $s \in \Gamma_1$, then by induction on the first premise, we have $\Gamma' + \Gamma_1 \vdash [t'/s]t_1 : A \otimes B$. Applying the $\otimes_E$ rule to this and the second premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,(x, y) = [t'/s]t_1 \,\mathsf{in}\, t_2 : C$.

Since $s \notin \Gamma_2$, $[t'/s]\mathsf{let}\,(x, y) = t_1 \,\mathsf{in}\, t_2$ (which we need) is equivalent to $\mathsf{let}\,(x, y) = [t'/s]t_1 \,\mathsf{in}\, t_2$ (which we have).

– If $s \in \Gamma_2$, then by induction on the second premise, we have $\Gamma' + \Gamma_2, x : A, y : B \vdash [t'/s]t_2 : C$. Applying the $\otimes_E$ rule to this and the first premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,(x, y) = t_1 \,\mathsf{in}\, [t'/s]t_2 : C$.

Since $s \notin \Gamma_1$, $[t'/s]\mathsf{let}\,(x, y) = t_1 \,\mathsf{in}\, t_2$ (which we need) is equivalent to $\mathsf{let}\,(x, y) = t_1 \,\mathsf{in}\, [t'/s]t_2$ (which we have).

- (unitIntro)

$$\frac{}{[\Gamma] \vdash \mathsf{unit} : 1}\ 1_I$$

Holds trivial since there is no linear assumption in the context, so the premise is false.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : 1 \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,\mathsf{unit} = t_1 \,\mathsf{in}\, t_2 : B}\ 1_E$$

We need to show that $\Gamma' + \Gamma_1 + \Gamma_2 \vdash [t'/s]\mathsf{let}\,\mathsf{unit} = t_1 \,\mathsf{in}\, t_2 : B$.

By the definition of context addition and the fact that s is linear, we either have $s \in \Gamma_1$ or $s \in \Gamma_2$ but not both. So there are two cases to consider:

- If $s \in \Gamma_1$, then by induction on the first premise, we have $\Gamma' + \Gamma_1 \vdash [t'/s]t_1 : 1$. Applying the 1_E rule to this and the second premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathsf{let\ unit}\ = [t'/s]t_1\ \mathsf{in}\ t_2 : B$.

  Since $s \notin \Gamma_2$, $[t'/s]\mathsf{let\ unit}\ = t_1\ \mathsf{in}\ t_2$ (which we need) is equivalent to $\mathsf{let\ unit}\ = [t'/s]t_1\ \mathsf{in}\ t_2$ (which we have).

- If $s \in \Gamma_2$, then by induction on the second premise, we have $\Gamma' + \Gamma_2 \vdash [t'/s]t_2 : B$. Applying the 1_E rule to this and the first premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathsf{let\ unit}\ = t_1\ \mathsf{in}\ [t'/s]t_2 : B$.

  Since $s \notin \Gamma_1$, $[t'/s]\mathsf{let\ unit}\ = t_1\ \mathsf{in}\ t_2$ (which we need) is equivalent to $\mathsf{let\ unit}\ = t_1\ \mathsf{in}\ [t'/s]t_2$ (which we have).

- (share)

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathsf{share}\, t : !A}\ \textsc{share}$$

  By induction, we have $\Gamma' + \Gamma \vdash [t'/s]t : *A$. We need to show that $\Gamma' + \Gamma \vdash [t'/s](\mathsf{share}\, t) : !A$.

  Applying the SHARE rule to the induction hypothesis we get $\Gamma' + \Gamma \vdash (\mathsf{share}\, [t'/s]t) : !A$.

  $[t'/s](\mathsf{share}\, t)$ is equivalent to $(\mathsf{share}\, [t'/s]t)$, which we have.

- (clone)

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\, t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \textsc{clone}$$

  We need to show that $\Gamma' + \Gamma_1 + \Gamma_2 \vdash [t'/s]\mathsf{clone}\, t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : B$.

  By the definition of context addition and the fact that s is linear, we either have $s \in \Gamma_1$ or $s \in \Gamma_2$ but not both. So there are two cases to consider:

  - If $s \in \Gamma_1$, then by induction on the first premise, we have $\Gamma' + \Gamma_1 \vdash [t'/s]t_1 : !A$. Applying the CLONE rule to this and the second premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\, [t'/s]t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B$.

    Since $s \notin \Gamma_2$, $[t'/s]\mathsf{clone}\, t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2$ (which we need) is equivalent to $\mathsf{clone}\, [t'/s]t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2$ (which we have).

  - If $s \in \Gamma_2$, then by induction on the second premise, we have $\Gamma' + \Gamma_2, x : *A \vdash [t'/s]t_2 : !B$. Applying the CLONE rule to this and the first premise gives $\Gamma' + \Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\, t_1\ \mathsf{as}\ x\ \mathsf{in}\ [t'/s]t_2 : !B$.

    Since $s \notin \Gamma_1$, $[t'/s]\mathsf{clone}\, t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2$ (which we need) is equivalent to $\mathsf{clone}\, t_1\ \mathsf{as}\ x\ \mathsf{in}\ [t'/s]t_2$ (which we have).

- (nec)

$$\frac{\varnothing \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

Holds trivially since the typing here contains no linear assumptions and thus its context does not match the form of the lemma with linear assumption $s : S$.

- (primitives) The primitive operations have typing:

$$\frac{}{[\Gamma] \vdash \mathsf{newArray} : \mathbb{N} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{new}$$

$$\frac{}{[\Gamma] \vdash \mathsf{readArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{read}$$

$$\frac{}{[\Gamma] \vdash \mathsf{writeArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{write}$$

$$\frac{}{[\Gamma] \vdash \mathsf{deleteArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap 1}\ \mathsf{del}$$

The lemma holds trivially since the typing here contains no linear assumptions and thus its context does not match the form of the lemma with linear assumption $s : S$.

□

Lemma 6 (Non-linear substitution)**.** *Given* $[\Gamma'] \vdash t' : S$ *and* $\Gamma, s : [S] \vdash t : T$ *then* $[\Gamma'] + \Gamma \vdash [t'/s]t : T$.

Proof. By induction on the structure of $\Gamma, s : S \vdash t : T$.

- (var)

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \textsc{var}$$

Two cases:

1. $x = s$. Not possible as the typing doesn't match; s is a non-linear variable.
2. $x \neq s$ therefore $x \in \mathsf{dom}(\Gamma)$, i.e. the typing is:

$$[\Gamma'', s : S], x : A \vdash x : A\ \textsc{var}$$

Thus since $[t'/s]x = x$ we get the goal here by re-deriving the variable typing as follows (since it include arbitrary weakening in its definition):

$$[\Gamma + \Gamma'], x : A \vdash x : A\ \textsc{var}$$

satisfying the goal here.

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

By induction, we have $[\Gamma'] + \Gamma, x : A \vdash [t'/s]t : B$. We need to show that $[\Gamma'] + \Gamma \vdash [t'/s]\lambda x.t : A \multimap B$.

Applying the ABS rule to the induction hypothesis we get $[\Gamma'] + \Gamma \vdash \lambda x.[t'/s]t : A \multimap B$.

Then by the definition of substitution we have $[\Gamma'] + \Gamma \vdash [t'/s]\lambda x.t : A \multimap B$.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

We need to show that $[\Gamma'] + \Gamma_1 + \Gamma_2 \vdash [t'/s]t_1\, t_2 : B$.

By the definition of context addition we have either $s \in \Gamma_1$ or $s \in \Gamma_2$ or both, so there are three cases to consider.

1. If $s \in \Gamma_1$, then by induction on the first premise, we have $[\Gamma'] + \Gamma_1 \vdash [t'/s]t_1 : A \multimap B$. Applying the APP rule to this and the second premise gives $[\Gamma'] + \Gamma_1 + \Gamma_2 \vdash [t'/s]t_1\, t_2 : B$.

   Since $s \notin \Gamma_2$, $[t'/s]t_1\, [t'/s]t_2$ (which we need) is equivalent to $[t'/s]t_1\, t_2$ (which we have).

2. If $s \in \Gamma_2$, then by induction on the second premise, we have $[\Gamma'] + \Gamma_2 \vdash [t'/s]t_2 : A$. Applying the APP rule to this and the first premise gives $[\Gamma'] + \Gamma_1 + \Gamma_2 \vdash t_1\, ([t'/s]t_2) : B$.

   Since $s \notin \Gamma_1$, $[t'/s]t_1\, [t'/s]t_2$ (which we need) is equivalent to $t_1\, [t'/s]t_2$ (which we have).

3. If $s \in \Gamma_2$ and $s \in \Gamma_1$ then by induction on the both premise, we have $[\Gamma'] + \Gamma_2 \vdash [t'/s]t_2 : A$ and $[\Gamma'] + \Gamma_1 \vdash [t'/s]t_1 : A \multimap B$. Applying the APP rule to this and the first premise gives

   $[\Gamma'] + [\Gamma'] + \Gamma_1 + \Gamma_2 \vdash ([t'/s]t_1)\, ([t'/s]t_2) : B$.

   By the definition of context addition $[\Gamma'] + [\Gamma'] = [\Gamma']$ so we have the goal here.

- (dereliction)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A] \vdash t : B}\ \textsc{der}$$

There are two cases to consider depending on the location of s:

1. $x \equiv s$ Therefore the goal is $[\Gamma'] + \Gamma \vdash [t'/s]t : B$ which follows by applying Lemma 17 on the premise
2. $x \not\equiv s$, i.e., $x \in \mathsf{dom}(\Gamma)$ i.e., $\Gamma = \Gamma'', s : [S]$ then we induct on the premise to get $([\Gamma'] + \Gamma''), x : A \vdash [t'/s]t : B$ on which we apply dereliction again to get the goal.

- (bangIntro)

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash !t : !A}\ !_I$$

Let $\Gamma = \Gamma_1, s : S$ thus we can induct on the premise yielding $[\Gamma'] + [\Gamma_1] \vdash [t'/s]t : A$ upon which we can apply promotion again yielding:

$$\frac{[\Gamma'] + [\Gamma_1] \vdash [t'/s]t : A}{[\Gamma'] + [\Gamma_1] \vdash !([t'/s]t) : !A}\ !_I$$

which satisfies the goal since syntactic substitution is defined $[t'/s]!t = !([t'/s]t)$.

- (bangElim)

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ !x = t_1\ \mathsf{in}\ t_2 : B}\ !_E$$

We need to show that $[\Gamma'] + \Gamma_1 + \Gamma_2 \vdash [t'/s](\mathsf{let}\ !x = t_1\ \mathsf{in}\ t_2) : B$.

The reasoning is almost identical to that of the application case where (app) where we inductively apply the substitution and we rely on the property that $[\Gamma'] + [\Gamma'] = [\Gamma']$ in the case that s appears in both subterms.

- (pairIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

As above for (bangElim) and (app), same proof pattern.

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ (x, y) = t_1\ \mathsf{in}\ t_2 : C}\ \otimes_E$$

Same dual induction as for (app), (bangElim), (pairIntro).

- (unitIntro)

$$\frac{}{[\Gamma] \vdash \mathsf{unit} : 1}\ 1_I$$

Let $[\Gamma] = [\Gamma_1], s : [S], [\Gamma_2]$ then since $[t'/s]\mathsf{unit} = \mathsf{unit}$ then we can get the goal $\Gamma' + [\Gamma] \vdash [t'/s]\mathsf{unit} : 1$ simply by reapplying this typing rule with the $\Gamma' + [\Gamma]$ context.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : 1 \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let\ unit}\ = t_1\ \mathsf{in}\ t_2 : B}\ 1_E$$

Same proof pattern as in (app), (pairIntro) etc.

- (borrow)

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathsf{share}\, t : !A}\ \textsc{share}$$

By induction, we have $[\Gamma'] + \Gamma \vdash [t'/s]t : *A$. We need to show that $[\Gamma'] + \Gamma \vdash [t'/s](\mathsf{share}\, t) : !A$.

Applying the SHARE rule to the induction hypothesis we get $[\Gamma'] + \Gamma \vdash (\mathsf{share}\, [t'/s]t) : !A$.

$[t'/s](\mathsf{share}\, t)$ is equivalent to $(\mathsf{share}\, [t'/s]t)$, which we have.

- (copy)

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\, t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \textsc{clone}$$

Same pattern as (app), (pairElim) etc.

- (nec)

$$\frac{\emptyset \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

with $\Gamma = \Gamma_0, s : S$ to match the form of this lemma.

The (closed) premise implies that $[t'/s]t = t$. Then the typing rule NEC can be reapplied to yield:

$$\frac{\emptyset \vdash [t'/s]t : A}{[\Gamma_0 + \Gamma'] \vdash *([t'/s]t) : *A}\ \textsc{nec}$$

where $[\Gamma_0 + \Gamma'] = [\Gamma_0] + [\Gamma']$ and where $*([t'/s]t) = [t'/s](*t)$ by the definition of substitution, matching the goal of this lemma.

Holds trivially since the context is empty.

- (primitives - newArray):

$$\frac{}{[\Gamma] \vdash \mathsf{newArray} : \mathbb{N} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{new}$$

with the assumption that $\Gamma = \Gamma_0, s : S$. Then we can reapply the typing rule here to get:

$$\frac{}{[\Gamma_0 + \Gamma'] \vdash \mathsf{newArray} : \mathbb{N} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{new}$$

satisfying the lemma.

- (remaining primitives)

$$\frac{}{[\Gamma] \vdash \mathsf{readArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{read}$$

$$\frac{}{[\Gamma] \vdash \mathsf{writeArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{write}$$

$$\frac{}{[\Gamma] \vdash \mathsf{deleteArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap 1}\ \mathsf{del}$$

each has the same proof structure as for newArray where we simply reapply the typing with the different context (which is unconstrained in these rules).

□

Lemma 7 (Weakening is admissible)**.** *Given* $\Gamma \vdash t : A$ *then* $\Gamma, [\Gamma'] \vdash t : A$.

Proof. By induction on typing.

- (var)

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \textsc{var}$$

Typing of variables naturally includes weakening since we can introduce any additional context thus we get here:

$$\frac{}{[\Gamma, \Gamma'], x : A \vdash x : A}\ \textsc{var}$$

which gives the goal by the definition of the all-non-linear predicate (Def. 13).

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

By induction.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

By induction on either premise to add the weakening context, which is then preserved by $+$.

- (derelict)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A] \vdash t : B}\ \textsc{der}$$

By induction.

- (bangIntro)

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash\, !t : !A}\ !_I$$

By induction via the definition of the all-non-linear predicate (Def. 13).

- (bangElim)

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\, !x = t_1\, \mathsf{in}\, t_2 : B}\ !_E$$

By induction on either premise to add the weakening context, which is then preserved by $+$.

- (pairIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

By induction on either premise to add the weakening context, which is then preserved by $+$.

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\, (x, y) = t_1\, \mathsf{in}\, t_2 : C}\ \otimes_E$$

By induction on either premise to add the weakening context, which is then preserved by $+$.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : 1 \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let\ unit}\ = t_1\, \mathsf{in}\, t_2 : B}\ 1_E$$

By induction on either premise to add the weakening context, which is then preserved by $+$.

- (borrow)

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathsf{share}\, t : !A}\ \textsc{share}$$

By induction.

- (copy)

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\, t_1\, \mathsf{as}\, x\, \mathsf{in}\, t_2 : !B}\ \textsc{clone}$$

By induction on either premise to add the weakening context, which is then preserved by $+$.

- (nec)

$$\frac{\emptyset \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

The context in the premise is empty so we can naturally introduce any additional context in the succedent, giving the goal by the definition of the all-non-linear predicate (Def. 13).

- (newArray)

$$\frac{}{[\Gamma] \vdash \mathsf{newArray} : \mathbb{N} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{new}$$

By re-applying the rule and adding in the extra weakened assumptions, where $[\Gamma, \Gamma'] = [\Gamma], [\Gamma']$.

- (readArray)

$$\frac{}{[\Gamma] \vdash \mathsf{readArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{read}$$

As above, by re-applying the rule and adding in the extra weakened assumptions, where $[\Gamma, \Gamma'] = [\Gamma], [\Gamma']$.

- (writeArray)

$$\frac{}{[\Gamma] \vdash \mathsf{writeArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{write}$$

As above, by re-applying the rule and adding in the extra weakened assumptions, where $[\Gamma, \Gamma'] = [\Gamma], [\Gamma']$.

- (deleteArray)

$$\frac{}{[\Gamma] \vdash \mathsf{deleteArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap 1}\ \mathsf{del}$$

As above, by re-applying the rule and adding in the extra weakened assumptions, where $[\Gamma, \Gamma'] = [\Gamma], [\Gamma']$.

□

B.3 TYPE SAFETY

B.3.1 Value lemma

Lemma b.3.1 (Value lemma)**.** *Given* $\Gamma \vdash v : A$ *then depending on the type, the shape of* v *can be inferred:*

- $A = A' \to B$ *then* $v = \lambda x.t$ *or a partially applied primitive term* p
- $A = !A'$ *then* $v = !t$
- $A = A' \otimes B'$ *then* $v = (t_1, t_2)$
- $A = 1$ *then* $v = \mathsf{unit}$
- $A = *A'$ *then* $v = *v'$
- $A = \mathbb{N}$ *then* $v = n$
- $A = \mathbb{F}$ *then* $v = f$
- $A = \mathsf{Array}\ \mathbb{F}$ *then* $v = a$

Proof. Recall the value sub terms grammar is:

$$v ::= (t_1, t_2) \mid \mathsf{unit} \mid *t \mid\ !t \mid \lambda x.t \mid i \mid a \mid p$$
(value terms sub-grammar)

where p are partially-applied primitives, such as newArray, readArray, or readArray $(*a)$. i.e.,

$$p ::= \mathsf{newArray} \mid \mathsf{readArray} \mid \mathsf{readArray}\ (*a) \\ \mid \mathsf{writeArray} \mid \mathsf{writeArray}\ (*a) \mid \mathsf{writeArray}\ (*a)\ i \mid \mathsf{deleteArray}$$

We then proceed by case analysis on the type A to match the structure of the lemma. In each case we must consider what possible values can be assigned the type A and by which rules.

In all cases, there exists a derivation based on dereliction, e.g., for the case where $A = A' \multimap B$

$$\frac{\Gamma, x : A' \vdash t : A' \multimap B}{\Gamma, x : [A'] \vdash t : A' \multimap B}\ \textsc{der}$$

in which case we can apply induction on the premise to get the result since the term is preserved between the premise and the conclusion. We elide handling this separately each time in the cases that follow as the reasoning through dereliction is the same each time.

- $A = A' \to B$ then there are two classes of possible typing:
  - Abstract term:

$$\frac{\Gamma, x : A' \vdash t : B}{\Gamma \vdash \lambda x.t : A' \multimap B}\ \textsc{abs}$$

    thus $v = \lambda x.t$ as in the lemma statement.

– Primitive term p formed by an application of zero or more values to a primitive operation, of which there are then all seven possibilities:

1.

$$\frac{}{[\Gamma] \vdash \mathsf{newArray} : \mathbb{N} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{new}$$

thus $v = \mathsf{newArray}$

2.

$$\frac{}{[\Gamma] \vdash \mathsf{readArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{read}$$

thus $v = \mathsf{readArray}$

3.

$$\frac{\dfrac{}{[\Gamma] \vdash \mathsf{readArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{read} \quad \dfrac{}{[\Gamma], a : \mathsf{Array}\ A \vdash *a : *(\mathsf{Array}\ A)}\ *\mathsf{arr}}{[\Gamma], a : \mathsf{Array}\ A \vdash \mathsf{readArray}\ (*a) : \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \textsc{app}$$

thus $v = \mathsf{readArray}\ (*a)$

4.

$$\frac{}{[\Gamma] \vdash \mathsf{writeArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{write}$$

thus $v = \mathsf{writeArray}$

5.

$$\frac{\dfrac{}{[\Gamma] \vdash \mathsf{writeArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{write} \quad \dfrac{}{[\Gamma], a : \mathsf{Array}\ A \vdash *a : *(\mathsf{Array}\ A)}\ *\mathsf{arr}}{[\Gamma], a : \mathsf{Array}\ A \vdash \mathsf{writeArray}\ (*a) : \mathbb{N} \multimap \mathbb{F} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \textsc{app}$$

thus $v = \mathsf{writeArray}\ (*a)$

6.

$$\frac{\dfrac{\dfrac{}{[\Gamma] \vdash \mathsf{writeArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{write} \quad \dfrac{}{[\Gamma], a : \mathsf{Array}\ A \vdash *a : *(\mathsf{Array}\ A)}\ *\mathsf{arr}}{[\Gamma], a : \mathsf{Array}\ A \vdash \mathsf{writeArray}\ (*a) : \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \textsc{app} \quad \emptyset \vdash i : \mathbb{N}}{[\Gamma], a : \mathsf{Array}\ A \vdash \mathsf{writeArray}\ (*a)\ i : \mathbb{F} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \textsc{app}$$

thus $v = \mathsf{writeArray}\ (*a)$

7.

$$\frac{}{[\Gamma] \vdash \mathsf{deleteArray} : *(\mathsf{Array}\ \mathbb{F}) \multimap 1}\ \mathsf{del}$$

thus $v = \mathsf{deleteArray}$

- $A = !A'$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{[\Gamma] \vdash t : A'}{[\Gamma] \vdash !t : !A'} \; !_I$$

thus $v = !t$ as in the lemma statement.

- $A = A' \otimes B'$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{\Gamma_1 \vdash t_1 : A' \quad \Gamma_2 \vdash t_2 : B'}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A' \otimes B'} \; \otimes_I$$

thus $v = (t_1, t_2)$ as in the lemma statement.

- $A = 1$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{}{[\Gamma] \vdash \mathsf{unit} : 1} \; 1_I$$

thus $v = \mathsf{unit}$ as in the lemma statement.

- $A = *A'$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{\emptyset \vdash t : A'}{[\Gamma] \vdash *t : *A'} \; \textsc{nec}$$

thus $v = *t$ as in the lemma statement.

- $A = \mathbb{N}$ then $v = n$ Trivial case on typing of constants which is elided in this paper for brevity (but covered by the core type theory of Granule for example).

- $A = \mathbb{F}$ then $v = f$ Trivial case on typing of constants which is elided in this paper for brevity (but covered by the core type theory of Granule for example).

- $A = \mathsf{Array}\ \mathbb{F}$ then $v = a$ then the only possible typing that is a value is given by:

$$\frac{}{[\Gamma], a : \mathsf{Array}\ A \vdash a : \mathsf{Array}\ A} \; \mathsf{arr}$$

(and since the only type of arrays is $\mathbb{F}$ currently).

□

B.3.2 Progress proof

Theorem 3 (Progress). *Values of the heap model v are given by:*

$$v ::= (t_1, t_2) \mid \mathsf{unit} \mid *t \mid \ !t \mid \lambda x.t \mid i \mid a \mid p$$
(value terms sub-grammar)

where p are partially-applied primitives, such as newArray, readArray, *or* readArray $(*a)$. *Given* $\Gamma \vdash t : A$, *then t is either a value, or if* $H \bowtie \Gamma_0 + \Gamma$ *there exists a heap* H', *term* t', *usage context* Δ, *and context* Γ' *such that* $H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma' \mid \Delta$.

Proof. By induction on typing.

- (var)

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \textsc{var}$$

with $H \bowtie [\Gamma], x : A$ which by inversion of heap compatibility could imply either $H = H', x{\mapsto_1}t'$ or $H = H', x{\mapsto_\omega}t'$, for which each has a reduction:

1. $H', x{\mapsto_1}t'$

$$\frac{}{H, x{\mapsto_1}t \vdash x \rightsquigarrow H \vdash t \mid \emptyset \mid x : 1}\ \rightsquigarrow_{\textsc{var}1}$$

2. $H', x{\mapsto_\omega}t'$

$$\frac{}{H, x{\mapsto_\omega}t \vdash x \rightsquigarrow H, x{\mapsto_\omega}t \vdash t \mid \emptyset \mid x : 1}\ \rightsquigarrow_{\textsc{var}\omega}$$

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

A value.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

By induction on the first premise, there are two possibilities.

1. t_1 is a value and therefore by the value lemma there are a number of choices:
   - $t_1 = \lambda x.t_1'$. Therefore we can reduce by $\rightsquigarrow_\beta$.
   - $t_1 = \mathsf{newArray}$ therefore $A = \mathbb{N}$

     Therefore we induct on the second argument:

* t_2 is a value and therefore by the value lemma (Lemma 10) $t_2 = n$ and thus the typing is:

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{newArray}\, n : *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{new}$$

with $H \bowtie (\Gamma_0 + \Gamma)$.

Thus there is a reduction as follows:

$$\frac{a \# H}{H \vdash \mathsf{newArray}\, n \rightsquigarrow H, a \mapsto_1 \mathbf{arr} \vdash *a \mid \emptyset \mid \emptyset}\ \rightsquigarrow_{\mathsf{newArray}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{newArray}\, t_2 \rightsquigarrow H' \vdash \mathsf{newArray}\, t_2' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{prim}}$$

– $t_1 = \mathsf{readArray}$ therefore $A = *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})$

* t_2 is a value and therefore by the value lemma on $*(\mathsf{Array}\ \mathbb{F})$ (Lemma 10) $t_2 = (*a)$ and thus $t_1\, t_2 = \mathsf{readArray}\, (*a)$ which is also a value.
* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{readArray}\, t_2 \rightsquigarrow H' \vdash \mathsf{readArray}\, t_2' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{prim}}$$

– $t_1 = \mathsf{readArray}\, (*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})$

* t_2 is a value and therefore by the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{N}$ (Lemma 10) implies $t_2 = n$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \mathsf{Array}\ \mathbb{F} \vdash a : (\mathsf{Array}\ \mathbb{F})\ \mathsf{arr}}{[\Gamma_1], a : \mathsf{Array}\ \mathbb{F} \vdash *a : *(\mathsf{Array}\ \mathbb{F})}\ *\mathsf{arr} \qquad \Gamma_2 \vdash i : \mathbb{N}}{[\Gamma_1] + \Gamma_2, a : \mathsf{Array}\ \mathbb{F} \vdash \mathsf{readArray}\, a\, i : \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{read}$$

with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2, a : \mathsf{Array}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_r \mathbf{arr}$.

Then there is a reduction as follows:

$$\frac{}{H, a \mapsto_r (\mathbf{arr}[i] = v) \vdash \mathsf{readArray}\, (*a)\, i \rightsquigarrow H, a \mapsto_r (\mathbf{arr}[i] = v) \vdash (v, *a) \mid \emptyset \mid \emptyset}\ \rightsquigarrow_{\mathsf{readArray}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{readArray}\,(*a)\,t_2 \rightsquigarrow H' \vdash \mathsf{readArray}\,(*a)\,t_2' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{prim}}$$

– $t_1 = \mathsf{writeArray}$ therefore $A = *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F})$

* t_2 is a value therefore by the value lemma (Lemma 10) $t_2 = (*a)$ and thus $t_1\,t_2 = \mathsf{writeArray}\,(*a)$ which is also a value.
* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{writeArray}\,t_2 \rightsquigarrow H' \vdash \mathsf{writeArray}\,t_2' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{prim}}$$

– $t_1 = \mathsf{writeArray}\,(*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F})$

* t_2 is a value therefore by the value lemma (Lemma 10) $t_2 = n$ and thus $t_1\,t_2 = \mathsf{writeArray}\,(*a)\,n$ which is also a value.
* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{writeArray}\,(*a)\,t_2 \rightsquigarrow H' \vdash \mathsf{writeArray}\,(*a)\,t_2' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{prim}}$$

– $t_1 = \mathsf{writeArray}\,(*a)\,i$ therefore $A = \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})$

* t_2 is a value and therefore by the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{F}$ (Lemma 10) implies $t_2 = f$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \mathsf{Array}\ \mathbb{F} \vdash a : (\mathsf{Array}\ \mathbb{F})\ \mathsf{arr}}{[\Gamma_1], a : \mathsf{Array}\ \mathbb{F} \vdash *a : *(\mathsf{Array}\ \mathbb{F})}\ *\mathsf{arr} \qquad \Gamma_2 \vdash i : \mathbb{N} \quad \Gamma_3 \vdash f : \mathbb{F}}{[\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}\ \mathbb{F} \vdash \mathsf{writeArray}\,a\,i\,f : *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{write}$$

with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_r \mathbf{arr}$.

Then there is a reduction as follows:

$$\frac{}{H, a \mapsto_r \mathbf{arr} \vdash \mathsf{writeArray}\,(*a)\,i\,v \rightsquigarrow H, a \mapsto_r (\mathbf{arr}[i] = v) \vdash *a \mid \emptyset \mid \emptyset}\ \rightsquigarrow_{\mathsf{writeArray}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{writeArray}\,(*a)\,i\,t_2 \rightsquigarrow H' \vdash \mathsf{writeArray}\,(*a)\,i\,t_2' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{prim}}$$

– $t_1 = \mathsf{deleteArray}$ therefore $A = *(\mathsf{Array}\ \mathbb{F}) \multimap 1$

* t_2 is a value and therefore by the value lemma (Lemma 10) $t_2 = a$ thus the typing is refined at runtime as follows:

$$\dfrac{\dfrac{[\Gamma], a : \mathsf{Array}\ \mathbb{F} \vdash a : (\mathsf{Array}\ \mathbb{F})\ \mathsf{arr}}{[\Gamma], a : \mathsf{Array}\ \mathbb{F} \vdash *a : *(\mathsf{Array}\ \mathbb{F})}\ *\mathsf{arr}}{[\Gamma], a : \mathsf{Array}\ \mathbb{F} \vdash \mathsf{deleteArray} : 1}\ \mathsf{del}$$

with $H' \bowtie (\Gamma_0 + [\Gamma], a : \mathsf{Array}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_r \mathbf{arr}$.

There there is a reduction as follows:

$$\dfrac{}{H, a \mapsto_r \mathbf{arr} \vdash \mathsf{deleteArray}\ (*a) \rightsquigarrow H \vdash \mathsf{unit} \mid \emptyset \mid \emptyset}\ \rightsquigarrow\mathsf{deleteArray}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\dfrac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{deleteArray}\ t_2 \rightsquigarrow H' \vdash \mathsf{deleteArray}\ t_2' \mid \Gamma \mid \Delta}\ \rightsquigarrow\mathsf{prim}$$

2. otherwise, by induction on the premise we then have that there exists heap H', term t_1', usage context Δ, and context Γ' such that $H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma' \mid \Delta$. Therefore we can reduce by $\rightsquigarrow_{\text{APP}}$.

- (derelict)

$$\dfrac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A] \vdash t : B}\ \textsc{der}$$

Goal achieved immediately by induction on the premise.

- (bangIntro)

$$\dfrac{[\Gamma] \vdash t : A}{[\Gamma] \vdash\ !t : !A}\ !_I$$

A value.

- (bangElim)

$$\dfrac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ !x = t_1\ \mathsf{in}\ t_2 : B}\ !_E$$

By induction on the first premise, there are two possibilities.

1. t_1 is a value and therefore by the value lemma $t_1 = !t_1'$. This refines the typing as follows:

$$\dfrac{\dfrac{[\Gamma_1] \vdash t_1' : A}{[\Gamma_1] \vdash !t_1' : !A}\,!_I \qquad \Gamma_2, x : [A] \vdash t_2 : B}{[\Gamma_1] + \Gamma_2 \vdash \mathsf{let}\, !x = !t_1' \,\mathsf{in}\, t_2 : B}\,!_E$$

   Therefore we can reduce by $\rightsquigarrow_{!\beta}$.

2. Otherwise by induction on the premise we then have that there exists heap H', term t_1', usage context Δ, and context Γ' such that $H| - t1 \; > H'| - t1'; G'; D$. Therefore we can reduce by $\rightsquigarrow_{\text{LET}!}$.

- (pairIntro)

$$\dfrac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\,\otimes_I$$

  A value.

- (pairElim)

$$\dfrac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,(x, y) = t_1 \,\mathsf{in}\, t_2 : C}\,\otimes_E$$

  By induction on the first premise, there are two possibilities.

1. t_1 is a value and therefore by the value lemma $t_1 = (t_1', t_1'')$. This refines the typing as follows:

$$\dfrac{\dfrac{\Gamma_1 \vdash t_1' : A \qquad \Gamma_2 \vdash t_1'' : B}{\Gamma_1 + \Gamma_2 \vdash (t_1', t_1'') : A \otimes B}\,\otimes_I \qquad \Gamma_3, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \mathsf{let}\,(x, y) = (t_1', t_1'') \,\mathsf{in}\, t_2 : C}\,\otimes_E$$

   Therefore we can reduce by $\rightsquigarrow_{\otimes\beta}$.

2. Otherwise by induction on the premise we then have that there exists heap H', term t_1', usage context Δ, and context Γ' such that $H| - t \; > H'| - t1'; G'; D$. Therefore we can reduce by $\rightsquigarrow_{\text{LET}\otimes}$.

- (unitIntro)

$$\dfrac{}{[\Gamma] \vdash \mathsf{unit} : 1}\,1_I$$

  A value.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : 1 \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let\ unit}\ = t_1\ \mathsf{in}\ t_2 : B}\ 1_E$$

By induction on the first premise, there are two possibilities.

1. t is a value and therefore by the value lemma $t = \mathsf{unit}$. Therefore we can reduce by $\rightsquigarrow_{\beta\mathsf{unit}}$.
2. Otherwise by induction on the premise we then have that there exists heap H', term t_1', usage context Δ, and context Γ' such that $H|-t\ > H'|-t1'; G'; D$. Therefore we can reduce by $\rightsquigarrow_{\text{LET}\mathsf{unit}}$.

- (share)

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathsf{share}\, t : !A}\ \textsc{share}$$

By induction on the premise, there are two possibilities.

1. t is a value and therefore by the value lemma $t = *t'$. Therefore we can reduce by $\rightsquigarrow_{\beta\mathsf{share}}$.
2. Otherwise by induction on the premise we then have that there exists heap H', term t', usage context Δ, and context Γ' such that $H|-t\ > H'|-t'; G'; D$. Therefore we can reduce by $\rightsquigarrow_{\mathsf{share}}$.

- (clone)

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\, t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \textsc{clone}$$

By induction on the first premise, there are two possibilities.

1. t_1 is a value and therefore by the value lemma $t_1 = !t$. Then there are again two possibilities.
   a) t is itself a value. This refines the typing as follows:

$$\frac{\dfrac{[\Gamma_1] \vdash v : A}{[\Gamma_1] \vdash !v : !A}\ !_I \qquad \Gamma_2, x : *A \vdash t_2 : !B}{[\Gamma_1] + \Gamma_2 \vdash \mathsf{clone}\, !v\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \textsc{clone}$$

   Therefore we can reduce using $\rightsquigarrow_{\mathsf{clone}\beta}$.
   b) Otherwise by induction on the premise we then have that there exists heap H', term t', usage context Δ, and context Γ' such that $H|-t\ > H'|-t'; G'; D$. Therefore we can reduce by $\rightsquigarrow_{\mathsf{clone}!}$.

2. Otherwise by induction on the premise we then have that there exists heap H', term t_1', usage context Δ, and context Γ' such that $H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma' \mid \Delta$. Therefore we can reduce by $\rightsquigarrow_{\mathsf{clone}}$.

- (nec)

$$\frac{\varnothing \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

Then by induction either t is a value so $*t$ is a value or there is a reduction which we can plug into the following reduction:

$$\frac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash *t \rightsquigarrow H' \vdash *t' \mid \Gamma \mid \Delta}\ \rightsquigarrow_*$$

- (array)

$$\frac{}{[\Gamma], a : \mathsf{Array}\ A \vdash a : \mathsf{Array}\ A}\ \mathsf{arr}$$

A value.

- (uniqueArray)

$$\frac{}{[\Gamma], a : \mathsf{Array}\ A \vdash *a : *(\mathsf{Array}\ A)}\ *\mathsf{arr}$$

A value.

□

B.3.3 Conservation proof

Theorem 4 (Conservation). *For a well-typed term $\Gamma \vdash t : A$ and all Γ_0 and H such that $H \bowtie (\Gamma_0 + \Gamma)$ and a reduction $H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma_1 \mid \Delta$ we have:*

$$\exists \Gamma'.\ \Gamma' \vdash t' : A \ \wedge\ H' \bowtie (\Gamma_0 + \Gamma') \ \wedge\ (\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{\Gamma_1})$$

Proof. By induction on the structure of typing and reductions.

- (var)

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \textsc{var}$$

With two possible reductions and heap compatibilities:

– With heap compatibility derivation:

$$\frac{H' \bowtie \Gamma_0 + [\Gamma] + \Gamma' \qquad \Gamma' \vdash t : A}{H', x \mapsto_1 (\Gamma' \vdash t : A) \bowtie (\Gamma_0 + [\Gamma], x : A)}\ \textsc{Lin}$$

and the reduction:

$$\frac{}{H', x{\mapsto_1}(\Gamma' \vdash t' : A) \vdash x \rightsquigarrow H' \vdash t' \mid \emptyset \mid x : 1} \rightsquigarrow_{\text{VAR1}}$$

The resulting typing judgment is then given by $\Gamma' \vdash t : A$, which by Lemma 7 (weakening) can be weakened to $[\Gamma], \Gamma' \vdash t : A$ to give the typing result here (first conjunct) and then the required concluding heap compatibility derivation (second conjunct) of $H' \bowtie \Gamma_0 + [\Gamma] + \Gamma'$ is given by the first premise of the incoming heap compatibility above.

Finally, the third goal conjunct is:

$$\begin{array}{lll} & \overline{(H')} + x : 1 & \sqsubseteq \quad \overline{(H', x{\mapsto_1}(\Gamma' \vdash t : A))}, \emptyset \\ \Longrightarrow & \overline{H'}, x : 1 & \sqsubseteq \quad \overline{H'}, x : 1 \end{array}$$

which follows by reflexivity of the preorder.

– With heap compatibility derivation:

$$\frac{H_1 \bowtie \Gamma_0 + [\Gamma] + [\Gamma'] \qquad [\Gamma'] \vdash t' : A}{H_1, x{\mapsto_\omega}(\Gamma' \vdash t : A) \bowtie (\Gamma_0 + [\Gamma], x : A)} \textsc{Lin}$$

and thus the only possible reduction is:

$$\frac{}{H_1, x{\mapsto_\omega}([\Gamma'] \vdash t' : A) \vdash x \rightsquigarrow H_1, x{\mapsto_\omega}([\Gamma'] \vdash t' : A) \vdash t' \mid \emptyset \mid x : 1} \rightsquigarrow_{\text{VAR}\omega}$$

The resulting typing judgment is now given by $[\Gamma] + [\Gamma'], x : [A] \vdash t' : A$ obtained by weakening (Lemma 7) on $[\Gamma'] \vdash t' : A$ to introduce the assumption of $x : [A]$ (note, now non-linear) and the previous context $[\Gamma]$. This then allows the final heap to be compatible with typing.

The final heap is $H' = H_1, x{\mapsto_\omega}([\Gamma'] \vdash t' : A)$ and then the required heap compatibility derivation is derived uses the premises of the incoming heap compatibility above but using ω and relying on idempotence of context addition:

$$\frac{\dfrac{H_1 \bowtie \Gamma_0 + [\Gamma] + [\Gamma']}{H_1 \bowtie \Gamma_0 + [\Gamma] + [\Gamma'] + [\Gamma']}\ \textit{idem.} \qquad [\Gamma'] \vdash t' : A}{H_1, x{\mapsto_\omega}([\Gamma'] \vdash t : A) \bowtie (\Gamma_0 + [\Gamma] + [\Gamma'], x : [A])}\ \omega$$

Finally, the third goal conjunct is:

$$\begin{array}{lll} & \overline{(H_1, x{\mapsto_\omega}([\Gamma'] \vdash t : A))} + x : 1 & \sqsubseteq \quad \overline{(H_1, x{\mapsto_\omega}([\Gamma'] \vdash t : A))} + \emptyset \\ \Longrightarrow & \overline{H_1}, x : \omega + x : 1 & \sqsubseteq \quad \overline{H}, x : \omega \\ \Longrightarrow & \overline{H_1}, x : \omega & \sqsubseteq \quad \overline{H}, x : \omega \end{array}$$

which follows by reflexivity of the preorder and by absorption as $\omega + 1 = \omega$.

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

Has no reduction so the case is trivial here.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

For which there are three possible reductions (one of which refines the typing)

  - $t_1 = \lambda x.t_1'$ thus the typing is:

$$\frac{\dfrac{\Gamma_1, x : A \vdash t_1' : B}{\Gamma_1 \vdash \lambda x.t_1' : A \to B}\ \textsc{abs} \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash (\lambda x.t_1')\, t_2 : B}\ \textsc{app}$$

    and we have compatibility $H \bowtie \Gamma_0 + (\Gamma_1 + \Gamma_2)$ with one possible reduction:

$$\frac{}{H \vdash (\lambda x.t_1')\, t_2 \rightsquigarrow H, x \mapsto_1 (\Gamma_2 \vdash t_2 : A) \vdash t_1' \mid x : A \mid \emptyset}\ \rightsquigarrow_\beta$$

    Therefore the resulting typing judgment is $\Gamma_1, x : A \vdash t_1' : B$ and we construct the goal compatibility judgment as follows, using the incoming compatibility assumption:

$$\frac{H \bowtie \Gamma_0 + \Gamma_1 + \Gamma_2}{(H, x \mapsto_1 (\Gamma_2 \vdash t_2 : A)) \bowtie \Gamma_0 + (\Gamma_1, x : A)}\ \textsc{Lin}$$

    (where $x \notin \mathsf{dom}(\Gamma_0)$ since it was bound in the term here).

    The remaining goal is then:

$$\begin{array}{lll} & (\overline{(H, x \mapsto_1 (\Gamma \vdash t_2 : A))} + \emptyset) & \sqsubseteq \quad (\overline{H}, \overline{x : A}) \\ \Longrightarrow & (\overline{H}, x : 1) & \sqsubseteq \quad (\overline{H}, x : 1) \end{array}$$

    which follows by reflexivity of the preorder.

  - otherwise application, evaluating the left

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma \mid \Delta}{H \vdash t_1\, t_2 \rightsquigarrow H' \vdash t_1'\, t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\textsc{app}}$$

    with compatibility $H \bowtie (\Gamma_0 + (\Gamma_1 + \Gamma_2))$.

    Apply this theorem inductively on $\Gamma_1 \vdash t_1 : A \to B$ with Γ_0 (in the recursive call) instantiated to $\Gamma_0 + \Gamma_2$ which yields $\Gamma_1' \vdash t_1' : A \to B$ with H' such that $H' \bowtie ((\Gamma_0 + \Gamma_2) + \Gamma_1')$

and $(\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{\Gamma})$ (which subsequently provides the third goal immediately).

Thus the resulting typing judgment for this case is the application:

$$\frac{\Gamma_1' \vdash t_1' : A \to B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1' + \Gamma_2 \vdash (\lambda x.t_1')\, t_2 : B}\ \text{APP}$$

Therefore the resulting context is $\Gamma' = \Gamma_1' + \Gamma_2$ for which the heap compatibility judgment $H' \bowtie (\Gamma_0 + (\Gamma_1' + \Gamma_2))$ follows from the inductive call via associativity and commutativity of context +.

– or $t_1 = p$ (primitive partially-applied value) and t_2 is not a value thus we have the reduction

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash p\, t_2 \rightsquigarrow H' \vdash p\, t_2' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\text{prim}}$$

with compatibility $H \bowtie (\Gamma_0 + (\Gamma_1 + \Gamma_2))$.

Applying the theorem inductively on $\Gamma_2 \vdash t_2 : A$ (second typing premise) with the Γ_0 parameter of the recursive call instantiated to $\Gamma_0 + \Gamma_1$ which yields $\Gamma_2' \vdash t_2' : A$ with H' such that $H' \bowtie ((\Gamma_0 + \Gamma_1) + \Gamma_2')$ and $(\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{\Gamma})$ which then provides the third goal conjunct here immediately.

The result typing judgment is the application:

$$\frac{\Gamma_1 \vdash p : A \to B \quad \Gamma_2' \vdash t_2' : A}{\Gamma_1 + \Gamma_2' \vdash p\, t_2' : B}\ \text{APP}$$

Therefore the resulting context is $\Gamma' = \Gamma_1 + \Gamma_2'$ for which the heap compatibility judgment $H' \bowtie (\Gamma_0 + (\Gamma_1 + \Gamma_2'))$ follows from the inductive call via associativity and commutativity of context +.

– $t_1 = \mathsf{newArray}$ and $t_2 = n$ There is only one possible reduction:

$$\frac{a \# H}{H \vdash \mathsf{newArray}\, n \rightsquigarrow H, a \mapsto_1 \mathbf{arr} \vdash *a \mid \emptyset \mid \emptyset}\ \rightsquigarrow_{\mathsf{newArray}}$$

But here $\Gamma = \Delta = \emptyset$ and there is no new grading information on the heap, so the goal follows by reflexivity of the preorder.

– $t_2 = \mathsf{readArray}\,(*a)$ and $t_2 = n$. There is only one possible reduction:

$$\frac{}{H, a \mapsto_r (\mathbf{arr}[i] = v) \vdash \mathsf{readArray}\,(*a)\, i \rightsquigarrow H, a \mapsto_r (\mathbf{arr}[i] = v) \vdash (v, *a) \mid \emptyset \mid \emptyset}\ \rightsquigarrow_{\mathsf{readArray}}$$

But here $\Gamma = \Delta = \emptyset$ and there is no new grading information on the heap, so the goal follows by reflexivity of the preorder.

– $t_2 = \mathsf{writeArray}\,(*a)\,n$ and $t_2 = f$. There is only one possible reduction:

$$\frac{}{H, a{\mapsto_r}\mathbf{arr} \vdash \mathsf{writeArray}\,(*a)\,i\,v \rightsquigarrow H, a{\mapsto_r}(\mathbf{arr}[i] = v) \vdash *a \mid \emptyset \mid \emptyset}\ {\rightsquigarrow}\mathsf{writeArray}$$

But here $\Gamma = \Delta = \emptyset$ and there is no new grading information on the heap, so the goal follows by reflexivity of the preorder.

– $t_2 = \mathsf{deleteArray}$ and $t_2 = *a$. There is only one possible reduction:

$$\frac{}{H, a{\mapsto_r}\mathbf{arr} \vdash \mathsf{deleteArray}\,(*a) \rightsquigarrow H \vdash \mathsf{unit} \mid \emptyset \mid \emptyset}\ {\rightsquigarrow}\mathsf{deleteArray}$$

But here $\Gamma = \Delta = \emptyset$ and there is no new grading information on the heap, so the goal follows by reflexivity of the preorder.

- (der)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A] \vdash t : B}\ \textsc{der}$$

with compatibility $H \bowtie (\Gamma, x : [A])$, which must have derivation (only possible one):

$$\frac{H \bowtie \Gamma_0 + \Gamma + [\Gamma_1] \qquad [\Gamma_1] \vdash t_1' : A}{(H, x{\mapsto_\omega}([\Gamma_1] \vdash t_1' : A)) \bowtie \Gamma_0 + (\Gamma, x : [A])}\ \omega$$

We can re-derive heap compatibility from the premises here as:

$$\frac{H \bowtie \Gamma_0 + \Gamma + [\Gamma_1] \qquad [\Gamma_1] \vdash t_1' : A}{(H, x{\mapsto_\omega}([\Gamma_1] \vdash t_1' : A)) \bowtie \Gamma_0 + (\Gamma, x : A)}\ \textsc{Lin}$$

This provides heap compatibility which can be used with the premise of typing to inductively apply conservation, yielding some output judgment $\Gamma' \vdash t' : B$ the goal $(\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{\Gamma_1})$ and the heap compatibility with $H' \bowtie (\Gamma_0 + \Gamma')$ which provides the goals here.

- (bangIntro)

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash\, !t : !A}\ !_I$$

Has no reduction so the case is trivial here.

- (bangElim)

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\, !x = t_1 \,\mathsf{in}\, t_2 : B}\ !_E$$

  For which there are two possible reductions:

  - $t_1 = !t_1'$ thus the typing is

$$\frac{\dfrac{[\Gamma_1] \vdash t_1' : A}{[\Gamma_1] \vdash !t_1' : !A}\ !_I \qquad \Gamma_2, x : [A] \vdash t_2 : B}{[\Gamma_1] + \Gamma_2 \vdash \mathsf{let}\, !x = !t_1' \,\mathsf{in}\, t_2 : B}\ !_E$$

    and we have compatibility $H \bowtie \Gamma_0 + ([\Gamma_1] + \Gamma_2)$ with one possible reduction:

$$\frac{}{H \vdash \mathsf{let}\, !x = !t_1' \,\mathsf{in}\, t_2 \rightsquigarrow H, x \mapsto_\omega t_1' \vdash t_2 \mid x : [A] \mid \emptyset}\ \rightsquigarrow_{!\beta}$$

    Therefore the outgoing heap is $H' = H, x \mapsto_\omega ([\Gamma_1] \vdash t_1' : A)$ and the resulting typing judgment is $\Gamma_2, x : [A] \vdash t_2 : B$ and we can construct the goal compatibility judgment as follows, using the incoming compatibility assumption:

$$\frac{H \bowtie (\Gamma_0 + [\Gamma_1]) + \Gamma_2 \qquad [\Gamma_1] \vdash t_1' : A}{(H, x \mapsto_\omega ([\Gamma_1] \vdash t_1' : A)) \bowtie \Gamma_0 + (\Gamma_2, x : [A])}\ \omega$$

    (where $x \notin \mathsf{dom}(\Gamma_0)$ since it was bound in the term here)

    The remaining goal is then:

$$\begin{array}{lll} & (\overline{(H, x \mapsto_\omega ([\Gamma_1] \vdash t_1' : A))} + \emptyset) & \sqsubseteq \quad (\overline{(H)}, \overline{x : [A]}) \\ \Longrightarrow & (\overline{H}, x : \omega) & \sqsubseteq \quad (\overline{H}, x : \omega) \end{array}$$

    which follows by reflexivity of the preorder.

  - otherwise, the reduction can proceed by evaluating t_1:

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma \mid \Delta}{H \vdash \mathsf{let}\, !x = t_1 \,\mathsf{in}\, t_2 \rightsquigarrow H' \vdash \mathsf{let}\, !x = t_1' \,\mathsf{in}\, t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathrm{LET!}}$$

    with compatibility $H \bowtie (\Gamma_0 + (\Gamma_1 + \Gamma_2))$.

    Apply this lemma inductively on $\Gamma_1 \vdash t_1 : !A$ with Γ_0 (in the recursive call) instantiated to $\Gamma_0 + \Gamma_2$ which yields $\Gamma_1' \vdash t_1' : !A$ with H' such that $H' \bowtie ((\Gamma_0 + \Gamma_2) + \Gamma_1')$ and $(\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{\Gamma})$ (which subsequently provides the third goal immediately).

- (pairIntro) with typing:

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

  Has no reduction so the case is trivial here.

- (pairElim) with typing:

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,(x,y) = t_1\,\mathsf{in}\,t_2 : C}\ \otimes_E$$

For which there are two possible reductions (one of which refines the typing):

- $t_1 = (t_1', t_1'')$ thus the typing is:

$$\frac{\dfrac{\Gamma_1 \vdash t_1' : A \qquad \Gamma_2 \vdash t_1'' : B}{\Gamma_1 + \Gamma_2 \vdash (t_1', t_1'') : A \otimes B}\ \otimes_I \qquad \Gamma_3, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \mathsf{let}\,(x,y) = (t_1', t_1'')\,\mathsf{in}\,t_2 : C}\ \otimes_E$$

with $H \bowtie \Gamma_0 + (\Gamma_1 + \Gamma_2 + \Gamma_3)$ with one possible reduction:

$$\frac{\Gamma_1 \vdash t_1' : A \qquad \Gamma_2 \vdash t_1'' : B}{H \vdash \mathsf{let}\,(x,y) = (t_1', t_1'')\,\mathsf{in}\,t_3 \rightsquigarrow H, x{\mapsto_1}(\Gamma_1 \vdash t_1' : A), y{\mapsto_1}(\Gamma_2 \vdash t_1'' : B) \vdash t_3 \mid \emptyset, x : A, y : B \mid \emptyset}\ \rightsquigarrow_{\otimes\beta}$$

Therefore the resulting typing judgment is $\Gamma_3, x : A, y : B \vdash t_2 : C$ and we construct the goal compatibility judgment as follows, using the incoming compatibility assumption:

$$\frac{\dfrac{H \bowtie \Gamma_0 + (\Gamma_1 + \Gamma_2 + \Gamma_3) \qquad \Gamma_1 \vdash t_1' : A}{H, x{\mapsto_1}(\Gamma_1 \vdash t_1' : A) \bowtie \Gamma_0 + (\Gamma_2 + \Gamma_3), x : A}\ \textsc{Lin} \qquad \Gamma_2 \vdash t_1'' : B}{H, x{\mapsto_1}(\Gamma_1 \vdash t_1' : A), y{\mapsto_1}(\Gamma_2 \vdash t_1'' : B) \bowtie \Gamma_0 + (\Gamma_3, x : A, y : B)}\ \textsc{Lin}$$

(where $x \notin \mathsf{dom}(\Gamma_0)$ and $y \notin \mathsf{dom}(\Gamma_0)$ since they were bound in the terms here).

The remaining goal is then:

$$\begin{aligned} \overline{(H, x{\mapsto_1}(\Gamma_1 \vdash t_1' : A), y{\mapsto_1}(\Gamma_2 \vdash t_1'' : B))} + \emptyset &\sqsubseteq \overline{H}, \overline{(x : A, y : B)} \\ \overline{H}, x : 1, y : 1 &\sqsubseteq \overline{H}, x : 1, y : 1 \end{aligned}$$

which follows by reflexivity of the preorder.

- otherwise evaluate the left subterm:

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma \mid \Delta}{H \vdash \mathsf{let}\,(x,y) = t_1\,\mathsf{in}\,t_2 \rightsquigarrow H' \vdash \mathsf{let}\,(x,y) = t_1'\,\mathsf{in}\,t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\textsc{let}\otimes}$$

with compatibility $H \bowtie \Gamma_0 + (\Gamma_1 + \Gamma_2)$.

Apply this lemma inductively on $\Gamma_1 \vdash t_1 : A \otimes B$ with Γ_0 (in the recursive call) instantiated to $\Gamma_0 + \Gamma_2$ which yields $\Gamma_1' \vdash t_1' : A \otimes B$ with H' such that $H' \bowtie ((\Gamma_0 + \Gamma_2) + \Gamma_1')$ and $(\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{\Gamma})$.

Thus the resulting conclusion here is the typing: $\Gamma_1' + \Gamma_2 \vdash \mathsf{let}\,(x,y) = t_1'\,\mathsf{in}\,t_2 : C$ with heap compatibility from the

induction by commutativity of associativity of + $H' \bowtie (\Gamma_0 + (\Gamma_1' + \Gamma_2))$ and since output heap and usage matches the induction then the last conjunct is from the inductive evidence $(\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{\Gamma})$.

- (unitIntro) with typing:

$$\frac{}{[\Gamma] \vdash \mathsf{unit} : 1}\ 1_I$$

Has no reduction so the case is trivial here.

- (unitElim) with typing:

$$\frac{\Gamma_1 \vdash t_1 : 1 \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let\ unit}\ = t_1\ \mathsf{in}\ t_2 : B}\ 1_E$$

Has two possible reductions.

  - $t_1 = \mathsf{unit}$

    Then the typing is:

$$\frac{\dfrac{}{[\Gamma_1] \vdash \mathsf{unit} : 1}\ 1_I \qquad \Gamma_2 \vdash t_2 : B}{[\Gamma_1] + \Gamma_2 \vdash \mathsf{let\ unit}\ = \mathsf{unit}\ \mathsf{in}\ t_2 : B}\ 1_E$$

    with $H \bowtie \Gamma_0 + ([\Gamma_1] + \Gamma_2)$ with one possible reduction:

$$\frac{}{H \vdash \mathsf{let\ unit}\ = \mathsf{unit}\ \mathsf{in}\ t_2 \rightsquigarrow H \vdash t_2 \mid \emptyset \mid \emptyset}\ \rightsquigarrow_{\beta\mathsf{unit}}$$

    The resultant typing is then obtained via weakening (Lemma 7) on the second premise to get $[\Gamma_1], \Gamma_2 \vdash t_2 : B$ such that the heap compatibility is just as before: $H \bowtie \Gamma_0 + ([\Gamma_1] + \Gamma_2)$.

    Since in the reduction both the opened up binders Γ_1 and usage context Δ here are both $\emptyset$, so the goal here follows by reflexivity of the preorder as $\overline{H} \sqsubseteq \overline{H}$.

  - $t_1 \neq \mathsf{unit}$.

    Then we have $H \bowtie \Gamma_0 + (\Gamma_1 + \Gamma_2)$ with one possible reduction:

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma \mid \Delta}{H \vdash \mathsf{let\ unit}\ = t_1\ \mathsf{in}\ t_2 \rightsquigarrow H' \vdash \mathsf{let\ unit}\ = t_1'\ \mathsf{in}\ t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{LETunit}}$$

    Apply the lemma inductively on $\Gamma_1 \vdash t_1 : 1$ with Γ_0 (in the recursive call) instantiated to $\Gamma_0 + \Gamma_2$ which yields $\Gamma_1' \vdash$

$t_1' : 1$ with H' such that $H' \bowtie ((\Gamma_0 + \Gamma_2) + \Gamma_1')$ and $(\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{\Gamma})$.

The goal typing is then $\Gamma_1' + \Gamma_2 \vdash \mathsf{let\,unit} = t_1' \,\mathsf{in}\, t_2 : B$ with heap compatibility from the inductive evidence by commutativity and associativity of $+$ $H' \bowtie ((\Gamma_0 + \Gamma_2) + \Gamma_1')$ and $(\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{\Gamma})$ from the inductive evidence provides the final conjunct goal (as the usage context and opened binders in this reduction are the same as the inductive premise).

- (share) with typing:

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathsf{share}\, t : !A}\ \textsc{share}$$

There are two possible reductions:

  - t is a value $*v$ and we reduce by:

$$\frac{\mathsf{dom}(H) \equiv \mathsf{arrRefs}(t)}{H, H' \vdash (\mathsf{share}\,(*v)) \rightsquigarrow ([H]_\omega), H' \vdash\, !v \mid \emptyset \mid \emptyset}\ \rightsquigarrow_{\beta\mathsf{share}}$$

with compatibility $H, H' \bowtie (\Gamma_0 + \Gamma)$.

By the value lemma (Lemma 10) on the typing $\Gamma \vdash *v : *A$ and by the fact that we only have unique arrays, we know that $v = a$ and so the typing here can be refined to:

$$\frac{\dfrac{}{[\Gamma_1], a : \mathsf{Array}\ A \vdash a : \mathsf{Array}\ A}\ \mathsf{arr}}{[\Gamma_1], a : \mathsf{Array}\ A \vdash *a : *(\mathsf{Array}\ A)}\ \textsc{share}$$

with heap compatibility then $H, H' \bowtie (\Gamma_0 + [\Gamma_1], a : \mathsf{Array}\ A)$. Furthermore $\mathsf{dom}(H) \equiv \mathsf{arrRefs}(t)$ then refines to $\mathsf{dom}(H) \equiv a$.

Therefore the incoming heap is of the form $(H', a \mapsto_r \mathbf{arr}) \bowtie (\Gamma_0 + [\Gamma_1], a : \mathsf{Array}\ A)$ with the only possible derivation:

$$\frac{H' \bowtie \Gamma_0 + [\Gamma_1]}{(H', a \mapsto_r \mathbf{arr}) \bowtie (\Gamma_0 + [\Gamma_1], a : \mathsf{Array}\ A)}\ \textsc{Ref}$$

The resulting typing judgment for this reduction is then derived as:

$$\frac{\dfrac{}{[\Gamma_1], a : \mathsf{Array}\ A \vdash a : \mathsf{Array}\ A}\ \mathsf{arr}}{[\Gamma_1], a : \mathsf{Array}\ A \vdash\, !a : !(\mathsf{Array}\ A)}\ !_I$$

(using Definition b.1.1 on the all-non-linear predicate redefined for runtime contexts which treats a as non-linear here, enabling the promotion).

Thus, the goal heap compatibility is $([H]_\omega), H' \bowtie (\Gamma_0 + [\Gamma_1], a : \mathsf{Array}\ A)$ where $([H]_\omega) = ([\emptyset, a \mapsto_r \mathbf{arr}]_\omega) = a \mapsto_\omega \mathbf{arr}$, by Definition b.1.2. This goal heap compatibility judgment is then derived.

$$\frac{H' \bowtie \Gamma_0 + [\Gamma_1]}{H', a \mapsto_\omega \mathbf{arr} \bowtie (\Gamma_0 + [\Gamma_1], a : \mathsf{Array}\ A)}\ \textsc{Ref}$$

Lastly the final goal conjunct specialises to $(\overline{H', a \mapsto_\omega \mathbf{arr}}) \sqsubseteq (\overline{H', a \mapsto_r \mathbf{arr}})$ which holds by reflexivity of the preorder since reference counts are ignored by usage context extraction, so this just becomes $\overline{H'} \sqsubseteq \overline{H'}$.

– t is not a value, and thus there is one possible reduction:

$$\frac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash \mathsf{share}\, t \rightsquigarrow H' \vdash \mathsf{share}\, t' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{share}}$$

with compatibility $H \bowtie \Gamma_0 + \Gamma$. Here, the goal follows straightforwardly from applying the lemma inductively on $\Gamma \vdash t : *A$.

- (clone) with typing:

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\, t_1\, \mathsf{as}\, x\, \mathsf{in}\, t_2 : !B}\ \textsc{clone}$$

Here, there are three possible reductions.

– $t_1 = !v$ for some value v.

Then the typing is as follows:

$$\frac{\dfrac{[\Gamma_1] \vdash v : A}{[\Gamma_1] \vdash !v : !A}\ !_I \qquad \Gamma_2, x : *A \vdash t_2 : !B}{[\Gamma_1] + \Gamma_2 \vdash \mathsf{clone}\, !v\, \mathsf{as}\, x\, \mathsf{in}\, t_2 : !B}\ \textsc{clone}$$

with heap compatibility $H \bowtie \Gamma_0 + [\Gamma_1] + \Gamma_2$.

We can then reduce with:

$$\frac{[\Gamma_1] \vdash v : A \quad \mathsf{dom}(H') \equiv \mathsf{arrRefs}(v) \quad (H'', \theta) \equiv \mathsf{copy}(H')}{H, H' \vdash \mathsf{clone}\, !v\, \mathsf{as}\, x\, \mathsf{in}\, t_2 \rightsquigarrow H, H', H'', x \mapsto_1 ([\Gamma_1] \vdash *(\theta(v)) : *A) \vdash t_2 \mid x : *A \mid \emptyset}\ \rightsquigarrow_{\mathsf{clone}\beta}$$

Here, H' and H'' contain no grading information, as they are made up solely of the array references present in the overall heap. Thus, we only need to consider H. The resulting typing judgment is $\Gamma_2, x : *A \vdash t_2 : !B$ by the second

premise of the initial typing and we construct the goal compatibility judgment as follows, using the incoming compatibility assumption:

$$\frac{H \bowtie \Gamma_0 + [\Gamma_1] + \Gamma_2 \quad [\Gamma_1] \vdash *v : *A \quad x \notin \mathsf{dom}(H)}{H, x{\mapsto_1}([\Gamma_1] \vdash *v : *A) \bowtie (\Gamma_0 + \Gamma_2), x : *A}\ \textsc{Lin}$$

and the grading goal is then given by:

$$\begin{array}{rcl} (\overline{(H, x{\mapsto_1}([\Gamma_1] \vdash *v : *A))} + \emptyset) & \sqsubseteq & (\overline{H}, \overline{x : *A}) \\ \overline{H}, x : 1 & \sqsubseteq & \overline{H}, x : 1 \end{array}$$

which follows by reflexivity.

– $t_1 = !t$ for some t.

Then the typing is as follows:

$$\frac{\dfrac{[\Gamma_1] \vdash t : A}{[\Gamma_1] \vdash !t : !A}\,!_I \qquad \Gamma_2, x : *A \vdash t_2 : !B}{[\Gamma_1] + \Gamma_2 \vdash \mathsf{clone}\ !t\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \textsc{clone}$$

We can reduce:

$$\frac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash \mathsf{clone}\ !t\ \mathsf{as}\ x\ \mathsf{in}\ t_2 \rightsquigarrow H' \vdash \mathsf{clone}\ !t'\ \mathsf{as}\ x\ \mathsf{in}\ t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{clone!}}$$

with compatibility $H \bowtie \Gamma_0 + ([\Gamma_1] + \Gamma_2)$.

Apply this lemma inductively on $[\Gamma_1] \vdash t : A$ with Γ_0 (in the recursive call) instantiated to $\Gamma_0 + \Gamma_2$ which yields $[\Gamma_1'] \vdash t_1' : A$ with H' such that $H' \bowtie ((\Gamma_0 + \Gamma_2) + [\Gamma_1'])$ and $(\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{[\Gamma]})$

The resultant typing for this case is then: $[\Gamma_1'] + \Gamma_2 \vdash \mathsf{clone}\ !t'\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B$ by reapplying typing rule CLONE. The goal compatibility follows then by commutativity and associativity of $+$ $H' \bowtie \Gamma_0 + ([\Gamma_1'] + \Gamma_2)$ and the third goal follows immediately from the inductive evidence.

– Otherwise, there is only one reduction:

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma \mid \Delta}{H \vdash \mathsf{clone}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 \rightsquigarrow H' \vdash \mathsf{clone}\ t_1'\ \mathsf{as}\ x\ \mathsf{in}\ t_2 \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{clone}}$$

with compatibility $H \bowtie \Gamma_0 + ([\Gamma_1] + \Gamma_2)$.

Apply this lemma inductively on $\Gamma_1 \vdash t_1 : !A$ with Γ_0 (in the recursive call) instantiated to $\Gamma_0 + \Gamma_2$ which yields

$[\Gamma_1'] \vdash t_1' : !A$ with H' such that $H' \bowtie ((\Gamma_0 + \Gamma_2) + \Gamma_1')$ and $(\overline{H'} + \Delta) \sqsubseteq (\overline{H}, \overline{\Gamma_1})$ (which subsequently provides the third goal immediately) with the outgoing typing $[\Gamma_1'] + \Gamma_2 \vdash \mathsf{clone}\, t_1 \,\mathsf{as}\, x \,\mathsf{in}\, t_2 : !B$ and compatibility from the inductive evidence via commutativity and associativity of $+$.

- (nec) with typing:

$$\frac{\varnothing \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

The only possible reduction is

$$\frac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash *t \rightsquigarrow H' \vdash *t' \mid \Gamma \mid \Delta}\ \rightsquigarrow_*$$

then by induction we get $\varnothing \vdash t' : A$ from which we can rebuild the typing $[\Gamma] \vdash *t' : *A$ and the results of the induction give the goal here as the heap and output contexts and unchanged.

- (newArray), (readArray), (writeArray), (deleteArray) have no reduction as they are unapplied primitives.

□

B.4 UNIQUENESS PROOF

Lemma b.4.1 (Single-step uniqueness). *For a well-typed term $\Gamma \vdash t_1 : *A$ and all Γ_0 and heaps H such that $H \bowtie (\Gamma_0 + \Gamma)$ and given a single-step reduction $H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma' \mid \Delta$ then for all $a \in \mathsf{arrRefs}(t_1')$ (array references in t_1') we have:*

$$\begin{aligned} a \mapsto_1 t' \in H &\implies \exists t''. a \mapsto_1 t'' \in H' \\ \wedge\ a \notin \mathsf{dom}(H) &\implies \exists t''. a \mapsto_1 t'' \in H' \end{aligned}$$

i.e., any array references contributing to the final term that in the incoming heap start unique stay unique, and any new array references contributing to the final term are unique.

Proof. By induction on typing.

- (var)

$$\frac{}{[\Gamma], x : *A \vdash x : *A}\ \textsc{var}$$

Then we also have a heap H such that $H \bowtie ([\Gamma], x : *A)$.

There are two possibilities:

– By inversion of heap-compatibility implies that there exists a subheap H_1 such that $H = H_1, x \mapsto_1 (\Gamma' \vdash t' : *A)$.

$$\frac{H_1 \bowtie ([\Gamma] + \Gamma') \qquad \Gamma' \vdash t' : *A \qquad x \notin \mathsf{dom}(H_1)}{(H_1, x \mapsto_1 (\Gamma' \vdash t' : *A)) \bowtie ([\Gamma], x : *A)}\ \textsc{Lin}$$

and reduction:

$$\frac{}{H_1, x \mapsto_1 t' \vdash x \rightsquigarrow H_1 \vdash t' \mid \emptyset \mid x : 1}\ \rightsquigarrow_{\textsc{var1}}$$

Then we have that $a \in \mathsf{arrRefs}(t')$ we have:

$$(a \mapsto_1 t_2 \in H_1, x \mapsto_1 t' \implies \exists t_3 . a \mapsto_1 t_3 \in H_1)$$

trivially (with $t_3 = t_2$) as no array references were modified or manipulated here (just change in variables), and

$$(a \notin \mathsf{dom}(H_1)) \implies \exists t_3 . a \mapsto_1 t_3 \in H_1$$

trivially since the premise must always be false.

– By inversion of heap-compatibility implies that there exists a subheap H_1 such that $H = H_1, x \mapsto_\omega (\Gamma' \vdash t' : *A)$.

$$\frac{H_1 \bowtie ([\Gamma] + \Gamma') \qquad \Gamma' \vdash t' : *A \qquad x \notin \mathsf{dom}(H_1)}{(H_1, x \mapsto_\omega (\Gamma' \vdash t' : *A)) \bowtie ([\Gamma], x : *A)}\ \textsc{Lin}$$

and reduction:

$$\frac{}{H_1, x \mapsto_\omega t' \vdash x \rightsquigarrow H_1, x \mapsto_\omega t' \vdash t' \mid \emptyset \mid x : 1}\ \rightsquigarrow_{\textsc{var1}}$$

Then we have that $a \in \mathsf{arrRefs}(t')$ we have:

$$\begin{aligned}&(a \in \mathsf{dom}(H_1, x \mapsto_\omega t')) \implies (a \mapsto_1 t_2 \in H_1 \implies \exists t_3 . a \mapsto_1 t_3 \in H_1, x \mapsto_\omega t') \\ \wedge\ &(a \notin \mathsf{dom}(H_1, x \mapsto_\omega t')) \implies \exists t_3 . a \mapsto_1 t_3 \in H_1, x \mapsto_\omega t'\end{aligned}$$

which both follow trivially the first as essentially the identity proof as no array objects are manipulated and the second as the premise is always false.

- (abs) Trivial as it doesn't match the typing

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap *B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : *B}\ \textsc{app}$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and two reductions (beta and application congruence on the left) then several possible reductions following from primitive applications:

1.
$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma_1 \mid \Delta_1}{H \vdash t_1\, t_2 \rightsquigarrow H' \vdash t_1'\, t_2 \mid \Gamma_1 \mid \Delta_1}\ \rightsquigarrow_{\text{APP}}$$

Then the goal follows by induction.

2. Alternatively $t_1 = \lambda x.t_1'$ such that typing is:

$$\frac{\dfrac{\Gamma_1, x : A \vdash t_1' : *B}{\Gamma_1 \vdash t_1' : A \multimap *B}\ \text{ABS} \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash (\lambda x.t_1')\, t_2 : *B}\ \text{APP}$$

and we have reduction:

$$\frac{\Gamma_2 \vdash t_2 : A}{H \vdash (\lambda x.t_1')\, t_2 \rightsquigarrow H, x \mapsto_1 (\Gamma_2 \vdash t_2 : A) \vdash t_1' \mid x : A \mid \emptyset}\ \rightsquigarrow_{\beta}$$

Then for all $a \in \mathsf{arrRefs}(t_1'')$ we have that:

$$a \mapsto_1 t' \in H \implies \exists t''.a \mapsto_1 t'' \in H, x \mapsto_1 t_2$$

trivially as the right hand side does not refer to array references and the heap is preserved, and:

$$a \notin \mathsf{dom}(H) \implies \exists t''.a \mapsto_1 t'' \in H'$$

is trivially true as the antecedent is always false.

3. $t_1 = \mathsf{newArray}$ therefore $A = \mathbb{N}$

Therefore we induct on the second argument:

- t_2 is a value and therefore by the value lemma (Lemma 10) $t_2 = n$ and thus the typing is:

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{newArray}\, n : *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{new}$$

with $H \bowtie (\Gamma_0 + \Gamma)$.

Thus there is a reduction as follows:

$$\frac{a \# H}{H \vdash \mathsf{newArray}\, n \rightsquigarrow H, a \mapsto_1 \mathbf{arr} \vdash *a \mid \emptyset \mid \emptyset}\ \rightsquigarrow_{\mathsf{newArray}}$$

where $a \notin \mathsf{dom}(H)$ but we have that $\exists t''.a \mapsto_1 t'' \in H, a \mapsto_1 \mathbf{arr}$ satisfying the goal.

- t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{newArray}\, t_2 \rightsquigarrow H' \vdash \mathsf{newArray}\, t_2' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{prim}}$$

Then the result holds by induction

4. $t_1 = \mathsf{readArray}$ therefore $A = *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})$

   and there is a reduction:

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{readArray}\, t_2 \rightsquigarrow H' \vdash \mathsf{readArray}\, t_2' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{prim}}$$

   Therefore the single-step uniqueness result holds by induction

5. $t_1 = \mathsf{readArray}\,(*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})$

   - t_2 is a value and therefore by the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{N}$ (Lemma 10) implies $t_2 = n$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \mathsf{Array}\ \mathbb{F} \vdash a : (\mathsf{Array}\ \mathbb{F})\ \mathsf{arr}}{[\Gamma_1], a : \mathsf{Array}\ \mathbb{F} \vdash *a : *(\mathsf{Array}\ \mathbb{F})}\ *\mathsf{arr} \qquad \Gamma_2 \vdash i : \mathbb{N}}{[\Gamma_1] + \Gamma_2, a : \mathsf{Array}\ \mathbb{F} \vdash \mathsf{readArray}\, a\, i : \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{read}$$

     with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2, a : \mathsf{Array}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_r \mathbf{arr}$.

     Then there is a reduction as follows:

$$\frac{}{H, a \mapsto_r (\mathbf{arr}[i] = v) \vdash \mathsf{readArray}\,(*a)\, i \rightsquigarrow H, a \mapsto_r (\mathbf{arr}[i] = v) \vdash (v, *a) \mid \emptyset \mid \emptyset}\ \rightsquigarrow_{\mathsf{readArray}}$$

     where $a \in \mathsf{dom}(H)$ and thus if that $\exists t''.a \mapsto_1 t'' \in H, a \mapsto_r \mathbf{arr}$ i.e. $r = 1$ then we have $\exists t''.a \mapsto_r t'' \in H, a \mapsto_1 \mathbf{arr}$ for the output heap satisfying the goal for this array reference for all other $a' \in \mathsf{arrRefs}(t)$ we have:

$$\begin{aligned} a' \mapsto_1 t' \in H, a \mapsto_r (\mathbf{arr}[i] = t) &\implies \exists t''.a' \mapsto_1 t'' \in H, a \mapsto_r (\mathbf{arr}[i] = t) \\ \wedge\ a' \notin \mathsf{dom}(H, a \mapsto_r (\mathbf{arr}[i] = t)) &\implies \exists t''.a' \mapsto_1 t'' \in H, a \mapsto_r (\mathbf{arr}[i] = t) \end{aligned}$$

     follows trivially since the rest of the heap is preserved.

   - t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{readArray}\,(*a)\, t_2 \rightsquigarrow H' \vdash \mathsf{readArray}\,(*a)\, t_2' \mid \Gamma \mid \Delta}\ \rightsquigarrow_{\mathsf{prim}}$$

     And therefore the single-step uniqueness result holds by induction.

6. $t_1 = \mathsf{writeArray}$ therefore $A = *(\mathsf{Array}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F})$

with reduction

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{writeArray}\, t_2 \rightsquigarrow H' \vdash \mathsf{writeArray}\, t_2' \mid \Gamma \mid \Delta}\ {}^{\rightsquigarrow}\mathsf{prim}$$

Therefore the single-step uniqueness result holds by induction.

7. $t_1 = \mathsf{writeArray}\,(*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}\ \mathbb{F})$

   with reduction:

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{writeArray}\,(*a)\, t_2 \rightsquigarrow H' \vdash \mathsf{writeArray}\,(*a)\, t_2' \mid \Gamma \mid \Delta}\ {}^{\rightsquigarrow}\mathsf{prim}$$

   Therefore the single-step uniqueness result holds by induction.

8. $t_1 = \mathsf{writeArray}\,(*a)\, i$ therefore $A = \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F})$

   – t_2 is a value and therefore by the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{F}$ (Lemma 10) implies $t_2 = f$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \mathsf{Array}\ \mathbb{F} \vdash a : (\mathsf{Array}\ \mathbb{F})\ \mathsf{arr}}{[\Gamma_1], a : \mathsf{Array}\ \mathbb{F} \vdash *a : *(\mathsf{Array}\ \mathbb{F})}\ {}^{*}\mathsf{arr} \qquad \Gamma_2 \vdash i : \mathbb{N} \quad \Gamma_3 \vdash f : \mathbb{F}}{[\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}\ \mathbb{F} \vdash \mathsf{writeArray}\, a\, i\, f : *(\mathsf{Array}\ \mathbb{F})}\ \mathsf{write}$$

   with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_r \mathbf{arr}$.

   Then there is a reduction as follows:

$$\frac{}{H, a \mapsto_r \mathbf{arr} \vdash \mathsf{writeArray}\,(*a)\, i\, v \rightsquigarrow H, a \mapsto_r (\mathbf{arr}[i] = v) \vdash *a \mid \emptyset \mid \emptyset}\ {}^{\rightsquigarrow}\mathsf{writeArray}$$

   thus for all $a' \in \mathsf{arrRefs}(v)$ (array references in v) we have:

$$a' \mapsto_1 t' \in H, a \mapsto_r \mathbf{arr} \implies \exists t''. a' \mapsto_1 t'' \in H, a \mapsto_r (\mathbf{arr}[i] = v)$$
$$\wedge\ a' \notin \mathsf{dom}(H, a \mapsto_r \mathbf{arr}) \implies \exists t''. a' \mapsto_1 t'' \in H, a \mapsto_r (\mathbf{arr}[i] = v)$$

   where the first conjunct follows trivially as the only a' is a and if $r = 1$ the result is direct (identity) otherwise the premise is false. The second conjunct is trivially since the premise as we have the only a' here as a so the premise is always false.

   – t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma \mid \Delta}{H \vdash \mathsf{writeArray}\,(*a)\, i\, t_2 \rightsquigarrow H' \vdash \mathsf{writeArray}\,(*a)\, i\, t_2' \mid \Gamma \mid \Delta}\ {}^{\rightsquigarrow}\mathsf{prim}$$

Therefore the single-step uniqueness result holds by induction.

9. $t_1 = \mathsf{deleteArray}$ therefore $A = *(\mathsf{Array}\ \mathbb{F}) \multimap 1$

   But the typing does not match the lemma as the result type is therefore 1.

- (derelict)

$$\frac{\Gamma, x : A \vdash t : *B}{\Gamma, x : [A] \vdash t : *B}\ \textsc{der}$$

and a single-reduction $H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta$ and we have heap compatibility $H \bowtie (\Gamma_0 + \Gamma, x : [A])$ which by inversion means there exists a heap H_1 such that $H = H_1, x \mapsto_\omega (\Gamma' \vdash t' : A)$.

$$\frac{H_1 \bowtie (\Gamma + [\Gamma']) \qquad \Gamma' \vdash t' : *A \qquad x \notin \mathsf{dom}(H_1)}{H_1, x \mapsto_\omega (\Gamma' \vdash t' : *A) \bowtie (\Gamma, x : [A])}\ \omega$$

We can thus rebuild the heap compatibility from these premises for the typing premise:

$$\frac{H_1' \bowtie (\Gamma + [\Gamma']) \qquad \Gamma' \vdash t' : *A \qquad x \notin \mathsf{dom}(H_1)}{H_1', x \mapsto_\omega (\Gamma' \vdash t' : *A) \bowtie (\Gamma, x : A)}\ \textsc{Lin}$$

On which we can induct with $\Gamma, x : A \vdash t : *B$ and the same reduction $H \vdash t \rightsquigarrow H' \vdash *v \mid \Gamma \mid \Delta$ to get that for all $a \in \mathsf{arrRefs}(v)$ (array references in v) we have:

$$\begin{aligned} a \mapsto_1 t' \in H &\implies \exists t''. a \mapsto_1 t'' \in H' \\ \wedge\ a \notin \mathsf{dom}(H) &\implies \exists t''. a \mapsto_1 t'' \in H' \end{aligned}$$

which satisfies the goal here.

- (bangIntro)

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash\ !t : !A}\ !_I$$

Trivially satisfies the theorem since the typing cannot conclude with a type of form $*A$.

- (bangElim)

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : *B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ !x = t_1\ \mathsf{in}\ t_2 : *B}\ !_E$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash \mathsf{let}\ !x = t_1\ \mathsf{in}\ t_2 \rightsquigarrow H' \vdash t' \mid \Gamma' \mid \Delta$$

which has two possible derivations:

1.

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma' \mid \Delta_1}{H \vdash \mathsf{let}\,!x = t_1\,\mathsf{in}\,t_2 \rightsquigarrow H' \vdash \mathsf{let}\,!x = t_1'\,\mathsf{in}\,t_2 \mid \Gamma' \mid \Delta_1}\ \rightsquigarrow_{\text{LET!}}$$

Then induction provides the goal.

2. Alternatively $t_1 = !t_1'$ such that the typing is:

$$\frac{\dfrac{[\Gamma_1] \vdash t_1' : A}{[\Gamma_1] \vdash !t_1' : !A}\ !_I \qquad \Gamma_2, x : [A] \vdash t_2 : *B}{[\Gamma_1] + \Gamma_2 \vdash \mathsf{let}\,!x = !t_1'\,\mathsf{in}\,t_2 : *B}\ !_E$$

and we have reduction:

$$\frac{[\Gamma_1] \vdash t_1' : A}{H \vdash \mathsf{let}\,!x = !t_1'\,\mathsf{in}\,t_2 \rightsquigarrow H, x{\mapsto_\omega}([\Gamma_1] \vdash t_1' : A) \vdash t_2 \mid x : [A] \mid \emptyset}\ \rightsquigarrow_{!\beta}$$

Which trivially satisfies the goal since all array references are then in H and we get the conditions trivially (since no array references are manipulated) similar to the β proof above.

- (pairIntro) Trivially satisfies the theorem since the typing cannot conclude with a type of form $*A$.

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : *C}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,(x, y) = t_1\,\mathsf{in}\,t_2 : *C}\ \otimes_E$$

Two possible reductions:

1.

$$\frac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma' \mid \Delta}{H \vdash \mathsf{let}\,(x, y) = t_1\,\mathsf{in}\,t_2 \rightsquigarrow H' \vdash \mathsf{let}\,(x, y) = t_1'\,\mathsf{in}\,t_2 \mid \Gamma' \mid \Delta}\ \rightsquigarrow_{\otimes\beta}$$

and induction provides the goal.

2. otherwise $t_1 = (t_3, t_4)$ such that the typing is:

$$\frac{\dfrac{\Gamma_3 \vdash t_3 : A \quad \Gamma_4 \vdash t_4 : B}{\Gamma_3 + \Gamma_4 \vdash (t_3, t_4) : (A \otimes B)}\ \otimes_I \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_3 + \Gamma_4 + \Gamma_2 \vdash \mathsf{let}\,(x, y) = (t_3, t_4)\,\mathsf{in}\,t_2 : C}\ \otimes_E$$

and we have reduction:

$$\frac{\Gamma_1 \vdash t_1' : A \quad \Gamma_2 \vdash t_1'' : B}{H \vdash \mathsf{let}\,(x, y) = (t_1', t_1'')\,\mathsf{in}\,t_3 \rightsquigarrow H, x{\mapsto_1}(\Gamma_1 \vdash t_1' : A), y{\mapsto_1}(\Gamma_2 \vdash t_1'' : B) \vdash t_3 \mid \emptyset, x : A, y : B \mid \emptyset}\ \rightsquigarrow_{\text{LET}\otimes}$$

The goal is then straightforward: for all $a \in \mathsf{arrRefs}(t_3)$ we have:

$$\begin{aligned} a{\mapsto_1}t' \in H &\implies \exists t''.a{\mapsto_1}t'' \in H, x{\mapsto_1}(\Gamma_1 \vdash t_1' : A), y{\mapsto_1}(\Gamma_2 \vdash t_1'' : B) \\ \wedge\ a \notin \mathsf{dom}(H) &\implies \exists t''.a{\mapsto_1}t'' \in H, x{\mapsto_1}(\Gamma_1 \vdash t_1' : A), y{\mapsto_1}(\Gamma_2 \vdash t_1'' : B) \end{aligned}$$

where the first conjunct holds trivially since there is no manipulation of the array references here and the second conjunct is always true.

- (unitIntro)

  Trivially satisfies the theorem since the typing cannot conclude with a type of form $*A$.

- (unitElim) Following essentially the same structure as the tensor proof where array reference counting is not used so induction provides the goal.

- (return), (bind) All trivially satisfy the theorem since the typing cannot conclude with a type of form $*A$.

- (nec)

  $$\frac{\varnothing \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

  with reduction

  $$\frac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash *t \rightsquigarrow H' \vdash *t' \mid \Gamma \mid \Delta}\ \rightsquigarrow_*$$

  Then by induction on the premise term we get the required result here since the output heap is preserved between the two.

- (newArray) (readArray) (writeArray) (deleteArray)

  All trivial as they don't match the typing and don't reduce.

□

Theorem 5 (Uniqueness). *For a well-typed term $\Gamma \vdash t : *A$ and all Γ_0 and H such that $H \bowtie (\Gamma_0 + \Gamma)$ and given a multi-reduction to a value $H \vdash t \Rightarrow H' \vdash *v \mid \Gamma' \mid \Delta$, for all $a \in \mathsf{arrRefs}(v)$ (array references in v) we have:*

$$\begin{aligned} a{\mapsto_1}t' \in H &\implies \exists t''.a{\mapsto_1}t'' \in H' \\ \wedge\ a \notin \mathsf{dom}(H) &\implies \exists t''.a{\mapsto_1}t'' \in H' \end{aligned}$$

i.e., any array references contributing to the final term that are unique in the incoming heap stay unique in the resulting term, and any new array references contributing to the final term are also unique.

Proof. By induction on typing $\Gamma \vdash t : *A$.

- (var)

$$\frac{}{[\Gamma], x : *A \vdash x : *A}\ \textsc{var}$$

Then we also have a heap H such that $H \bowtie ([\Gamma], x : *A)$ which by inversion of heap-compatibility implies that there exists a subheap H_1 such that $H = H_1, x \mapsto_1 (\Gamma' \vdash t' : *A)$.

$$\frac{H_1 \bowtie ([\Gamma] + \Gamma') \qquad \Gamma' \vdash t' : *A \qquad x \notin \mathsf{dom}(H_1)}{(H_1, x \mapsto_1 (\Gamma' \vdash t' : *A)) \bowtie ([\Gamma], x : *A)}\ \textsc{Lin}$$

and reduction:

$$H \vdash x \ \Rightarrow\ H' \vdash *v \mid \Gamma' \mid \Delta$$

which necessarily must be derived by following since there is only one possible (single) reduction for variables:

$$\frac{\overline{H_1, x \mapsto_1 t' \vdash x \rightsquigarrow H_1 \vdash t' \mid \emptyset \mid x : 1}^{\ \rightsquigarrow_{\textsc{var1}}} \qquad H_1 \vdash t' \ \Rightarrow\ H' \vdash *v \mid \Gamma_2 \mid \Delta_2}{H_1, x \mapsto_1 t' \vdash x \ \Rightarrow\ H'' \vdash *v \mid \Gamma_2 \mid x : 1 + \Delta_2}\ \textsc{ext}$$

Then by induction on the second premise with $H_1 \bowtie ([\Gamma] + \Gamma')$ (i.e., instantiating Γ_0 to $[\Gamma]$) $\Gamma' \vdash t' : *A$ we have that $a \in \mathsf{arrRefs}(v)$ (array references in v) we have:

$$\begin{aligned}&(a \in \mathsf{dom}(H_1, x \mapsto_1 t')) \implies (a \mapsto_1 t'' \in H_1 \implies \exists t'''. a \mapsto_1 t'' \in H') \\ \wedge\ &(a \notin \mathsf{dom}(H_1, x \mapsto_1 t'')) \implies \exists t'''. a \mapsto_1 t'' \in H'\end{aligned}$$

and since $H = H_1, x \mapsto_1 t'$ then the above satisfies the goal here too.

The case for the reduction $H_1, x \mapsto_\omega t' \vdash x \rightsquigarrow H_1, x \mapsto_\omega t' \vdash t' \mid \emptyset \mid x : 1$ is the same.

- (abs) Is trivial since the typing cannot conclude with a type of the form $*A$.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap *B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : *B}\ \textsc{app}$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash t_1\, t_2 \ \Rightarrow\ H' \vdash *v \mid \Gamma' \mid \Delta$$

which has two possible derivations on terms:

1.

$$\frac{\dfrac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma_1'' \mid \Delta_1}{H \vdash t_1\, t_2 \rightsquigarrow H' \vdash t_1'\, t_2 \mid \Gamma_1'' \mid \Delta_1}\rightsquigarrow_{\text{APP}} \quad H' \vdash t_1'\, t_2 \Rightarrow H'' \vdash *v \mid \Gamma_2'' \mid \Delta_2}{H \vdash t_1\, t_2 \Rightarrow H'' \vdash *v \mid \Gamma_1'' + \Gamma_2'' \mid \Delta_1 + \Delta_2}\,\text{EXT}$$

By conservation on the single reduction for t_1 we get type preservation which gives us $\Gamma_1' \vdash t_1' : A \multimap *B$ and thus $\Gamma_1' + \Gamma_2 \vdash t_1'\, t_2 : *B$, and also that $H' \bowtie (\Gamma_0 + \Gamma_1' + \Gamma_2)$. Thus we can induct on the multi-reduction here to get that $a \in \mathsf{arrRefs}(v)$ then:

$$\begin{aligned}&(a \mapsto_1 t' \in H' \implies \exists t''. a \mapsto_1 t'' \in H'')\\ \wedge\, &(a \notin \mathsf{dom}(H')) \implies \exists t''. a \mapsto_1 t'' \in H''\end{aligned}$$

which provides the goal.

2. Alternatively $t_1 = \lambda x. t_1'$ such that typing is:

$$\frac{\dfrac{\Gamma_1, x : A \vdash t_1' : *B}{\Gamma_1 \vdash t_1' : A \multimap *B}\,\text{ABS} \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash (\lambda x. t_1')\, t_2 : *B}\,\text{APP}$$

and we have reduction:

$$\frac{\dfrac{\Gamma_2 \vdash t_2 : A}{H \vdash (\lambda x. t_1')\, t_2 \rightsquigarrow H, x \mapsto_1 (\Gamma_2 \vdash t_2 : A) \vdash t_1' \mid x : A \mid \emptyset}\rightsquigarrow_{\beta} \quad H, x \mapsto_1 (\Gamma_2 \vdash t_2 : A) \vdash t_1' \Rightarrow H'' \vdash *v \mid \Gamma_2 \mid \Delta_2}{H \vdash (\lambda x. t_1')\, t_2 \Rightarrow H'' \vdash *v \mid \Gamma_2, x : A \mid \Delta_2}\,\text{EXT}$$

From the incoming heap compatibility $H \bowtie (\Gamma_1 + \Gamma_2)$ we can construct the following heap compatibility:

$$\frac{H \bowtie (\Gamma_1 + \Gamma_2) \qquad \Gamma_2 \vdash t_2 : A \qquad x \notin \mathsf{dom}(H)}{H, x \mapsto_1 (\Gamma_2 \vdash t_2 : A) \bowtie (\Gamma_1, x : A)}\,\textsc{Lin}$$

and thus we can induct on the second multi reduction with this heap compatibility and the typing $\Gamma_1, x : A \vdash t_1' : *B$ to get that $x \in \mathsf{arrRefs}(v)$ then: $x \mapsto_1 t' \in H''$ which satisfies the goal here.

3. Alternatively $t_1 = p$ and then there are two classes of possible reduction: congruence on a partial primitive application (dealt with first) or primitive beta rules

   a) $t_1 = p$ and t_2 is not a value therefore the reduction is:

$$\frac{\dfrac{H \vdash t_2 \rightsquigarrow H' \vdash t_2' \mid \Gamma_1'' \mid \Delta_1}{H \vdash p\, t_2 \rightsquigarrow H' \vdash p\, t_2' \mid \Gamma_1'' \mid \Delta_1}\rightsquigarrow_{\mathsf{prim}} \quad H' \vdash p\, t_2' \Rightarrow H'' \vdash *v \mid \Gamma_2'' \mid \Delta_2}{H \vdash p\, t_2 \Rightarrow H'' \vdash *v \mid \Gamma_1'' + \Gamma_2 \mid \Delta_1 + \Delta_2}\,\text{EXT}$$

By single-step uniqueness on the first premise we have:

$$\begin{aligned} & a{\mapsto_1}t' \in H \implies \exists t''.a{\mapsto_1}t'' \in H' \\ \wedge \quad & a \notin \mathsf{dom}(H) \implies \exists t''.a{\mapsto_1}t'' \in H' \end{aligned}$$

By conservation on the first premise we get that $\Gamma_1 + \Gamma_2' \vdash p\, t_2 : *B$ and $H' \bowtie (\Gamma_0 + \Gamma_1 + \Gamma_2')$ which then enables induction of multi-step uniqueness on the tail multi-reduction to get:

$$\begin{aligned} & a{\mapsto_1}t' \in H' \implies \exists t''.a{\mapsto_1}t'' \in H'' \\ \wedge \quad & a \notin \mathsf{dom}(H') \implies \exists t''.a{\mapsto_1}t'' \in H'' \end{aligned}$$

which composes with the single-step uniqueness (transitivity of implication) to give the goal here.

b) $t_1 = \mathsf{newArray}$ and $t_2 = n$

$$\frac{\cdots}{\Gamma \vdash \mathsf{newArray}\, n : *(\mathsf{Array}\ \mathbb{F})}\ \textsc{newArray}$$

and given $H \bowtie \Gamma_0 + \Gamma$ then there is reduction:

$$\frac{\dfrac{a\#H}{H \vdash \mathsf{newArray}\, n \rightsquigarrow H, a{\mapsto_1}\mathbf{arr} \vdash *a \mid \emptyset \mid \emptyset}\ \rightsquigarrow_{\mathsf{newArray}} \quad \dfrac{}{H, a{\mapsto_1}\mathbf{arr} \vdash *a \rightsquigarrow H, a{\mapsto_1}\mathbf{arr} \vdash *a \mid \emptyset \mid \emptyset}\ \textsc{refl}}{H \vdash \mathsf{newArray}\, n \Rightarrow H, a{\mapsto_1}\mathbf{arr} \vdash *a \mid \emptyset \mid \emptyset}\ \textsc{ext}$$

where $a \notin \mathsf{dom}(H)$ but we have that $\exists t''.a{\mapsto_1}t'' \in H, a{\mapsto_1}\mathbf{arr}$ satisfying the goal.

c) $t_1 = \mathsf{readArray}\ (*a)$ and $t_2 = n$

$$\frac{\cdots}{\Gamma \vdash \mathsf{readArray}\ (*a)\, n : *(\mathsf{Array}\ \mathbb{F})}\ \textsc{readArray}$$

and given $H \bowtie \Gamma_0 + \Gamma$ then there is reduction:

$$\frac{\dfrac{}{H, a{\mapsto_r}(\mathbf{arr}[i] = v) \vdash \mathsf{readArray}\ (*a)\, i \rightsquigarrow H, a{\mapsto_r}(\mathbf{arr}[i] = v) \vdash (v, *a) \mid \emptyset \mid \emptyset}\ \rightsquigarrow_{\mathsf{readArray}} \quad \textit{reflexive...}}{H, a{\mapsto_r}(\mathbf{arr}[i] = t) \vdash \mathsf{readArray}\ (*a)\, n \Rightarrow H' \vdash (v, *a) \mid \Gamma_2 \mid \Delta_2}\ \textsc{ext}$$

Then applying Lemma b.4.1 we have:

$$\begin{aligned} & a'{\mapsto_1}t' \in H, a{\mapsto_r}(\mathbf{arr}[i] = t) \implies \exists t''.a'{\mapsto_1}t'' \in H, a{\mapsto_r}(\mathbf{arr}[i] = t) \\ \wedge \quad & a \notin \mathsf{dom}(H, a{\mapsto_r}(\mathbf{arr}[i] = t)) \implies \exists t''.a'{\mapsto_1}t'' \in H, a{\mapsto_r}(\mathbf{arr}[i] = t) \end{aligned}$$

and by induction on the second premise we have for all $a \in \mathsf{arrRefs}(v)$:

$$\begin{aligned} & a'{\mapsto_1}t' \in H, a{\mapsto_r}(\mathbf{arr}[i] = t) \implies \exists t''.a'{\mapsto_1}t'' \in H' \\ \wedge \quad & a' \notin \mathsf{dom}(H, a{\mapsto_r}(\mathbf{arr}[i] = t)) \implies \exists t''.a'{\mapsto_1}t'' \in H' \end{aligned}$$

Composing these two pieces of evidence together by instantiation and transitivity provides the goal.

d) $t_1 = \mathsf{writeArray}\ (*a)\ n$ and $t_2 = f$

$$\frac{\cdots}{\Gamma \vdash \mathsf{writeArray}\ (*a)\ n\ v : *(\mathsf{Array}\ \mathbb{F})}\ \textsc{writeArray}$$

and given $H \bowtie \Gamma_0 + \Gamma$ then there is reduction:

$$\frac{\dfrac{}{H, a{\mapsto_r}\mathbf{arr} \vdash \mathsf{writeArray}\ (*a)\ i\ v \rightsquigarrow H, a{\mapsto_r}(\mathbf{arr}[i] = v) \vdash *a \mid \emptyset \mid \emptyset}\ {\rightsquigarrow}\mathsf{writeArray} \qquad \dfrac{}{\cdots}\ \textsc{refl}}{H \vdash \mathsf{writeArray}\ (*a)\ n\ v \Rightarrow H, a{\mapsto_r}(\mathbf{arr}[i] = v) \vdash *a \mid \emptyset \mid \emptyset}\ \textsc{ext}$$

where the goal follows by Lemma b.4.1 on the first premise as the second premise is necessarily just the reflexive multi-reduction rule.

- (derelict)

$$\frac{\Gamma, x : A \vdash t : *B}{\Gamma, x : [A] \vdash t : *B}\ \textsc{der}$$

and a multi-reduction $H \vdash t \Rightarrow H' \vdash *v \mid \Gamma \mid \Delta$ and we have heap compatibility $H \bowtie (\Gamma_0 + \Gamma, x : [A])$ which by inversion means there exists a heap H_1 such that $H = H_1, x{\mapsto_\omega}(\Gamma' \vdash t' : A)$.

$$\frac{H_1 \bowtie (\Gamma + [\Gamma']) \qquad \Gamma' \vdash t' : *A \qquad x \notin \mathsf{dom}(H_1)}{H_1, x{\mapsto_\omega}(\Gamma' \vdash t' : *A) \bowtie (\Gamma, x : [A])}\ \omega$$

We can thus rebuild the heap compatibility from these premises for the typing premise:

$$\frac{H_1' \bowtie (\Gamma + [\Gamma']) \qquad \Gamma' \vdash t' : *A \qquad x \notin \mathsf{dom}(H_1)}{H_1', x{\mapsto_\omega}(\Gamma' \vdash t' : *A) \bowtie (\Gamma, x : A)}\ \textsc{Lin}$$

On which we can induct with $\Gamma, x : A \vdash t : *B$ and the multi-reduction $H \vdash t \Rightarrow H' \vdash *v \mid \Gamma \mid \Delta$ to get that for all $a \in \mathsf{arrRefs}(v)$ (array references in v) we have:

$$\begin{aligned} a{\mapsto_1}t' \in H &\implies \exists t''.a{\mapsto_1}t'' \in H' \\ \wedge\ a \notin \mathsf{dom}(H) &\implies \exists t''.a{\mapsto_1}t'' \in H' \end{aligned}$$

which satisfies the goal here.

- (bangIntro)

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash !t : !A}\ !_I$$

Trivially satisfies the theorem since the typing cannot conclude with a type of form $*A$.

- (bangElim)

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : *B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\, !x = t_1 \,\mathsf{in}\, t_2 : *B}\, !_E$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash \mathsf{let}\, !x = t_1 \,\mathsf{in}\, t_2 \;\Rightarrow\; H' \vdash *v \mid \Gamma' \mid \Delta$$

which has two possible derivations:

1.

$$\frac{\dfrac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma_1 \mid \Delta_1}{H \vdash \mathsf{let}\, !x = t_1 \,\mathsf{in}\, t_2 \rightsquigarrow H' \vdash \mathsf{let}\, !x = t_1' \,\mathsf{in}\, t_2 \mid \Gamma_1 \mid \Delta_1}\, \rightsquigarrow_{\text{APP}} \qquad H' \vdash \mathsf{let}\, !x = t_1' \,\mathsf{in}\, t_2 \;\Rightarrow\; H'' \vdash *v \mid \Gamma_2 \mid \Delta_2}{H \vdash \mathsf{let}\, !x = t_1 \,\mathsf{in}\, t_2 \;\Rightarrow\; H'' \vdash *v \mid \Gamma_1 + \Gamma_2 \mid \Delta_1 + \Delta_2}\, \text{EXT}$$

By conservation on the single reduction for t_1 we get type preservation which gives us $\Gamma_1 \vdash t_1' : !A$ and thus $\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\, !x = t_1' \,\mathsf{in}\, t_2 : *B$, and also that $H' \bowtie (\Gamma_0 + \Gamma')$. Thus we can induct on the multi-reduction here to get

$$\begin{aligned} a \mapsto_1 t' \in H' &\implies \exists t''. a \mapsto_1 t'' \in H'' \\ \wedge\; a \notin \mathsf{dom}(H') &\implies \exists t''. a \mapsto_1 t'' \in H'' \end{aligned}$$

which along with Lemma b.4.1 applied to the single reduction here gives us:

$$\begin{aligned} a \mapsto_1 t' \in H &\implies \exists t''. a \mapsto_1 t'' \in H' \\ \wedge\; a \notin \mathsf{dom}(H) &\implies \exists t''. a \mapsto_1 t'' \in H' \end{aligned}$$

which we can combine together via instantiation and transitivity to get the goal:

$$\begin{aligned} a \mapsto_1 t' \in H &\implies \exists t''. a \mapsto_1 t'' \in H'' \\ \wedge\; a \notin \mathsf{dom}(H) &\implies \exists t''. a \mapsto_1 t'' \in H'' \end{aligned}$$

2. Alternatively $t_1 = !t_1'$ such that the typing is:

$$\frac{\dfrac{[\Gamma_1] \vdash t_1' : A}{[\Gamma_1] \vdash !t_1' : !A}\, !_I \qquad \Gamma_2, x : [A] \vdash t_2 : *B}{[\Gamma_1] + \Gamma_2 \vdash \mathsf{let}\, !x = !t_1' \,\mathsf{in}\, t_2 : *B}\, !_E$$

and we have reduction:

$$\frac{\dfrac{[\Gamma_1] \vdash t_1' : A}{H \vdash \mathsf{let}\, !x = !t_1' \,\mathsf{in}\, t_2 \rightsquigarrow H, x \mapsto_\omega ([\Gamma_1] \vdash t_1' : A) \vdash t_2 \mid x : [A] \mid \emptyset}\, \rightsquigarrow_{!\beta} \qquad H, x \mapsto_\omega ([\Gamma_1] \vdash t_1' : A) \vdash t_2 \;\Rightarrow\; H'' \vdash *v \mid \Gamma_2 \mid \Delta_2}{H \vdash \mathsf{let}\, !x = !t_1' \,\mathsf{in}\, t_2 \;\Rightarrow\; H'' \vdash *v \mid \Gamma_2, x : [A] \mid \Delta_2}\, \text{EXT}$$

From the incoming heap compatibility $H \bowtie ([\Gamma_1] + \Gamma_2)$ we can construct the following heap compatibility:

$$\frac{H \bowtie ([\Gamma_1] + \Gamma_2) \qquad [\Gamma_1] \vdash t_1' : A \qquad x \notin \mathsf{dom}(H)}{H, x \mapsto_1 ([\Gamma_1] \vdash t_1' : A) \bowtie (\Gamma_2, x : [A])}\, \textsc{Lin}$$

and thus we can induct on the second multi reduction with this heap compatibility and the typing $\Gamma_2, x : [A] \vdash t_2 : *B$ to get that $a \in \mathsf{arrRefs}(v)$ (array references in v) we have:

$$a \mapsto_1 t' \in H, x \mapsto_\omega ([\Gamma_1] \vdash t_1' : A) \implies \exists t''. a \mapsto_1 t'' \in H''$$
$$\wedge\ a \notin \mathsf{dom}(H, varx| - > Inf([G1]| - t1' : A)) \implies \exists t''. a \mapsto_1 t'' \in H''$$

From which we can get the goal since heap assignment to x is not an array reference assignment so we have:

$$a \mapsto_1 t' \in H \implies \exists t''. a \mapsto_1 t'' \in H''$$
$$\wedge\ a \notin \mathsf{dom}(H) \implies \exists t''. a \mapsto_1 t'' \in H''$$

- (pairIntro) Trivially satisfies the theorem since the typing cannot conclude with a type of form $*A$.

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : *C}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\,(x, y) = t_1\,\mathsf{in}\,t_2 : *C}\ \otimes_E$$

  Proceeds in essentially the same way as the above. There are two cases that split into a beta and a congruence case, but neither head reduction manipulates the array references itself so apply induction and Lemma b.4.1 to get the outgoing uniqueness condition as in the (app) rule.

- (unitIntro)

  Trivially satisfies the theorem since the typing cannot conclude with a type of form $*A$.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : 1 \quad \Gamma_2 \vdash t_2 : *B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let\ unit}\ = t_1\,\mathsf{in}\,t_2 : *B}\ 1_E$$

  with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash \mathsf{let\ unit}\ = t_1\,\mathsf{in}\,t_2\ \Rightarrow\ H' \vdash *v \mid \Gamma' \mid \Delta$$

  which has two possible derivations:

  1.

$$\frac{\dfrac{H \vdash t_1 \rightsquigarrow H' \vdash t_1' \mid \Gamma_1 \mid \Delta_1}{H \vdash \mathsf{let\ unit}\ = t_1\,\mathsf{in}\,t_2 \rightsquigarrow H' \vdash \mathsf{let\ unit}\ = t_1'\,\mathsf{in}\,t_2 \mid \Gamma_1 \mid \Delta_1}\ \rightsquigarrow_{\text{APP}} \qquad H' \vdash \mathsf{let\ unit}\ = t_1'\,\mathsf{in}\,t_2\ \Rightarrow\ H'' \vdash *v \mid \Gamma_2 \mid \Delta_2}{H \vdash \mathsf{let\ unit}\ = t_1\,\mathsf{in}\,t_2\ \Rightarrow\ H'' \vdash *v \mid \Gamma_1 + \Gamma_2 \mid \Delta_1 + \Delta_2}\ \text{EXT}$$

  By conservation on the single reduction for t_1 we get type preservation which gives us $\Gamma_1 \vdash t_1' : 1$ and thus $\Gamma_1 + \Gamma_2 \vdash$

$\mathsf{let\,unit}\ = t_1'\ \mathsf{in}\ t_2 : *B$, and also that $H' \bowtie (\Gamma_0 + \Gamma')$. Thus we can induct on the multi-reduction here to get $a \in \mathsf{arrRefs}(v)$:

$$\begin{aligned} a \mapsto_1 t' \in H' &\implies \exists t''. a \mapsto_1 t'' \in H'' \\ \wedge\ a \notin \mathsf{dom}(H) &\implies \exists t''. a \mapsto_1 t'' \in H'' \end{aligned}$$

Applying this by transitivity (and instantiation) with Lemma b.4.1 gives us the goal:

$$\begin{aligned} a \mapsto_1 t' \in H &\implies \exists t''. a \mapsto_1 t'' \in H'' \\ \wedge\ a \notin \mathsf{dom}(H) &\implies \exists t''. a \mapsto_1 t'' \in H'' \end{aligned}$$

2. Alternatively $t_1 = \mathsf{unit}$ such that the typing is:

$$\frac{\dfrac{}{[\Gamma_1] \vdash \mathsf{unit} : 1}\,1_I \qquad \Gamma_2 \vdash t_2 : *B}{\Gamma_2 \vdash \mathsf{let\,unit}\ = \mathsf{unit}\ \mathsf{in}\ t_2 : *B}\,1_E$$

and we have reduction:

$$\frac{\dfrac{}{H \vdash \mathsf{let\,unit}\ = \mathsf{unit}\ \mathsf{in}\ t \rightsquigarrow H \vdash t \mid \emptyset \mid \emptyset}\,\rightsquigarrow_{\beta\mathsf{unit}} \qquad H \vdash t \Rightarrow H' \vdash t'' \mid \Gamma_2 \mid \Delta_2}{H \vdash \mathsf{let\,unit}\ = \mathsf{unit}\ \mathsf{in}\ t \Rightarrow H' \vdash *v \mid \Gamma_2 \mid \Delta_2}\ \textsc{ext}$$

Thus we arrive at the goal by induction on the second premise here.

- (return), (bind)

  All trivially satisfy the theorem since the typing cannot conclude with a type of form $*A$.

- (nec)

$$\frac{\emptyset \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

Then there is a possible multi-reduction:

$$\frac{\dfrac{H \vdash t \rightsquigarrow H' \vdash t' \mid \Gamma \mid \Delta}{H \vdash *t \rightsquigarrow H' \vdash *t' \mid \Gamma \mid \Delta} \rightsquigarrow_* \qquad H' \vdash *t' \Rightarrow H'' \vdash *v \mid \Gamma_2 \mid \Delta_2}{H \vdash *t \rightsquigarrow H'' \vdash *v \mid \Gamma_2 \mid \Delta_2}\ \textsc{ext}$$

By Lemma b.4.1 on the single reduction and induction on the second premise, we then get

$$\begin{aligned} a \mapsto_1 t' \in H &\implies \exists t''. a \mapsto_1 t'' \in H' \\ \wedge\ a \notin \mathsf{dom}(H) &\implies \exists t''. a \mapsto_1 t'' \in H' \end{aligned}$$

and

$$\begin{aligned} a \mapsto_1 t' \in H' &\implies \exists t''. a \mapsto_1 t'' \in H'' \\ \wedge\ a \notin \mathsf{dom}(H') &\implies \exists t''. a \mapsto_1 t'' \in H'' \end{aligned}$$

which we can combine together via instantiation and transitivity to get the goal:

$$\begin{aligned} a \mapsto_1 t' \in H &\implies \exists t''. a \mapsto_1 t'' \in H'' \\ \wedge\ a \notin \mathsf{dom}(H) &\implies \exists t''. a \mapsto_1 t'' \in H'' \end{aligned}$$

- (newArray) (readArray) (writeArray) (deleteArray) are all values and have no reduction, so the case is trivial here.

□

B.5 SOUNDNESS OF HEAP MODEL WRT. EQUATIONAL THEORY

Theorem 6 (Soundness with respect to the equational theory)**.** *For all t_1, t_2 such that $\Gamma \vdash t_1 : A$ and $\Gamma \vdash t_2 : A$ and $t_1 \equiv t_2$ and given H such that $H \bowtie \Gamma$, there exists a value (irreducible term) v and $\Gamma_1, \Gamma_2, \Delta_1, \Delta_2$ such that there are full β-reductions to the same value*

$$H \vdash t_1 \Rightarrow_\beta H' \vdash v \mid \Gamma_1 \mid \Delta_1 \quad \wedge \quad H \vdash t_2 \Rightarrow_\beta H' \vdash v \mid \Gamma_2 \mid \Delta_2$$

Proof. • (unitR)

$$\frac{}{\mathsf{clone}\, t\, \mathsf{as}\, x\, \mathsf{in}\, (\mathsf{share}\, x) \equiv t}\ \textsc{EqUnitR}$$

With the following typing derivation for the LHS:

$$\frac{\Gamma_1 \vdash t : !A \qquad \dfrac{x : *A \vdash x : *A}{x : *A \vdash \mathsf{share}\, x : !A}\ \textsc{share}}{\Gamma_1 \vdash \mathsf{clone}\, t\, \mathsf{as}\, x\, \mathsf{in}\, (\mathsf{share}\, x) : !A}\ \textsc{clone}$$

and from the premise we have $H \bowtie \Gamma_1$.

By conservation, we have that a well-typed term reduces to a value and so (by a value lemma) we can assume that $H \vdash t$ reduces to a value $H \vdash !t'$ for some t' (*) and then reduction under the ! yields $H'' \vdash !v$ (**). This gives the reduction of the RHS to a value.

We then have the following reduction sequence (where we elide the output binding context and usage context as they are not needed in the proof):

$$\begin{array}{rcl} & & H \vdash \mathsf{clone}\, t\, \mathsf{as}\, x\, \mathsf{in}\, (\mathsf{share}\, x) \\ (*) & \rightsquigarrow^* & H' \vdash \mathsf{clone}\, !t'\, \mathsf{as}\, x\, \mathsf{in}\, (\mathsf{share}\, x) \\ (**) & \rightsquigarrow^* & H' \vdash \mathsf{clone}\, !v\, \mathsf{as}\, x\, \mathsf{in}\, (\mathsf{share}\, x) \\ \rightsquigarrow_{\mathsf{clone}\beta} & \rightarrow & H', x \mapsto_1 * v \vdash (\mathsf{share}\, x) \\ \rightsquigarrow_{\mathsf{clone}} + \rightsquigarrow_{\mathsf{VAR1}} & \rightarrow & H', x \mapsto_1 * v \vdash (\mathsf{share}\, * v) \\ \rightsquigarrow_{\beta\mathsf{share}} & \rightarrow & H', x \mapsto_1 * t' \vdash\, !v \end{array}$$

which matches the result of deep reducing t (using the full beta) rules replaying (*) and (**) to get the same result of $!v$.

- (unitL)

$$\frac{}{\mathsf{clone}\, (\mathsf{share}\, v)\, \mathsf{as}\, x\, \mathsf{in}\, t' \equiv [v/x]t'}\,\textsc{EqunitL}$$

With the following typing derivation for the LHS:

$$\frac{\dfrac{\Gamma_1 \vdash v : *A}{\Gamma_1 \vdash \mathsf{share}\, v : !A}\,\textsc{share} \qquad \Gamma_2, x : *A \vdash t' : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{clone}\, (\mathsf{share}\, v)\, \mathsf{as}\, x\, \mathsf{in}\, t' : !B}\,\textsc{clone}$$

and from the premise we have $H \bowtie \Gamma_1 + \Gamma_2$.

By the value lemma (Lemma 10) we know that $v = *v'$ therefore we know that can perform the following reduction: (where we elide the output binding context and usage context as they are not needed in the proof):

$$\begin{array}{rcl} & & H \vdash \mathsf{clone}\, (\mathsf{share}\, (*v'))\, \mathsf{as}\, x\, \mathsf{in}\, t' \\ \rightsquigarrow_{\mathsf{clone}!} + \rightsquigarrow_{\beta\mathsf{share}} & \rightarrow & [H']_\omega, H'' \vdash \mathsf{clone}\, !v'\, \mathsf{as}\, x\, \mathsf{in}\, t' \\ \rightsquigarrow_{\mathsf{clone}\beta} & \rightarrow & [H']_\omega, H'', H''', x \mapsto_1 * (\theta(v)) \vdash t' \end{array}$$

where $H = H', H''$ and $\mathsf{dom}(H') \equiv \mathsf{arrRefs}(v)$ and $(H''', \theta) \equiv \mathsf{copy}(H')$.

Then we have to sub-cases that refine this reduction

- $v' = a$ (an array reference) then we can assume that that $H = H', a \mapsto_r \mathbf{arr}$ and we specialise the above reduction to get:

$$\begin{array}{rcl} & & H', a \mapsto_r \mathbf{arr} \vdash \mathsf{clone}\, (\mathsf{share}\, (*a))\, \mathsf{as}\, x\, \mathsf{in}\, t' \\ \rightsquigarrow_{\mathsf{clone}!} + \rightsquigarrow_{\beta\mathsf{share}} & \rightarrow & H', a \mapsto_\omega \mathbf{arr} \vdash \mathsf{clone}\, !v'\, \mathsf{as}\, x\, \mathsf{in}\, t' \\ \rightsquigarrow_{\mathsf{clone}\beta} & \rightarrow & H', a \mapsto_\omega \mathbf{arr}, a' \mapsto_\omega \mathbf{arr}, x \mapsto_1 * a' \vdash t' \end{array}$$

  Since t' cannot contain an a as it is a non-runtime term, then the resulting term here (in its heap context) reduces

down to a value that is equal $[*a/x]t'$ by full beta reduction; we have essentially just α-renamed the array reference here from a' to a.

– v' is not an array reference, therefore $\mathsf{arrRefs}(v') = \emptyset$ and $\theta = \emptyset$ and thus we have $H'' = \emptyset$, so the reduction specialises to:

$$\begin{array}{rcl} & & H', H'' \vdash \mathsf{clone}\,(\mathsf{share}\,(*v'))\,\mathsf{as}\,x\,\mathsf{in}\,t' \\ \rightsquigarrow_{\mathsf{clone!}} + \rightsquigarrow_{\beta\mathsf{share}} & \to & H', H'' \vdash \mathsf{clone}\,!v'\,\mathsf{as}\,x\,\mathsf{in}\,t' \\ \rightsquigarrow_{\mathsf{clone}\beta} & \to & H', H'', x{\mapsto_{\omega}} * v' \vdash t' \end{array}$$

which is equal to $[*v'/x]t'$ after full reduction.

- (assoc)

$$\frac{x \# t_3}{\mathsf{clone}\,t_1\,\mathsf{as}\,x\,\mathsf{in}\,(\mathsf{clone}\,t_2\,\mathsf{as}\,y\,\mathsf{in}\,t_3) \equiv \mathsf{clone}\,(\mathsf{clone}\,t_1\,\mathsf{as}\,x\,\mathsf{in}\,t_2)\,\mathsf{as}\,y\,\mathsf{in}\,t_3}\,\textsc{Eqassoc}$$

Thus we assume a reduction from t_1 to $!t_1'$ and then to $!v_1$ and likewise a reduction from t_2 to $!t_2'$ and then to $!v_2$ then we reduce as follows on the LHS:

$$\begin{array}{rl} & H \vdash \mathsf{clone}\,t_1\,\mathsf{as}\,x\,\mathsf{in}\,(\mathsf{clone}\,t_2\,\mathsf{as}\,y\,\mathsf{in}\,t_3) \\ \rightsquigarrow * & H' \vdash \mathsf{clone}\,!v_1\,\mathsf{as}\,x\,\mathsf{in}\,(\mathsf{clone}\,t_2\,\mathsf{as}\,y\,\mathsf{in}\,t_3) \\ \rightsquigarrow & H', x{\mapsto_1} * v_1 \vdash \mathsf{clone}\,t_2\,\mathsf{as}\,y\,\mathsf{in}\,t_3 \\ \rightsquigarrow * & H'' \vdash \mathsf{clone}\,!v_2\,\mathsf{as}\,y\,\mathsf{in}\,t_3 \\ \rightsquigarrow & H'', y{\mapsto_1} * v_2 \vdash t_3 \end{array}$$

and on the RHS:

$$\begin{array}{rl} & H \vdash \mathsf{clone}\,(\mathsf{clone}\,t_1\,\mathsf{as}\,x\,\mathsf{in}\,t_2)\,\mathsf{as}\,y\,\mathsf{in}\,t_3 \\ \rightsquigarrow * & H' \vdash \mathsf{clone}\,(\mathsf{clone}\,!v_1\,\mathsf{as}\,x\,\mathsf{in}\,t_2)\,\mathsf{as}\,y\,\mathsf{in}\,t_3 \\ \rightsquigarrow & H', x{\mapsto_1} * v_1 \vdash \mathsf{clone}\,t_2\,\mathsf{as}\,y\,\mathsf{in}\,t_3 \\ \rightsquigarrow * & H'' \vdash \mathsf{clone}\,!v_2\,\mathsf{as}\,y\,\mathsf{in}\,t_3 \\ \rightsquigarrow & H'', y{\mapsto_1} * v_2 \vdash t_3 \end{array}$$

matching in both sides.

□

C | APPENDIX FOR CHAPTER 5

C.1 COLLECTED RULES

C.1.1 Typing

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs} \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I \qquad \frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C}\ \otimes_E$$

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : B}\ 1_E$$

$$\frac{\Gamma \vdash t : A \qquad \neg \mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{elim} \qquad \frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx}$$

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathbf{share}\,t : \Box_r A}\ \textsc{share} \qquad \frac{\begin{array}{c}1 \sqsubseteq r \qquad \mathsf{cloneable}(A) \\ \Gamma_1, \overline{id} \vdash t_1 : \Box_r A \qquad \Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id}]) \vdash t_2 : B\end{array}}{(\Gamma_1 + \Gamma_2), \overline{id} \vdash \mathbf{clone}\,t_1\,\mathbf{as}\,x\,\mathbf{in}\,t_2 : B}\ \textsc{clone}$$

$$\frac{\Gamma_1 \vdash t_1 : *A \qquad \Gamma_2 \vdash t_2 : \&_1 A \multimap \&_1 B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{withBorrow}\ t_1\ t_2 : *B}\ \textsc{withBorrow}$$

$$\frac{\Gamma \vdash t : \&_p A}{\Gamma \vdash \mathbf{split}\,t : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\ \textsc{split} \qquad \frac{\Gamma_1 \vdash t_1 : \&_p A \qquad \Gamma_2 \vdash t_2 : \&_q A \qquad p + q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \mathbf{join}\ t_1\ t_2 : \&_{p+q} A}\ \textsc{join}$$

$$\frac{\Gamma \vdash t : \&_p (A \otimes B)}{\Gamma \vdash \mathbf{push}\,t : (\&_p A) \otimes (\&_p B)}\ \textsc{push} \qquad \frac{\Gamma \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma \vdash \mathbf{pull}\,t : \&_p (A \otimes B)}\ \textsc{pull}$$

$$\frac{\Gamma \vdash t : A \qquad id \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathbf{pack}\ \langle id', t \rangle : \exists id.A[id/id']}\ \textsc{pack} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t_1 : \exists id.A \\ \Gamma_2, id, x : A \vdash t_2 : B \qquad id \notin \mathsf{fv}(B)\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ \langle id, x \rangle = t_1\ \mathbf{in}\ t_2 : B}\ \textsc{unpack}$$

RUNTIME TYPING

$$\frac{\gamma \vdash t : A}{0 \cdot \Gamma, \gamma \vdash *t : \&_p A}\ \textsc{nec} \qquad \frac{}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash ref : Res_{id}\ A}\ \textsc{ref}$$

$$\frac{\Gamma \vdash t : \&_1 A}{\Gamma \vdash \textbf{unborrow}\ t : *A}\ \textsc{unborrow}$$

$$\frac{}{\emptyset \vdash \mathsf{init} : \mathsf{Array}_{id}\ \mathbb{F}}\ \textsc{arrayInit} \qquad \frac{\emptyset \vdash \textbf{arr} : \mathsf{Array}_{id}\ \mathbb{F} \qquad \emptyset \vdash v : \mathbb{F}}{\emptyset \vdash \textbf{arr}[n] = v : \mathsf{Array}_{id}\ \mathbb{F}}\ \textsc{arrayAt}$$

$$\frac{\gamma \vdash v : A}{\gamma \vdash \textbf{ref}(v) : \mathsf{Ref}_{id}\ A}\ \textsc{refStore} \qquad \frac{\Gamma \vdash t : A \qquad \neg \mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t]_r : \Box_r A}\ \textsc{pr}$$

We sometimes use an admissible rule to simplify some parts of the proofs:

$$\frac{}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash *ref : *(Res_{id}\ A)}\ *\textsc{ref}^*$$

which has derivation:

$$\frac{\dfrac{}{ref : Res_{id}\ A \vdash ref : Res_{id}\ A}\ \textsc{ref}}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash *ref : *(Res_{id}\ A)}\ \textsc{nec}$$

PRIMITIVES

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{newRef} : A \multimap \exists id.*(\mathsf{Ref}_{id}\ A)}\ \textsc{newRef}$$

$$\frac{p \equiv 1 \ \vee \ p \equiv *}{0 \cdot \Gamma \vdash \mathbf{swapRef} : \&_p(\mathsf{Ref}_{id}\ A) \multimap A \multimap A \otimes \&_p(\mathsf{Ref}_{id}\ A)}\ \textsc{swapRef}$$

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{freezeRef} : *(\mathsf{Ref}_{id}\ A) \multimap A}\ \textsc{freezeRef}$$

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{readRef} : \&_p(\mathsf{Ref}_{id}\ (\Box_{r+1}A)) \multimap A \otimes \&_p(\mathsf{Ref}_{id}\ (\Box_r A))}\ \textsc{readRef}$$

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{newArray} : \mathbb{N} \multimap \exists id.*(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{newArray}$$

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{readArray} : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{readArray}$$

$$\frac{p \equiv 1 \ \vee \ p \equiv *}{0 \cdot \Gamma \vdash \mathbf{writeArray} : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{writeArray}$$

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{deleteArray} : *(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathsf{unit}}\ \textsc{delArray}$$

Definition 17 (Graded contexts). $[\Gamma]$ *classifies those contexts which contain only graded variables:*

$$\frac{}{[\varnothing]} \qquad \frac{[\Gamma]}{[\Gamma, x : [A]_r]}$$

Definition 18 (Copyable predicate). *Predicate definition:*

$$\frac{}{\mathsf{copyable}(\mathsf{unit})} \quad \frac{}{\mathsf{copyable}(\mathbb{N})} \quad \frac{}{\mathsf{copyable}(\mathbb{F})} \quad \frac{\mathsf{copyable}(A) \quad \mathsf{copyable}(B)}{\mathsf{copyable}(A \otimes B)}$$

Definition 19 (Cloneable predicate). *Predicate definition:*

$$\frac{}{\mathsf{cloneable}(\mathsf{Array}_{id}\ \mathbb{F})} \quad \frac{\mathsf{cloneable}(A) \ \vee \ \mathsf{copyable}(A)}{\mathsf{cloneable}(\mathsf{Ref}_{id}\ A)} \quad \frac{\mathsf{cloneable}(A) \quad \mathsf{cloneable}(B)}{\mathsf{cloneable}(A \otimes B)}$$

Definition 20 (Resource allocating terms). *Predicate definition:*

$$\frac{}{\mathsf{resourceAllocator}(\textbf{\textit{newRef}})} \qquad \frac{}{\mathsf{resourceAllocator}(\textbf{\textit{newArray}})}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}((\lambda x.t_1)\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ () = t_1\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ () = t_1\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ (x, y) = t_1\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ (x, y) = t_1\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}((t_1, t_2))} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}((t_1, t_2))}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ [x] = t_1\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ [x] = t_1\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t)}{\mathsf{resourceAllocator}([t])} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{share}\ t_1)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{clone}\ t_1\ \textbf{\textit{as}}\ x\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{clone}\ t_1\ \textbf{\textit{as}}\ x\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{withBorrow}}\ t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{withBorrow}}\ t_1\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{split}}\ t_1)} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{join}}\ t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{join}}\ t_1\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{push}\ t_1)} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{pull}\ t_1)} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{pack}}\ \langle id', t_1\rangle)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{unpack}}\ \langle id', x\rangle = t_1\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{unpack}}\ \langle id', x\rangle = t_1\ \textbf{\textit{in}}\ t_2)}$$

C.1.2 Reduction rules for heap semantics

$$\frac{\exists r'.\ s + r' \sqsubseteq r}{H, x{\mapsto_r} v \vdash x \rightsquigarrow_s H, x{\mapsto_r} v \vdash v}\ \rightsquigarrow_{\text{VAR}} \qquad \frac{y\#\{H, v, t\}}{H \vdash (\lambda x.t)\, v \rightsquigarrow_s H, y{\mapsto_s} v \vdash t[y/x]}\ \rightsquigarrow_{\beta}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash t_1\, t_2 \rightsquigarrow_s H' \vdash t_1'\, t_2}\ \rightsquigarrow_{\text{APPL}} \qquad \frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash v\, t_2 \rightsquigarrow_s H' \vdash v\, t_2'}\ \rightsquigarrow_{\text{APPR}}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash (t_1, t_2) \rightsquigarrow_s H' \vdash (t_1', t_2)}\ \rightsquigarrow_{\otimes\text{L}} \qquad \frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash (v, t_2) \rightsquigarrow_s H' \vdash (v, t_2')}\ \rightsquigarrow_{\otimes\text{R}}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,(x, y) = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{LET}\otimes}$$

$$\frac{x'\#\{H, v_1, v_2, t\} \qquad y'\#\{H, v_1, v_2, t\}}{H \vdash \mathbf{let}\,(x, y) = (v_1, v_2)\,\mathbf{in}\,t \rightsquigarrow_s H, x'{\mapsto_s} v_1, y'{\mapsto_s} v_2 \vdash t[y'/y][x'/x]}\ \rightsquigarrow_{\otimes\beta}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,() = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{LETUNIT}} \qquad \frac{}{H \vdash \mathbf{let}\,() = ()\,\mathbf{in}\,t \rightsquigarrow_s H \vdash t}\ \rightsquigarrow_{\beta\text{UNIT}}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{LET}\Box}$$

$$\frac{H \vdash t \rightsquigarrow_{s*r} H' \vdash t'}{H \vdash [t]_r \rightsquigarrow_s H' \vdash [t']_r}\ \rightsquigarrow_{\Box}$$

$$\frac{y\#\{H, v, t\}}{H \vdash \mathbf{let}\,[x] = [v]_r\,\mathbf{in}\,t \rightsquigarrow_s H, y{\mapsto_{(s*r)}} v \vdash t[y/x]}\ \rightsquigarrow_{\Box\beta}$$

$$\frac{y\#\{H, v, t\}}{H \vdash \mathbf{unpack}\,\langle id, x\rangle = \mathbf{pack}\,\langle id', v\rangle\,\mathbf{in}\,t \rightsquigarrow_s H, y{\mapsto_r} v \vdash t[y/x]}\ \rightsquigarrow_{\exists\beta}$$

$$\frac{H \vdash t \rightsquigarrow_s H \vdash t'}{H \vdash \mathbf{pack}\,\langle id, t\rangle \rightsquigarrow_s H \vdash \mathbf{pack}\,\langle id, t'\rangle}\ \rightsquigarrow_{\text{PACK}}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H \vdash t_1'}{H \vdash \mathbf{unpack}\,\langle id, x\rangle = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H \vdash \mathbf{unpack}\,\langle id, x\rangle = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{UNPACK}}$$

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{share}\,t \rightsquigarrow_s H' \vdash \mathbf{share}\,t'}\ \rightsquigarrow_{\text{SHARE}} \qquad \frac{\mathsf{dom}(H) \equiv \mathsf{refs}(v)}{H, H' \vdash \mathbf{share}\,(*v) \rightsquigarrow_s ([H]_0), H' \vdash [v]}\ \rightsquigarrow_{\text{SHARE}\beta}$$

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash *t \rightsquigarrow_s H' \vdash *t'}\ \rightsquigarrow_{*} \qquad \frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{clone}\,t_1\,\mathbf{as}\,x\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{clone}\,t_1'\,\mathbf{as}\,x\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{CLONE}}$$

$$\frac{\mathsf{dom}(H') \equiv \mathsf{refs}(v) \qquad (H'', \theta, \overline{id}) = \mathsf{copy}(H') \qquad y\#\{H, v, t\}}{H, H' \vdash \mathbf{clone}\ [v]_r\ \mathbf{as}\ x\ \mathbf{in}\ t \rightsquigarrow_s H, H', H'', y \mapsto_s \mathbf{pack}\ \langle \overline{id}, *(\theta(v)) \rangle \vdash t[y/x]}\ \rightsquigarrow_{\text{CLONE}\beta}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{withBorrow}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \mathbf{withBorrow}\ t_1'\ t_2}\ \rightsquigarrow_{\text{WITH\&L}}$$

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{withBorrow}\ (\lambda x.t_1)\ t_2 \rightsquigarrow_s H' \vdash \mathbf{withBorrow}\ (\lambda x.t_1)\ t_2'}\ \rightsquigarrow_{\text{WITH\&R}}$$

$$\frac{y\#\{H, v, t\}}{H \vdash \mathbf{withBorrow}\ (\lambda x.t)\ (*v) \rightsquigarrow_s H, y \mapsto_s (*v) \vdash \mathbf{unborrow}\ t[y/x]}\ \rightsquigarrow_{\text{WITH\&}}$$

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{unborrow}\ t \rightsquigarrow_s H' \vdash \mathbf{unborrow}\ t'}\ \rightsquigarrow_{\text{UNBORROW}}$$

$$\frac{}{H \vdash \mathbf{unborrow}\ (*v) \rightsquigarrow_s H \vdash *v}\ \rightsquigarrow_{\text{UN\&}}$$

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{split}\ t \rightsquigarrow_s H' \vdash \mathbf{split}\ t'}\ \rightsquigarrow_{\text{SPLIT}}$$

$$\frac{ref_1\#H \qquad ref_2\#H}{H, ref \mapsto_p id, id \mapsto v \vdash \mathbf{split}\ (*ref) \rightsquigarrow_s H, ref_1 \mapsto_{\frac{p}{2}} id, ref_2 \mapsto_{\frac{p}{2}} id, id \mapsto v \vdash (*ref_1, *ref_2)}\ \rightsquigarrow_{\text{SPLITREF}}$$

$$\frac{\begin{array}{c} H \vdash \mathbf{split}\ (*v) \rightsquigarrow_s H' \vdash (*v_1, *v_2) \\ H' \vdash \mathbf{split}\ (*w) \rightsquigarrow_s H'' \vdash (*w_1, *w_2) \end{array}}{H \vdash \mathbf{split}\ (*(v, w)) \rightsquigarrow_s H'' \vdash (*(v_1, w_1), *(v_2, w_2))}\ \rightsquigarrow_{\text{SPLIT}\otimes}$$

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{join}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ t_1'\ t_2}\ \rightsquigarrow_{\text{JOINL}} \qquad \frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{join}\ v\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ v\ t_2'}\ \rightsquigarrow_{\text{JOINR}}$$

$$\frac{ref\#H}{H, ref_1 \mapsto_p id, ref_2 \mapsto_q id, id \mapsto v \vdash \mathbf{join}\ (*ref_1)\ (*ref_2) \rightsquigarrow_s H, ref \mapsto_{(p+q)} id, id \mapsto v \vdash *ref}\ \rightsquigarrow_{\text{JOINREF}}$$

$$\frac{\begin{array}{c} H \vdash \mathbf{join}\ (*v_1)\ (*v_2) \rightsquigarrow_s H' \vdash *v \\ H' \vdash \mathbf{join}\ (*w_1)\ (*w_2) \rightsquigarrow_s H'' \vdash *w \end{array}}{H \vdash \mathbf{join}\ (*(v_1, w_1))\ (*(v_2, w_2)) \rightsquigarrow_s H'' \vdash *(v, w)}\ \rightsquigarrow_{\text{JOIN}\otimes}$$

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{push}\ t \rightsquigarrow_s H' \vdash \mathbf{push}\ t'}\ \rightsquigarrow_{\text{PUSH}} \qquad \frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{pull}\ t \rightsquigarrow_s H' \vdash \mathbf{pull}\ t'}\ \rightsquigarrow_{\text{PULL}}$$

$$\frac{}{H \vdash \mathbf{push}\ (*(v_1, v_2)) \rightsquigarrow_s H \vdash (*v_1, *v_2)}\ \rightsquigarrow_{\text{PUSH}*}$$

$$\frac{}{H \vdash \mathbf{pull}\ (*v_1, *v_2) \rightsquigarrow_s H \vdash *(v_1, v_2)}\ \rightsquigarrow_{\text{PULL}*}$$

PRIMITIVE REDUCTION RULES

$$\frac{ref\#H \qquad id\#H}{H \vdash \mathbf{newArray}\, n \;\rightsquigarrow_s\; H, ref \mapsto_1 id, id \mapsto \mathsf{init} \vdash \mathbf{pack}\, \langle id, *ref \rangle}\; \rightsquigarrow_{\text{NEWARRAY}}$$

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{arr}[i] = v \vdash \mathbf{readArray}\, (*ref)\, i \;\rightsquigarrow_s\; H, ref \mapsto_p id, id \mapsto \mathbf{arr}[i] = v \vdash (v, *ref)}\; \rightsquigarrow_{\text{READARRAY}}$$

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\, (*ref)\, i\, v \;\rightsquigarrow_s\; H, ref \mapsto_p id, id \mapsto \mathbf{arr}[i] = v \vdash *ref}\; \rightsquigarrow_{\text{WRITEARRAY}}$$

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{arr} \vdash \mathbf{deleteArray}\, (*ref) \;\rightsquigarrow_s\; H \vdash ()}\; \rightsquigarrow_{\text{DELETEARRAY}}$$

$$\frac{ref\#H \qquad id\#H}{H \vdash \mathbf{newRef}\, v \;\rightsquigarrow_s\; H, ref \mapsto_1 id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{pack}\, \langle id, *ref \rangle}\; \rightsquigarrow_{\text{NEWREF}}$$

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{swapRef}\, (*ref)\, v' \;\rightsquigarrow_s\; H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v') \vdash v}\; \rightsquigarrow_{\text{SWAPREF}}$$

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{freezeRef}\, (*ref) \;\rightsquigarrow_s\; H \vdash v}\; \rightsquigarrow_{\text{FREEZEREF}}$$

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_{r+1}) \vdash \mathbf{readRef}\, (*ref) \;\rightsquigarrow_s\; H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_r) \vdash (v, *ref)}\; \rightsquigarrow_{\text{READREF}}$$

MULTI-REDUCTION RULES

$$\frac{}{H \vdash t \;\Rightarrow_s\; H \vdash t}\; \textsc{refl} \qquad \frac{H \vdash t_1 \;\rightsquigarrow_s\; H' \vdash t_2 \qquad H' \vdash t_2 \;\Rightarrow_s\; H'' \vdash t_3}{H \vdash t_1 \;\Rightarrow_s\; H'' \vdash t_3}\; \textsc{ext}$$

HEAP-CONTEXT COMPATIBILITY

$$\frac{}{\emptyset \bowtie \emptyset}\; \textsc{empty} \qquad \frac{H \bowtie \emptyset}{H, ref \mapsto_p id \bowtie \emptyset}\; \textsc{GCArr} \qquad \frac{H, id \mapsto v_r \bowtie \Gamma + \gamma \qquad \gamma \vdash v_r : Res_{id}\, A}{H, ref \mapsto_p id, id \mapsto v_r \bowtie (\Gamma, ref : Res_{id}\, A)}\; \textsc{extRes}$$

$$\frac{H \bowtie \Gamma + s \cdot \Gamma' \qquad x \notin \mathsf{dom}(H) \qquad \Gamma' \vdash v : A \qquad \exists r'.\, s + r' \equiv r}{(H, x \mapsto_r v) \bowtie (\Gamma, x : [A]_s)}\; \textsc{ext}$$

$$\frac{H \bowtie \Gamma + \Gamma' \qquad x \notin \mathsf{dom}(H) \qquad \Gamma' \vdash v : A \qquad \exists r'.\, 1 + r' \equiv r}{(H, x \mapsto_r v) \bowtie (\Gamma, x : A)}\; \textsc{extLin}$$

C.1.3 Equational theory

$$(\lambda x.t_2)\, t_1 \equiv t_2[t_1/x] \qquad (\beta)$$
$$\lambda x.(t\, x) \equiv t \qquad (\eta)$$
$$\mathbf{let}\,[x] = [t_1]\,\mathbf{in}\,t_2 \equiv t_2[t_1/x] \qquad (\beta_\Box)$$
$$\mathbf{let}\,[x] = t_1\,\mathbf{in}\,[x] \equiv t_1 \qquad (\eta_\Box)$$
$$[\mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2] \equiv \mathbf{let}\,[x] = t_1\,\mathbf{in}\,[t_2] \qquad (\Box\text{distrib})$$
$$\mathbf{let}\,() = ()\,\mathbf{in}\,t \equiv t \qquad (\beta_{\mathsf{unit}})$$
$$\mathbf{let}\,() = t\,\mathbf{in}\,() \equiv t \qquad (\eta_{\mathsf{unit}})$$
$$\mathbf{let}\,(x,y) = (t_1,t_2)\,\mathbf{in}\,t_3 \equiv t_3[t_2/y][t_1/x] \qquad (\beta_\otimes)$$
$$\mathbf{let}\,(x,y) = t_1\,\mathbf{in}\,(x,y) \equiv t_1 \qquad (\eta_\otimes)$$
$$(\mathbf{let}\,(x,y) = t_1\,\mathbf{in}\,t_2, t_3) \equiv \mathbf{let}\,(x,y) = t_1\,\mathbf{in}\,(t_2,t_3) \qquad (\otimes\text{distribL})$$
$$(t_1, \mathbf{let}\,(x,y) = t_2\,\mathbf{in}\,t_3) \equiv \mathbf{let}\,(x,y) = t_2\,\mathbf{in}\,(t_1,t_3) \qquad (\otimes\text{distribR})$$
$$\mathbf{unpack}\,\langle id, x\rangle = \mathbf{pack}\,\langle id', t_1\rangle\,\mathbf{in}\,t_2 \equiv t_2[t_1/x] \qquad (\beta_\exists)$$
$$\mathbf{unpack}\,\langle id, x\rangle = t_1\,\mathbf{in}\,\mathbf{pack}\,\langle id, x\rangle \equiv t_1 \qquad (\eta_\exists)$$
$$\mathbf{pack}\,\langle id, (\mathbf{unpack}\,\langle id', x\rangle = t_1\,\mathbf{in}\,t_2)\rangle \equiv \mathbf{unpack}\,\langle id', x\rangle = t_1\,\mathbf{in}\,\mathbf{pack}\,\langle id, t_2\rangle \qquad (\exists\text{distrib})$$

$$\mathbf{clone}\,(\mathbf{share}\,v)\,\mathbf{as}\,x\,\mathbf{in}\,t \equiv t[\mathbf{pack}\,\langle \overline{id}, v\rangle/x] \qquad (\beta_*)$$
$$\mathbf{clone}\,t_1\,\mathbf{as}\,x\,\mathbf{in}\,(\mathbf{clone}\,t_2\,\mathbf{as}\,y\,\mathbf{in}\,t_3) \equiv \mathbf{clone}\,(\mathbf{clone}\,t_1\,\mathbf{as}\,x\,\mathbf{in}\,t_2)\,\mathbf{as}\,y\,\mathbf{in}\,t_3 \quad (x \notin \mathsf{FV}(t_3)) \qquad (*\text{assoc})$$

$$\mathbf{withBorrow}\,(\lambda x.x)\,t \equiv t \qquad (\&\text{unit})$$
$$\mathbf{withBorrow}\,(\lambda x.f\,(g\,x))\,t \equiv \mathbf{withBorrow}\,f\,(\mathbf{withBorrow}\,g\,t) \qquad (\&\text{assoc})$$
$$(\mathbf{let}\,(x,y) = (\mathbf{split}\,t)\,\mathbf{in}\,(\mathbf{join}\,x\,y)) \equiv t \qquad (\&\text{rejoin})$$
$$\mathbf{split}\,(\mathbf{join}\,t_1\,t_2) \equiv (t_1,t_2) \qquad (\&\text{resplit})$$

C.1.4 Parallel sum example in Granule

```
-- A more involved example summing two borrowed halves of a unique
    array in parallel.
-- Run ‘main‘ to see the result.

-- Sized vectors
data Vec (n : Nat) t where
  Nil  : Vec 0 t;
  Cons : t → Vec n t → Vec (n+1) t

-- Length of a ‘Vec‘ into an indexed ‘N‘, preserving the elements
length' : ∀ {a : Type, n : Nat} . Vec n a → (Int, Vec n a)
length' Nil = (0, Nil);
```

```
length' (Cons x xs) = let (n, xs) = length' xs in (n + 1, Cons x xs)

-- Converts a vector of floats to a unique array of floats
toFloatArray : ∀ {n : Nat}
    . Vec n Float → ∃ {id : Name} . *(FloatArray id)
toFloatArray v =
  let (n', v) = length' v
  in unpack <id, arr> = newFloatArray n'
  in pack <id, (toFloatArrayAux arr [0] v)> as
    ∃ {id : Name} . *(FloatArray id)

-- Auxiliary function for `toFloatArray`
toFloatArrayAux : ∀ {n : Nat, id : Name}
    . *(FloatArray id) → Int [n] → Vec n Float → *(FloatArray id)
toFloatArrayAux a [n] Nil = a;
toFloatArrayAux a [n] (Cons x xs) =
  toFloatArrayAux (writeFloatArray a n x) [n + 1] xs

-- `sumFromTo a i n` sums the elements of a unique array `a` from
    index `i` to index `n`
sumFromTo : ∀ {id : Name, p : Fraction}
    . & p (FloatArray id) →  !Int → !Int → (Float,
    & p (FloatArray id))
sumFromTo array [i] [n] =
  if i == n then (0.0, array)
    else
      let (x, a) = readFloatArray array i;
          (y, arr) = sumFromTo a [i+1] [n]
      in  (x + y, arr)

-- Helper function `writeRef` for updating a reference where the old
     value is dropped
-- (A reference to a "Droppable" value can be written to without
    violating linearity)
writeRef : ∀ {id : Name, a : Type}
    . {Droppable a} ⇒ a → & 1 (Ref id a) → & 1 (Ref id a)
writeRef x r = let
    (y, r') = swapRef r x;
    () = drop@a y in r'

-- Parallel sum of two halves of a unique array, storing the result
    in a mutable
-- reference after the parallel computation is done.
parSum : ∀ {id id' : Name} . *(FloatArray id) → *(Ref id' Float)
                           → *(Ref id' Float, FloatArray id)
parSum array ref = let
      ([n], array) : (!Int, *(FloatArray id))
        = lengthFloatArray array;
      compIn
        = pull (ref, array)
  in withBorrow (λcompIn →
                let (ref, array)      = push compIn;
                    (array1, array2)  = split array;

            -- Compute in parallel
                    ((x, array1), (y, array2)) =
                        par (λ() → sumFromTo array1 [0] [div n 2])
```

```
                                (λ() → sumFromTo array2 [div n 2] [n]);

            -- Update the reference
                    ref'        = writeRef ((x : Float) + y) ref;
                    compOut     = pull (ref', join (array1, array2))

                in compOut) compIn

-- Main function to sum the elements of a unique array in parallel
main : Float
main =
  -- Some example data
  unpack <id , arr> = toFloatArray (Cons 10.0 (Cons 20.0 (Cons 30.0
    (Cons 40.0 Nil)))) in
  unpack <id', ref> = newRef 0.0 in
  let
      (result, array) = push (parSum arr ref);
      () = deleteFloatArray array
  in freezeRef result
```

C.2 SUBSTITUTION PROOFS

Lemma 8 (Linear substitution is admissible, extending Chapter 3)**.** *If* $\Gamma_1 \vdash t_1 : A$ *and* $\Gamma_2, x : A \vdash t_2 : B$ *then* $\Gamma_2 + \Gamma_1 \vdash t_2[t_1/x] : B$.

Proof. By induction on the typing derivation of t_2.

- (pr)

$$\frac{\Gamma \vdash t : A \qquad \neg \mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr}$$

  where $t_2 = [t]$. Trivial since the form of the typing does not match here: no linear variable possible.

- (share)

$$\frac{\Gamma_2, x : A \vdash t : *A}{\Gamma_2, x : A \vdash \mathbf{share}\, t : \Box_r A}\ \textsc{share}$$

  where $B = \Box_r A$.

  By induction on the premise then $\Gamma_1 + \Gamma_2 \vdash t[t_1/x] : *A$, from which we build the conclusion:

$$\frac{\Gamma_2 \vdash t[t_1/x] : *A}{\Gamma_2 \vdash \mathbf{share}\,(t[t_1/x]) : \Box_r A}\ \textsc{share}$$

- (bind) Two possibilities:

  1. Linear variable x in the left premise:

$$\frac{\Gamma_1', x : A \vdash t_1' : \Box_r A' \qquad \Gamma_2', y : *(\#A') \vdash t_2' : \Box_r B \qquad r \sqsubseteq 1}{\Gamma_1', x : A + \Gamma_2' \vdash \mathbf{clone}\, t_1' \,\mathbf{as}\, y \,\mathbf{in}\, t_2' : \Box_r B}\ \textsc{clone}'$$

By induction on the first premise: $\Gamma_1' + \Gamma_1 \vdash t_1'[t/x] : \Box_r A'$

Then we reconstruct the typing as:

$$\frac{\Gamma_1 + \Gamma_1 \vdash t_1'[t/x] : \Box_r A' \qquad \Gamma_2', y : *(\#A') \vdash t_2' : \Box_r B' \qquad r \sqsubseteq 1}{\Gamma_1' + \Gamma_1 + \Gamma_2' \vdash \mathbf{clone}\, t_1'[t/x]\, \mathbf{as}\, y\, \mathbf{in}\, t_2' : \Box_r B'}\ \textsc{clone}'$$

satisfying the goal (by commutativity of $+$).

2. Linear variable x in the right premise:

$$\frac{\Gamma_1' \vdash t_1' : \Box_r A' \qquad \Gamma_2', x : A, y : *(\#A') \vdash t_2' : \Box_r B' \qquad r \sqsubseteq 1}{\Gamma_1' + \Gamma_2', x : A \vdash \mathbf{clone}\, t_1'\, \mathbf{as}\, y\, \mathbf{in}\, t_2' : \Box_r B'}\ \textsc{clone}'$$

By induction on the second premise: $(\Gamma_2' + \Gamma_1), y : *A' \vdash t_2'[t/x] : \Box_r B'$

Then we reconstruct the typing as:

$$\frac{\Gamma_1' \vdash t_1' : \Box_r A' \qquad (\Gamma_2' + \Gamma_1), y : *(\#A') \vdash t_2'[t/x] : \Box_r B'}{\Gamma_1' + \Gamma_2' + \Gamma_1 \vdash \mathbf{clone}\, t_1'\, \mathbf{as}\, y\, \mathbf{in}\, t_2'[t/x] : \Box_r B'}\ \textsc{clone}$$

satisfying the goal.

- (withBorrow) Two possibilities:

1. Linear variable in the first premise:

$$\frac{\Gamma_1', x : A \vdash t : *A' \qquad \Gamma_2' \vdash f : \&_1 A' \multimap \&_1 B'}{\Gamma_1', x : A + \Gamma_2' \vdash \mathbf{withBorrow}\ f\ t : *B'}\ \textsc{withBorrow}$$

By induction on the first premise then $\Gamma_1' + \Gamma_1 \vdash t[t_1/x] : *A'$.

From this we construct the goal:

$$\frac{\Gamma_1' + \Gamma_1 \vdash t[t_1/x] : *A' \qquad \Gamma_2' \vdash f : \&_1 A' \multimap \&_1 B'}{\Gamma_1' + \Gamma_1 + \Gamma_2' \vdash \mathbf{withBorrow}\ f\ t[t_1/x] : *B'}\ \textsc{withBorrow}$$

satisfying the goal by commutativity of $+$.

2. Linear variable in the second premise:

$$\frac{\Gamma_1' \vdash t : *A' \qquad \Gamma_2', x : A \vdash f : \&_1 A' \multimap \&_1 B'}{\Gamma_1' + \Gamma_2', x : A \vdash \mathbf{withBorrow}\ f\ t : *B'}\ \textsc{withBorrow}$$

By induction on the second premise then $\Gamma_2' + \Gamma_1 \vdash f[t_1/x] : \&_1 A' \multimap \&_1 B'$.

From this we construct the goal:

$$\frac{\Gamma_1' \vdash t : *A' \qquad \Gamma_2' + \Gamma_1 \vdash f[t_1/x] : \&_1 A' \multimap \&_1 B'}{\Gamma_1' + \Gamma_2' + \Gamma_1 \vdash \mathbf{withBorrow}\ f[t_1/x]\ t : *B'}\ \textsc{withBorrow}$$

satisfying the goal.

- (split)

$$\frac{\Gamma, x : A \vdash t : \&_{p+q} A}{\Gamma, x : A \vdash \mathbf{split}\ t : \&_p A \otimes \&_q A}\ \textsc{withBorrow}$$

Then by induction on the premise we have: $\Gamma_1 + \Gamma \vdash t[t_1/x] : B$ from which we construct the goal:

$$\frac{\Gamma_1 + \Gamma \vdash t[t_1/x] : \&_{p+q} A}{\Gamma_1 + \Gamma \vdash \mathbf{split}\ (t[t_1/x]) : \&_p A \otimes \&_q A}\ \textsc{withBorrow}$$

- (join)

$$\frac{\Gamma_1 \vdash t_1' : \&_p A \qquad \Gamma_2 \vdash t_2' : \&_q A \qquad p + q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \mathbf{join}\ t_1'\ t_2' : \&_{p+q} A}\ \textsc{withBorrow}$$

Then there are two possibilities depending on the location of the linear typing variable:

1. (on the left):

$$\frac{\Gamma_1', x : A \vdash t_1' : \&_p A \qquad \Gamma_2' \vdash t_2' : \&_q A \qquad p + q \leq 1}{\Gamma_1', x : A + \Gamma_2' \vdash \mathbf{join}\ t_1'\ t_2' : \&_{p+q} A}\ \textsc{withBorrow}$$

Then by induction on the premise we have: $\Gamma_1' + \Gamma_1 \vdash t_1'[t_1/x] : \&_p A$ from which we construct the goal:

$$\frac{\Gamma_1', x : A \vdash t_1'[t/x] : \&_p A \qquad \Gamma_2' \vdash t_2' : \&_q A \qquad p + q \leq 1}{\Gamma_1' + \Gamma_1 + \Gamma_2' \vdash \mathbf{join}\ (t_1'[t_1/x])\ t_2' : \&_{p+q} A}\ \textsc{withBorrow}$$

2. (on the right):

$$\frac{\Gamma_1' \vdash t_1' : \&_p A \qquad \Gamma_2', x : A \vdash t_2' : \&_q A \qquad p + q \leq 1}{\Gamma_1' + \Gamma_2', x : A \vdash \mathbf{join}\ t_1'\ t_2' : \&_{p+q} A}\ \textsc{withBorrow}$$

Then by induction on the premise we have: $\Gamma_2' + \Gamma_1 \vdash t_2'[t_1/x] : \&_q A$ from which we construct the goal:

$$\frac{\Gamma_1' \vdash t_1' : \&_p A \qquad \Gamma_2' + \Gamma_1 \vdash t_2'[t_1/x] : \&_q A \qquad p + q \leq 1}{\Gamma_1' + \Gamma_2' + \Gamma_1 \vdash \mathbf{join}\ t_1'\ (t_2'[t_1/x]) : \&_{p+q} A}\ \textsc{withBorrow}$$

- (push)

$$\frac{\Gamma, x : A \vdash t : \&_p (A \otimes B)}{\Gamma, x : A \vdash \mathbf{push}\, t : (\&_p A) \otimes (\&_p B)}\ \textsc{push}$$

Then by induction on the premise we have: $\Gamma + \Gamma_1 \vdash t[t_1/x] : \&_p (A \otimes B)$ from which we construct the goal:

$$\frac{\Gamma + \Gamma_1 \vdash t[t_1/x] : \&_p (A \otimes B)}{\Gamma + \Gamma_1 \vdash \mathbf{push}\, t[t_1/x] : \&_p A \otimes \&_q A}\ \textsc{push}$$

- (pull)

$$\frac{\Gamma, x : A \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma, x : A \vdash \mathbf{pull}\, t : \&_p (A \otimes B)}\ \textsc{pull}$$

Then by induction on the premise we have: $\Gamma + \Gamma_1 \vdash t[t_1/x] : (\&_p A) \otimes (\&_p B)$ from which we construct the goal:

$$\frac{\Gamma + \Gamma_1 \vdash t[t_1/x] : (\&_p A) \otimes (\&_p B)}{\Gamma + \Gamma_1 \vdash \mathbf{pull}\, t[t_1/x] : \&_p (A \otimes B)}\ \textsc{pull}$$

- (newRef), (swapRef), (freezeRef), (readRef), (newArray), (readArray), (writeArray), (deleteArray) all trivial as they are atomic with substitution having no effect.

□

Lemma 9 (Graded substitution is admissible, extending Chapter 3)**.** *If* $[\Gamma_1] \vdash t_1 : A$ *and* $\Gamma_2, x : [A]_r \vdash t_2 : B$ *(where* $[\Gamma_1]$ *represents a context* Γ_1 *containing only graded assumptions) and* $\neg\mathsf{resourceAllocator}(t_1)$ *then* $\Gamma_2 + r \cdot \Gamma_1 \vdash t_2[t_1/x] : B$.

Proof. By induction on the typing derivation of t_2.

- (pr)

$$\frac{\Gamma_2', x : [A]_{r_2} \vdash t : A \qquad \neg\mathsf{resourceAllocator}(t)}{r_1 \cdot (\Gamma_2', x : [A]_{r_2}) \vdash [t] : \Box_{r_1} A}\ \textsc{pr}$$

where $r = r_1 * r_2$ and $t_2 = [t]$.

By induction on the premise then we have $\Gamma_2', r_2 \cdot \Gamma_1 \vdash t[t_1/x] : A$.

Then we construct the goal:

$$\frac{\Gamma_2', r_2 \cdot \Gamma_1 \vdash t[t_1/x] : A \qquad \neg\mathsf{resourceAllocator}(t[t_1/x])}{r_1 \cdot (\Gamma_2', r_2 \cdot \Gamma_1) \vdash [t[t_1/x]] : \Box_{r_1} A}\ \textsc{pr}$$

where $\neg\mathsf{resourceAllocator}(t[t_1/x])$ follows from $\neg\mathsf{resourceAllocator}(t)$ and $\neg\mathsf{resourceAllocator}(t_1)$ and which equals $r_1 \cdot \Gamma_2' + r_1 \cdot r_2 \cdot \Gamma_1 \vdash [t[t_1/x]] : \Box_{r_1} A$ satisfying the goal here.

- (share)

$$\frac{\Gamma_2, x : [A]_r \vdash t : *A}{\Gamma_2, x : [A]_r \vdash \mathbf{share}\, t : \Box_s A}\ \textsc{share}$$

where $B = \Box_s A$.

By induction on the premise then $\Gamma_2 + r \cdot \Gamma_1 \vdash t[t_1/x] : *A$, from which we build the conclusion:

$$\frac{\Gamma_2 + r \cdot \Gamma_1 \vdash t[t_1/x] : *A}{\Gamma_2 + r \cdot \Gamma_1 \vdash \mathbf{share}\, (t[t_1/x]) : \Box_s A}\ \textsc{share}$$

- (bind)

$$\frac{\Gamma_1', x : [A]_{r_1} \vdash t_1' : \Box_s A' \qquad \Gamma_2', y : *(\#A'), x : [A]_{r_2} \vdash t_2' : \Box_s B \qquad r \sqsubseteq 1}{\Gamma_1' + \Gamma_2', x : [A]_{r_1+r_2} \vdash \mathbf{clone}\, t_1' \,\mathbf{as}\, y \,\mathbf{in}\, t_2' : \Box_s B}\ \textsc{clone}'$$

with $r = r_1 + r_2$ without loss of generality (since any context not including x can instead have weakening applied to have either $r_1 = 0$ and/or $r_2 = 0$).

By induction on the premises, we have: (1) $\Gamma_1' + r_1 \cdot \Gamma_1 \vdash t_1'[t/x] : \Box_s A'$ (2) $(\Gamma_2' + r_2 \cdot \Gamma_1), y : *A' \vdash t_2'[t/x] : \Box_s B'$

Then we reconstruct the typing as:

$$\frac{\Gamma_1' + r_1 \cdot \Gamma_1 \vdash t_1'[t/x] : \Box_s A' \qquad (\Gamma_2' + r_2 \cdot \Gamma_1), y : *(\#A') \vdash t_2'[t/x] : \Box_s B'}{\Gamma_1' + \Gamma_2' + (r_1 + r_2) \cdot \Gamma_1 \vdash \mathbf{clone}\, t_1'[t/x] \,\mathbf{as}\, y \,\mathbf{in}\, t_2'[t/x] : \Box_s B'}\ \textsc{clone}$$

satisfying the goal.

- (withBorrow)

$$\frac{\Gamma_1', x : [A]_{r_1} \vdash t : *A' \qquad \Gamma_2', x : [A]_{r_2} \vdash f : \&_1 A' \multimap \&_1 B'}{(\Gamma_1' + \Gamma_2'), x : [A]_{r_1+r_2} \vdash \mathbf{withBorrow}\ f\ t : *B'}\ \textsc{withBorrow}$$

By induction on the premises, we have: (1) $\Gamma_1' + r_1 \cdot \Gamma_1 \vdash t[t_1/x] : *A'$. (2) $\Gamma_2' + r_2 \cdot \Gamma_1 \vdash f[t_1/x] : \&_1 A' \multimap \&_1 B'$.

From this we construct the goal:

$$\frac{\Gamma_1' + r_1 \cdot \Gamma_1 \vdash t[t_1/x] : *A' \qquad \Gamma_2' + r_2 \cdot \Gamma_1 \vdash f[t_1/x] : \&_1 A' \multimap \&_1 B'}{\Gamma_1' + \Gamma_2' + (r_1 + r_2) \cdot \Gamma_1 \vdash \mathbf{withBorrow}\ f[t_1/x]\ t : *B'}\ \textsc{withBorrow}$$

satisfying the goal.

- (split)

$$\frac{\Gamma, x : [A]_r \vdash t : \&_{p+q} A}{\Gamma, x : [A]_r \vdash \mathbf{split}\ t : \&_p A \otimes \&_q A}\ \textsc{withBorrow}$$

Then by induction on the premise we have: $\Gamma_1 + r \cdot \Gamma \vdash t[t_1/x] : B$ from which we construct the goal:

$$\frac{\Gamma_1 + r \cdot \Gamma \vdash t[t_1/x] : \&_{p+q} A}{\Gamma_1 + r \cdot \Gamma \vdash \mathbf{split}\ (t[t_1/x]) : \&_p A \otimes \&_q A}\ \textsc{withBorrow}$$

- (join)

$$\frac{\Gamma_1', x : [A]_{r_1} \vdash t_1' : \&_p A \qquad \Gamma_2', x : [A]_{r_2} \vdash t_2' : \&_q A \qquad p + q \leq 1}{(\Gamma_1' + \Gamma_2'), x : [A]_{(r_1+r_2)} \vdash \mathbf{join}\ t_1'\ t_2' : \&_{p+q} A}\ \textsc{withBorrow}$$

Then by induction on the premises we have: (1) $\Gamma_1' + r_1 \cdot \Gamma_1 \vdash t_1'[t_1/x] : \&_p A$ (2) $\Gamma_2' + r_2 \cdot \Gamma_1 \vdash t_2'[t_1/x] : \&_q A$ from which we construct the goal:

$$\frac{\Gamma_1' + r_1 \cdot \Gamma_1 \vdash t_1'[t_1/x] : \&_p A \qquad \Gamma_2' + r_2 \cdot \Gamma_1 \vdash t_2'[t_1/x] : \&_q A \qquad p + q \leq 1}{\Gamma_1' + \Gamma_2' + (r_1 + r_2) \cdot \Gamma_1 \vdash \mathbf{join}\ t_1'\ (t_2'[t_1/x]) : \&_{p+q} A}\ \textsc{withBorrow}$$

satisfying the goal.

- (push)

$$\frac{\Gamma, x : [A]_r \vdash t : \&_p (A \otimes B)}{\Gamma, x : [A]_r \vdash \mathbf{push}\, t : (\&_p A) \otimes (\&_p B)}\ \textsc{push}$$

Then by induction on the premise we have: $\Gamma + r \cdot \Gamma_1 \vdash t[t_1/x] : \&_p (A \otimes B)$ from which we construct the goal:

$$\frac{\Gamma + r \cdot \Gamma_1 \vdash t[t_1/x] : \&_p (A \otimes B)}{\Gamma + r \cdot \Gamma_1 \vdash \mathbf{push}\, t[t_1/x] : \&_p A \otimes \&_q A}\ \textsc{push}$$

- (pull)

$$\frac{\Gamma, x : [A]_r \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma, x : [A]_r \vdash \mathbf{pull}\, t : \&_p (A \otimes B)}\ \textsc{pull}$$

Then by induction on the premise we have: $\Gamma + r \cdot \Gamma_1 \vdash t[t_1/x] : (\&_p A) \otimes (\&_p B)$ from which we construct the goal:

$$\frac{\Gamma + r \cdot \Gamma_1 \vdash t[t_1/x] : (\&_p A) \otimes (\&_p B)}{\Gamma + r \cdot \Gamma_1 \vdash \mathbf{pull}\, t[t_1/x] : \&_p (A \otimes B)}\ \textsc{pull}$$

- (newRef), (swapRef), (freezeRef), (readRef), (newArray), (readArray), (writeArray), (deleteArray) all trivial as they are atomic with substitution having no effect.

□

C.3 TYPE SAFETY

C.3.1 Value lemmas

Lemma 10. *Value lemma*
Given $\Gamma \vdash v : A$ then, depending on the type, the shape of v can be inferred:

- *$A = A' \to B$ then $v = \lambda x.t$ or a partially applied primitive term p.*
- *$A = \square_r A'$ then $v = [v']$.*

- $A = A' \otimes B$ *then* $v = (v_1, v_2)$.
- $A = 1$ *then* $v = ()$.
- $A = *A'$ *then* $v = *v'$.
- $A = \&_p A'$ *then* $v = *v'$.
- $A = \mathbb{N}$ *then* $v = n$.
- $A = \mathbb{F}$ *then* $v = f$.
- $A = \mathsf{Ref}_{id}\ A'$ *then* $v = ref$.
- $A = \mathsf{Array}_{id}\ \mathbb{F}$ *then* $v = a$.
- $A = \exists id.A'$ *then* $v = \textbf{\textit{pack}}\ \langle id', v' \rangle$.

Proof. Recall that the value terms sub-grammar is:

$$v ::= (v_1, v_2) \mid () \mid *v \mid [v] \mid \lambda x.t \mid i \mid ref \mid a \mid p \mid \textbf{pack}\ \langle id', v' \rangle$$
(value terms sub-grammar)

where p are partially-applied primitives:

$$\begin{aligned} p ::= {} & \textbf{newRef} \mid \textbf{swapRef} \mid \textbf{swapRef}\,(*ref) \\ & \mid \textbf{freezeRef} \mid \textbf{readRef} \\ & \mid \textbf{newArray} \mid \textbf{readArray} \mid \textbf{readArray}\,(*a) \\ & \mid \textbf{writeArray} \mid \textbf{writeArray}\,(*a) \mid \textbf{writeArray}\,(*a)\,n \mid \textbf{deleteArray} \end{aligned}$$

We then proceed by case analysis on the type A to match the structure of the lemma. In each case we must consider what possible values can be assigned the type A and by which rules.

In all cases, there exists additional derivations based on dereliction and approximation, e.g., for the case where $A = A' \to B$:

$$\frac{\Gamma, x : A'' \vdash t : A' \to B}{\Gamma, x : [A'']_1 \vdash t : A' \to B}\ \textsc{der}$$

$$\frac{\Gamma, y : [A'']_r, \Gamma' \vdash t : A' \to B \quad r \sqsubseteq s}{\Gamma, y : [A'']_s, \Gamma' \vdash t : A' \to B}\ \textsc{approx}$$

In all of these cases we can apply induction on the premise to get the result since the term is preserved between the premise and the conclusion.

We elide handling this separately each time in the cases that follow as the reasoning through dereliction and approximation is the same each time.

- $A = A' \to B$ then there are two classes of possible typing:

– Abstract term:

$$\frac{\Gamma, x : A' \vdash t : B}{\Gamma \vdash \lambda x.t : A' \to B}\ \textsc{abs}$$

thus $v = \lambda x.t$ as in the lemma statement.

– Primitive term p formed by an application of zero or more values to a primitive operation, of which there are then twelve possibilities:

1.

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{newRef} : A \multimap \exists id.*(\mathsf{Ref}_{id}\ A)}\ \textsc{newRef}$$

thus $v = \mathbf{newRef}$

2.

$$\frac{p \equiv 1 \ \vee \ p \equiv *}{0 \cdot \Gamma \vdash \mathbf{swapRef} : \&_p(\mathsf{Ref}_{id}\ A) \multimap A \multimap A \otimes \&_p(\mathsf{Ref}_{id}\ A)}\ \textsc{swapRef}$$

thus $v = \mathbf{swapRef}$

3.

$$\frac{\dfrac{\dfrac{}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash ref : Res_{id}\ A}\ \textsc{ref}}{0 \cdot \Gamma, ref : \mathsf{Ref}_{id}\ A \vdash *ref : \&_1 \mathsf{Ref}_{id}\ A}\ \textsc{nec}}{0 \cdot \Gamma, ref : \mathsf{Ref}_{id}\ A \vdash \mathbf{swapRef}\ (*ref) : A \otimes \&_1(\mathsf{Ref}_{id}\ A)}\ \textsc{app}$$

thus $v = \mathbf{swapRef}\ (*ref)$

4.

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{freezeRef} : *(\mathsf{Ref}_{id}\ A) \multimap A}\ \textsc{freezeRef}$$

thus $v = \mathbf{freezeRef}$

5.

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{readRef} : \&_p(\mathsf{Ref}_{id}\ (\Box_{r+1}A)) \multimap A \otimes \&_p(\mathsf{Ref}_{id}\ (\Box_r A))}\ \textsc{readRef}$$

thus $v = \mathbf{readRef}$

6.

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{newArray} : \mathbb{N} \multimap \exists id.*(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{newArray}$$

thus $v = \mathbf{newArray}$

7.

$$\frac{}{0 \cdot \Gamma \vdash \mathbf{readArray} : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{readArray}$$

thus $v = \mathbf{readArray}$

8.

$$\dfrac{\dfrac{\overline{0\cdot\Gamma, ref : Res_{id}\ A \vdash ref : Res_{id}\ A}^{\text{REF}}}{0\cdot\Gamma, a : \mathsf{Array}_{id}\ A \vdash *a : \&_1\mathsf{Array}_{id}\ \mathbb{F}}^{\text{NEC}}}{0\cdot\Gamma, a : \mathsf{Array}_{id}\ A \vdash \mathbf{readArray}\ (*a) : \mathbb{N} \multimap \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\ \mathbb{F})}^{\text{APP}}$$

thus $v = \mathbf{readArray}\ (*a)$

9.

$$\dfrac{p \equiv 1 \ \vee\ p \equiv *}{0\cdot\Gamma \vdash \mathbf{writeArray} : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap \&_p(\mathsf{Array}_{id}\ \mathbb{F})}^{\text{WRITEARRAY}}$$

thus $v = \mathbf{writeArray}$

10.

$$\dfrac{\dfrac{\overline{0\cdot\Gamma, ref : Res_{id}\ A \vdash ref : Res_{id}\ A}^{\text{REF}}}{0\cdot\Gamma, a : \mathsf{Array}_{id}\ A \vdash *a : \&_1\mathsf{Array}_{id}\ \mathbb{F}}^{\text{NEC}}}{0\cdot\Gamma, a : \mathsf{Array}_{id}\ A \vdash \mathbf{writeArray}\ (*a) : \mathbb{N} \multimap \mathbb{F} \multimap \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\ \mathbb{F})}^{\text{APP}}$$

thus $v = \mathbf{writeArray}\ (*a)$

11.

$$\dfrac{\dfrac{\dfrac{\overline{0\cdot\Gamma, ref : Res_{id}\ A \vdash ref : Res_{id}\ A}^{\text{REF}}}{0\cdot\Gamma, a : \mathsf{Array}_{id}\ A \vdash *a : \&_1\mathsf{Array}_{id}\ \mathbb{F}}^{\text{NEC}}}{0\cdot\Gamma, a : \mathsf{Array}_{id}\ A \vdash \mathbf{writeArray}\ (*a) : \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\ \mathbb{F})}^{\text{APP}} \quad \emptyset \vdash n : \mathbb{N}}{0\cdot\Gamma, a : \mathsf{Array}_{id}\ A \vdash \mathbf{writeArray}\ (*a)\ n : \mathbb{F} \multimap \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\ \mathbb{F})}^{\text{APP}}$$

thus $v = \mathbf{writeArray}\ (*a)\ n$

12.

$$\dfrac{}{0\cdot\Gamma \vdash \mathbf{deleteArray} : *(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathsf{unit}}^{\text{DELARRAY}}$$

thus $v = \mathbf{deleteArray}$

- $A = \Box_r A'$ then there is only one possible non-dereliction/non-approximation typing of a value at that type:

$$\dfrac{\Gamma \vdash v' : A'}{r\cdot\Gamma \vdash [v'] : \Box_r A'}^{\text{PR}}$$

thus $v = [v']$ as in the lemma statement.

- $A = A' \otimes B$ then there is only one possible non-dereliction/non-approximation typing of a value at that type:

$$\dfrac{\Gamma_1 \vdash v_1 : A' \quad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : A' \otimes B}\otimes_I$$

thus $v = (v_1, v_2)$ as in the lemma statement.

- $A = 1$ then there is only one possible non-dereliction/non-approximation typing of a value at that type:

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}} \; 1_I$$

thus $v = ()$ as in the lemma statement.

- $A = *A'$ then there is only one possible non-dereliction/non-approximation typing of a value at that type:

$$\frac{\emptyset \vdash v' : A'}{0 \cdot \Gamma \vdash *v' : *A'} \; \textsc{nec}$$

thus $v = *t$ as in the lemma statement.

- $A = \&_p A'$ then there is only one possible non-dereliction/non-approximation typing of a value at that type:

$$\frac{\emptyset \vdash v' : A'}{0 \cdot \Gamma \vdash *v' : \&_1 A'} \; \textsc{nec}$$

thus $v = *v'$ as in the lemma statement.

- $A = \mathbb{N}$ then $v = n$ Trivial case on typing of constants which is elided in this paper for brevity (but covered by the core type theory of Granule for example).

- $A = \mathbb{F}$ then $v = f$ Trivial case on typing of constants which is elided in this paper for brevity (but covered by the core type theory of Granule for example).

- $A = \mathsf{Ref}_{id}\ A$ then $v = ref$ then the only possible typing that is a value is given by:

$$\frac{}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash ref : Res_{id}\ A} \; \textsc{ref}$$

- $A = \mathsf{Array}_{id}\ \mathbb{F}$ then $v = a$ then the only possible typing that is a value is given by:

$$\frac{}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash ref : Res_{id}\ A} \; \textsc{ref}$$

(and since the only type of arrays is $\mathbb{F}$ currently).

- $A = \exists id.A'$ then there is only one possible non-dereliction/non-approximation typing of a value at that type:

$$\frac{\Gamma \vdash v' : A \quad id \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathbf{pack}\ \langle id', t \rangle : \exists id.A[id/id']} \; \textsc{pack}$$

thus $v = \mathbf{pack}\ \langle id', v' \rangle$ as in the lemma statement.

□

Lemma 11 (Closed value lemma)**.** *Given $\Gamma \vdash v : A$ where A does not comprise a function type, then there exists a runtime only context γ such that $\gamma \vdash v : A$, i.e., v is closed with respect to normal variables.*

Proof. Similar to the value lemma proof structure, and where:

- $A = A' \to B'$ is excluded by the lemma statement.
- $A = \Box_r A'$ then $v = [v']$ by induction.
- $A = A' \otimes B$ then $v = (v_1, v_2)$ by induction.
- $A = 1$ then $v = ()$ is closed.
- $A = *A'$ then $v = *v'$ by induction.
- $A = \&_p A'$ then $v = *v'$ by induction.
- $A = \mathbb{N}$ then $v = n$ is closed.
- $A = \mathbb{F}$ then $v = f$ is closed.
- $A = \mathsf{Ref}_{id}\ A'$ then $v = ref$ has only a runtime context.
- $A = \mathsf{Array}_{id}\ \mathbb{F}$ then $v = a$ has only a runtime context.
- $A = \exists id.A'$ then $v = \mathbf{pack}\ \langle id', v' \rangle$ by induction.

□

Lemma 12 (Unique value lemma)**.** *Given $\Gamma \vdash *v : *A$ then, depending on the type A, the shape of v can be inferred:*

- $A = A' \otimes B'$ *then* $v = (v_1, v_2)$.
- $A = \mathsf{Ref}_{id}\ A$ *then* $v = ref$.
- $A = \mathsf{Array}_{id}\ \mathbb{F}$ *then* $v = a$.

and there are no other possible typings for $*v$. Furthermore, $\exists \Gamma', \gamma$ such that $0 \cdot \Gamma', \gamma \vdash *v : *A$, i.e., it can be type in a runtime context only.

Proof. There are only three possible typings for $*v$.

- $A = A' \otimes B'$ where there is only one possible non-dereliction/non-approximation typing of a value at the type $*(A' \otimes B')$:

$$\dfrac{\dfrac{\dfrac{}{0 \cdot \Gamma_1, \gamma_1 \vdash *v_1 : *A'}\ \text{INDUCTION.} \quad \dfrac{}{0 \cdot \Gamma_2, \gamma_2 \vdash *v_2 : *B'}\ \text{INDUCTION.}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + \gamma_1 + \gamma_2 \vdash (*v_1, *v_2) : *A' \otimes *B'}\ \otimes_I}{0 \cdot (\Gamma_1 + \Gamma_2) + \gamma_1 + \gamma_2 \vdash *(v_1, v_2) : *(A' \otimes B')}\ \text{PULL}$$

  thus $v = (v_1, v_2)$ as in the lemma statement and $\Gamma' = \Gamma_1 + \Gamma_2$ and $\gamma = \gamma_1 + \gamma_2$.

- $A = \mathsf{Ref}_{id}\ A$ where there is only one possible non-dereliction/non-approximation typing of a value at the type $*(\mathsf{Ref}_{id}\ A)$:

$$\frac{}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash *ref : *(Res_{id}\ A)}\ {*\textbf{REF}^*}$$

  thus $v = ref$ as in the lemma statement and $\Gamma' = \Gamma$ and $\gamma = ref : \mathsf{Ref}_{id}\ A$.

- $A = \mathsf{Array}_{id}\ \mathbb{F}$ where there is only one possible non-dereliction/non-approximation typing of a value at the type $*(\mathsf{Array}_{id}\ \mathbb{F})$:

$$\frac{}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash *ref : *(Res_{id}\ A)}\ {*\textbf{REF}^*}$$

  thus $v = a$ as in the lemma statement (and since the only type of arrays is $\mathbb{F}$ currently) and $\Gamma' = \Gamma$ and $\gamma = a : \mathsf{Array}_{id}\ A$.

□

C.3.2 Progress proof

Theorem 7 (Progress). *Given $\Gamma \vdash t : A$, then t is either a value, or for all grades s and contexts Γ_0 then if $H \bowtie \Gamma_0 + s \cdot \Gamma$ there exists a heap H' and term t' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$.*

Proof. By induction on typing.

- (var)

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textbf{VAR}$$

  Here, $H \bowtie s \cdot [\Gamma], x : [A]_s$, which by inversion of heap compatibility implies that $H = H', x \mapsto_r v$ and $\exists r'.\ s + r' \equiv r$. Hence, we can reduce by the following rule:

$$\frac{\exists r'.\ s + r' \sqsubseteq r}{H, x \mapsto_r v \vdash x \rightsquigarrow_s H, x \mapsto_r v \vdash v}\ \rightsquigarrow_{\textbf{VAR}}$$

  where $\exists r'.\ s + r' \equiv r$ implies $\exists r'.\ s + r' \sqsubseteq r$ by reflexivity, satisfying the premise.

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textbf{ABS}$$

  A value.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \textbf{APP}$$

  By induction on the first premise, there are two possibilities.

1. t_1 is a value and therefore by the value lemma there are a number of choices:
   - $t_1 = \lambda x.t_1'$. Therefore we induct on the second premise providing two possibilities:
     * If $t_2 = v$ for some value v, then we can reduce by the following rule:

$$\frac{y\#\{H, v, t\}}{H \vdash (\lambda x.t)\, v \rightsquigarrow_s H, y \mapsto_s v \vdash t[y/x]} \rightsquigarrow_\beta$$

     * If t_2 is not a value, then there exists heap H', term t_1' and context Γ' such that $H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash v\, t_2 \rightsquigarrow_s H' \vdash v\, t_2'} \rightsquigarrow_{\text{APPR}}$$

   - $t_1 = \mathbf{newRef}$

     Therefore we induct on the second argument:

     * t_2 is a value v and thus we can reduce:

$$\frac{ref\#H \qquad id\#H}{H \vdash \mathbf{newRef}\, v \rightsquigarrow_s H, ref \mapsto_1 id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{pack}\, \langle id, *ref \rangle} \rightsquigarrow_{\text{NEWREF}}$$

     * t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{newRef}\, t_2 \rightsquigarrow_s H' \vdash \mathbf{newRef}\, t_2'} \rightsquigarrow_{\text{PRIM}}$$

   - $t_1 = \mathbf{swapRef}$
     * t_2 is a value and therefore by the value lemma on $\&_p(\mathsf{Ref}_{id}\ A)$ (Lemma 10) $t_2 = *ref$ and thus $t_1\, t_2 = \mathbf{swapRef}\,(*ref)$ which is also a value.
     * t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{swapRef}\, t_2 \rightsquigarrow_s H' \vdash \mathbf{swapRef}\, t_2'} \rightsquigarrow_{\text{PRIM}}$$

   - $t_1 = \mathbf{swapRef}\,(*ref)$

     Therefore we induct on the second argument:

* t_2 is a value v and thus we can reduce:

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{swapRef}\,(*ref)\,v' \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v') \vdash v}\ \rightsquigarrow_{\text{SWAPREF}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{swapRef}\,(*ref)\,t_2 \rightsquigarrow_s H' \vdash \mathbf{swapRef}\,(*ref)\,t_2'}\ \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{freezeRef}$

Therefore we induct on the second argument:

* t_2 is a value which by the value and unique value lemmas has the form $*ref$, and thus we can reduce:

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{freezeRef}\,(*ref) \rightsquigarrow_s H \vdash v}\ \rightsquigarrow_{\text{FREEZEREF}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{freezeRef}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{freezeRef}\,t_2'}\ \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{readRef}$

Therefore we induct on the second argument:

* t_2 is a value which by the value and unique value lemmas has the form $*ref$, and thus we can reduce:

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_{r+1}) \vdash \mathbf{readRef}\,(*ref) \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_r) \vdash (v, *ref)}\ \rightsquigarrow_{\text{READREF}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{readRef}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{readRef}\,t_2'}\ \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{newArray}$ therefore $A = \mathbb{N}$

Therefore we induct on the second argument:

* t_2 is a value and therefore by the value lemma (Lemma 10) $t_2 = n$ and thus the typing is:

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathbf{newArray}\,n : *(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{newArray}$$

with $H \bowtie (\Gamma_0 + s \cdot \Gamma)$.

Thus there is a reduction as follows:

$$\frac{ref\#H \qquad id\#H}{H \vdash \mathbf{newArray}\, n \rightsquigarrow_s H, ref \mapsto_1 id, id \mapsto \mathsf{init} \vdash \mathbf{pack}\, \langle id, *ref \rangle} \rightsquigarrow_{\text{NEWARRAY}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{newArray}\, t_2 \rightsquigarrow_s H' \vdash \mathbf{newArray}\, t_2'} \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{readArray}$ therefore $A = \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})$

* t_2 is a value and therefore by the value lemma on $\&_p(\mathsf{Array}_{id}\ \mathbb{F})$ (Lemma 10) $t_2 = *a$ and thus $t_1\, t_2 = \mathbf{readArray}\, (*a)$ which is also a value.

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{readArray}\, t_2 \rightsquigarrow_s H' \vdash \mathbf{readArray}\, t_2'} \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \mathbf{readArray}\, (*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})$

* t_2 is a value and therefore by the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{N}$ (Lemma 10) implies $t_2 = n$ and thus the typing is refined at runtime as follows:

$$\frac{\overline{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (*a) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}^{\text{NEC}} \qquad \Gamma_2 \vdash n : \mathbb{N}}{[\Gamma_1] + \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{readArray}\, (*a)\, n : \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})} \text{READARRAY}$$

with $H' \bowtie \Gamma_0 + s \cdot ([\Gamma_1] + \Gamma_2), a : \mathsf{Array}_{id}\ \mathbb{F}$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_p id, id \mapsto \mathbf{arr}$.

Then there is a reduction as follows:

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{arr}[n] = v \vdash \mathbf{readArray}\, (*ref)\, n \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{arr}[n] = v \vdash (v, *ref)} \rightsquigarrow_{\text{READARRAY}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{readArray}\, (*a)\, t_2 \rightsquigarrow_s H' \vdash \mathbf{readArray}\, (*a)\, t_2'} \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \textbf{writeArray}$ therefore $A = \&_1(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap \&_1(\mathsf{Array}_{id}\ \mathbb{F})$

  * t_2 is a value therefore by the value lemma (Lemma 10) $t_2 = (*a)$ and thus $t_1\ t_2 = \textbf{writeArray}\ (*a)$ which is also a value.

  * t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \textbf{writeArray}\ t_2 \rightsquigarrow_s H' \vdash \textbf{writeArray}\ t_2'}\ \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \textbf{writeArray}\ (*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \multimap \&_1(\mathsf{Array}_{id}\ \mathbb{F})$

  * t_2 is a value therefore by the value lemma (Lemma 10) $t_2 = n$ and thus $t_1\ t_2 = \textbf{writeArray}\ (*a)\ n$ which is also a value.

  * t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \textbf{writeArray}\ (*a)\ t_2 \rightsquigarrow_s H' \vdash \textbf{writeArray}\ (*a)\ t_2'}\ \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \textbf{writeArray}\ (*a)\ n$ therefore $A = \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\ \mathbb{F})$

  * t_2 is a value and therefore the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{F}$ (Lemma 10) implies $t_2 = f$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \text{REF}}{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : \&_1(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{NEC} \qquad \Gamma_2 \vdash n : \mathbb{N} \quad \Gamma_3 \vdash f : \mathbb{F}}{[\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \textbf{writeArray}\ (*a)\ n\ f : \&_1(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{WRITEARRAY}$$

  with $H' \bowtie (\Gamma_0 + [\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_p id, id \mapsto \textbf{arr}$.

  Then there is a reduction as follows:

$$\frac{}{H, ref \mapsto_p id, id \mapsto \textbf{arr} \vdash \textbf{writeArray}\ (*ref)\ n\ v \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \textbf{arr}[n] = v \vdash *ref}\ \rightsquigarrow_{\text{WRITEARRAY}}$$

  * t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \textbf{writeArray}\ (*a)\ n\ t_2 \rightsquigarrow_s H' \vdash \textbf{writeArray}\ (*a)\ n\ t_2'}\ \rightsquigarrow_{\text{PRIM}}$$

– $t_1 = \textbf{deleteArray}$ therefore $A = *(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathsf{unit}$

* t_2 is a value and therefore by the value lemma (Lemma 10) $t_2 = *a$ thus the typing is refined at runtime as follows:

$$\dfrac{\dfrac{[\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \textsc{ref}}{[\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : *(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{nec}}{[\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \textbf{deleteArray}\ *a : \mathsf{unit}}\ \textsc{delArray}$$

with $H' \bowtie (\Gamma_0 + [\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_p id, id \mapsto \textbf{arr}$.

There there is a reduction as follows:

$$\dfrac{}{H, ref \mapsto_p id, id \mapsto \textbf{arr} \vdash \textbf{deleteArray}\ (*ref) \rightsquigarrow_s H \vdash ()}\ \rightsquigarrow_{\textsc{deleteArray}}$$

* t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\dfrac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \textbf{deleteArray}\ t_2 \rightsquigarrow_s H' \vdash \textbf{deleteArray}\ t_2'}\ \rightsquigarrow_{\textsc{prim}}$$

2. Otherwise, by induction on the premise we then have that there exists heap H', term t_1', and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\dfrac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash t_1\ t_2 \rightsquigarrow_s H' \vdash t_1'\ t_2}\ \rightsquigarrow_{\textsc{appL}}$$

- (pr)

$$\dfrac{\Gamma \vdash t : A \qquad \neg \mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t]_r : \Box_r A}\ \textsc{pr}$$

with heap compatibility $H \bowtie \Gamma_0 + s \cdot (r \cdot \Gamma)$, which by associativity of $*$ is equal to $H \bowtie \Gamma_0 + (s * r) \cdot \Gamma$. Then, by induction on the premise (with $s' = s * r$), there are two possibilities:

1. If t is a value, say v, then $[t]$ is also a value, $[v]$.
2. Otherwise, there exists H' and t' such that $H \vdash t \rightsquigarrow_{s*r} H' \vdash t'$, from which we can then reduce by the following rule:

$$\dfrac{H \vdash t \rightsquigarrow_{s*r} H' \vdash t'}{H \vdash [t]_r \rightsquigarrow_s H' \vdash [t']_r}\ \rightsquigarrow_{\Box}$$

- (elim)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

By induction on the first premise, there are two possibilities.

1. t_1 is a value and therefore by the value lemma $t_1 = [v]$. This refines the typing as follows:

$$\frac{\dfrac{\Gamma_1 \vdash v : A}{r \cdot \Gamma_1 \vdash [v]_r : \Box_r A}\ \textsc{pr} \qquad \Gamma_2, x : \Box_r A \vdash t_2 : B}{r \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = [v]_r\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

Therefore we can reduce by the following rule:

$$\frac{y\#\{H, v, t\}}{H \vdash \mathbf{let}\,[x] = [v]_r\,\mathbf{in}\,t \ \rightsquigarrow_s\ H, y \mapsto_{s*r} v \vdash t[y/x]}\ \rightsquigarrow_{\Box\beta}$$

2. Otherwise by induction on the premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \ \rightsquigarrow_s\ H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \ \rightsquigarrow_s\ H' \vdash t_1'}{H \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \ \rightsquigarrow_s\ H' \vdash \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\textsc{let}\Box}$$

- (der)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

Goal achieved immediately by induction on the premise.

- (approx)

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx}$$

Goal achieved immediately by induction on the premise.

- (pairIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_I$$

By induction on the premises there are three possible cases.

1. If both t_1 and t_2 are values then (t_1, t_2) is also a value.

2. If only t_1 is a value then by induction on the second premise we then have that there exists heap H', term t_2' and context Γ' such that $H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash (v, t_2) \rightsquigarrow_s H' \vdash (v, t_2')} \rightsquigarrow_{\otimes \mathrm{R}}$$

3. If neither t_1 nor t_2 are values then by induction on the first premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash (t_1, t_2) \rightsquigarrow_s H' \vdash (t_1', t_2)} \rightsquigarrow_{\otimes \mathrm{L}}$$

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C} \otimes_E$$

By induction on the first premise, there are two possibilities.

1. t_1 is a value and therefore by the value lemma $t_1 = (v_1, v_2)$. This refines the typing as follows:

$$\frac{\dfrac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : A \otimes B} \otimes_I \qquad \Gamma_3, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \mathbf{let}\,(x, y) = (v_1, v_2)\,\mathbf{in}\,t_2 : C} \otimes_E$$

Therefore, we can reduce by the following rule:

$$\frac{x' \# \{H, v_1, v_2, t\} \qquad y' \# \{H, v_1, v_2, t\}}{H \vdash \mathbf{let}\,(x, y) = (v_1, v_2)\,\mathbf{in}\,t \rightsquigarrow_s H, x' \mapsto_s v_1, y' \mapsto_s v_2 \vdash t[y'/y][x'/x]} \rightsquigarrow_{\otimes \beta}$$

2. Otherwise, by induction on the premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,(x, y) = t_1'\,\mathbf{in}\,t_2} \rightsquigarrow_{\mathrm{LET}\otimes}$$

- (unitIntro)

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I$$

A value.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ () = t_1\ \mathbf{in}\ t_2 : B}\ 1_E$$

By induction on the first premise, there are two possibilities.

1. t is a value and therefore by the value lemma $t = ()$. Therefore we can reduce by the following rule:

$$\frac{}{H \vdash \mathbf{let}\ () = ()\ \mathbf{in}\ t \rightsquigarrow_s H \vdash t}\ \rightsquigarrow\beta_{\textsc{unit}}$$

2. Otherwise by induction on the premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\ () = t_1\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\ () = t_1'\ \mathbf{in}\ t_2}\ \rightsquigarrow_{\textsc{letunit}}$$

- (returnGen)

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathbf{share}\, t : \Box_r A}\ \textsc{share}$$

By induction on the premise, there are two possibilities.

1. t is a value and therefore by the value lemma $t = *v$. Therefore we can reduce by the following rule:

$$\frac{\mathsf{dom}(H) \equiv \mathsf{refs}(v)}{H, H' \vdash \mathbf{share}\ (*v) \rightsquigarrow_s ([H]_0), H' \vdash [v]}\ \rightsquigarrow_{\textsc{share}\beta}$$

2. Otherwise by induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{share}\, t \rightsquigarrow_s H' \vdash \mathbf{share}\, t'}\ \rightsquigarrow_{\textsc{share}}$$

- (bindGen)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : *A \vdash t_2 : \Box_r B \qquad 1 \sqsubseteq r}{\Gamma_1 + \Gamma_2 \vdash \mathbf{clone'}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 : \Box_r B}\ \textsc{clone}'$$

By induction on the first premise, there are two possibilities.

1. t_1 is a value and therefore by the value lemma $t_1 = [v]$. This refines the typing as follows:

$$\frac{\dfrac{\Gamma_1 \vdash v : A}{r \cdot \Gamma_1 \vdash [v] : \Box_r A}\,\text{PR} \qquad \Gamma_2, x : *A \vdash t_2 : \Box_r B}{r \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{clone}\,[v]\,\mathbf{as}\,x\,\mathbf{in}\,t_2 : \Box_r B}\,\text{CLONE}'$$

   Therefore we can reduce using the following rule (with $t = t_2$):

$$\frac{\mathsf{dom}(H') \equiv \mathsf{refs}(v) \qquad (H'', \theta, \overline{id}) = \mathsf{copy}(H') \qquad y\#\{H, v, t\}}{H, H' \vdash \mathbf{clone}\,[v]_r\,\mathbf{as}\,x\,\mathbf{in}\,t \;\leadsto_s\; H, H', H'', y{\mapsto_s}\mathbf{pack}\,\langle \overline{id}, *(\theta(v))\rangle \vdash t[y/x]}\,\leadsto_{\text{CLONE}\beta}$$

2. Otherwise by induction on the premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \;\leadsto_s\; H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \;\leadsto_s\; H' \vdash t_1'}{H \vdash \mathbf{clone}\,t_1\,\mathbf{as}\,x\,\mathbf{in}\,t_2 \;\leadsto_s\; H' \vdash \mathbf{clone}\,t_1'\,\mathbf{as}\,x\,\mathbf{in}\,t_2}\,\leadsto_{\text{CLONE}}$$

- (withBorrow)

$$\frac{\Gamma_1 \vdash t_1 : *A \qquad \Gamma_2 \vdash t_2 : \&_1 A \multimap \&_1 B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{withBorrow}\;t_1\;t_2 : *B}\,\text{WITHBORROW}$$

  By induction on the first premise, there are two possibilities.

1. t_1 is a value and therefore by the value lemma $t_1 = (\lambda x.t)$. Then, again, we have two possibilities:
   - t_2 is also a value, and therefore by the value lemma $t_2 = (*v)$ for some v. Then we can reduce by the following rule:

$$\frac{y\#\{H, v, t\}}{H \vdash \mathbf{withBorrow}\;(\lambda x.t)\;(*v) \;\leadsto_s\; H, y{\mapsto_s}(*v) \vdash \mathbf{unborrow}\;t[y/x]}\,\leadsto_{\text{WITH\&}}$$

   - t_2 is not a value. Then, by induction on the premise we have that there exists heap H', term t_2' and context Γ' such that $H \vdash t_1 \;\leadsto_s\; H' \vdash t_2'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_2 \;\leadsto_s\; H' \vdash t_2'}{H \vdash \mathbf{withBorrow}\;(\lambda x.t_1)\;t_2 \;\leadsto_s\; H' \vdash \mathbf{withBorrow}\;(\lambda x.t_1)\;t_2'}\,\leadsto_{\text{WITH\&R}}$$

2. t_1 is not a value. Then, by induction on the premise we have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \textbf{withBorrow}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \textbf{withBorrow}\ t_1'\ t_2}\ \rightsquigarrow_{\text{WITH\&L}}$$

- (split)

$$\frac{\Gamma \vdash t : \&_p A}{\Gamma \vdash \textbf{split}\ t : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\ \text{SPLIT}$$

By induction on the premise, there are two cases.

1. If t is a value, then by the value lemma and the unique value lemma there are three possibilities for the form of t.
   - t could have the form $(*ref)$. This refines the typing as follows:

$$\frac{[\Gamma], ref : \mathsf{Ref}_{id}\ A \vdash (*ref) : \&_p(\mathsf{Ref}_{id}\ A)\ \text{REF + NEC}}{[\Gamma], ref : \mathsf{Ref}_{id}\ A \vdash \textbf{split}\ (*ref) : \&_{\frac{p}{2}}(\mathsf{Ref}_{id}\ A) \otimes \&_{\frac{p}{2}}(\mathsf{Ref}_{id}\ A)}\ \text{SPLIT}$$

     Then we can reduce by the following rule:

$$\frac{ref_1 \# H \qquad ref_2 \# H}{H, ref \mapsto_p id, id \mapsto v \vdash \textbf{split}\ (*ref) \rightsquigarrow_s H, ref_1 \mapsto_{\frac{p}{2}} id, ref_2 \mapsto_{\frac{p}{2}} id, id \mapsto v \vdash (*ref_1, *ref_2)}\ \rightsquigarrow_{\text{SPLITREF}}$$

   - t could have the form $(*a)$. This refines the typing as follows:

$$\frac{[\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (*a) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})\ \text{REF + NEC}}{[\Gamma], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \textbf{split}\ (*a) : \&_{\frac{p}{2}}(\mathsf{Array}_{id}\ \mathbb{F}) \otimes \&_{\frac{p}{2}}(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{SPLIT}$$

     Then we can reduce by the following rule:

$$\frac{a_1 \# H \qquad a_2 \# H}{H, a \mapsto_p id, id \mapsto \textbf{arr} \vdash \textbf{split}\ (*a) \rightsquigarrow_s H, a_1 \mapsto_{\frac{p}{2}} id, a_2 \mapsto_{\frac{p}{2}} id, id \mapsto \textbf{arr} \vdash (*a_1, *a_2)}\ \rightsquigarrow_{\text{SPLITARR}}$$

   - t could have the form $(*(v, w))$. There are two possible typings in this case:

$$\dfrac{\dfrac{\dfrac{\gamma_1 \vdash v_1 : A' \quad \gamma_2 \vdash v_2 : B}{\gamma_1 + \gamma_2 \vdash (v_1, v_2) : A' \otimes B}\otimes_I}{\gamma_1 + \gamma_2 \vdash *(v_1, v_2) : *(A' \otimes B)}\textsc{nec}}{\gamma_1 + \gamma_2 \vdash \mathbf{split}\ (*(v_1, v_2)) : (\&_{\frac{1}{2}}(A' \otimes B) \otimes \&_{\frac{1}{2}}(A' \otimes B))}\textsc{split}$$

In either instance, we can reduce by the following rule:

$$\dfrac{\begin{array}{c} H \vdash \mathbf{split}\ (*v) \rightsquigarrow_s H' \vdash (*v_1, *v_2) \\ H' \vdash \mathbf{split}\ (*w) \rightsquigarrow_s H'' \vdash (*w_1, *w_2) \end{array}}{H \vdash \mathbf{split}\ (*(v, w)) \rightsquigarrow_s H'' \vdash (*(v_1, w_1), *(v_2, w_2))} \rightsquigarrow_{\textsc{split}\otimes}$$

2. If t is not a value, then by induction on the premise we have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\dfrac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{split}\ t \rightsquigarrow_s H' \vdash \mathbf{split}\ t'} \rightsquigarrow_{\textsc{split}}$$

- (join)

$$\dfrac{\Gamma_1 \vdash t_1 : \&_p A \qquad \Gamma_2 \vdash t_2 : \&_q A \qquad p + q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \mathbf{join}\ t_1\ t_2 : \&_{p+q} A}\textsc{join}$$

By induction on the first premise, there are two cases.

1. If t_1 is a value, then first we should consider whether t_2 is also a value.
   - If t_2 is also a value, then by the value lemma and the unique value lemma there are three possibilities for the form of t_1.
     a) t_1 could have the form $(*ref_1)$. This refines the typing as follows:

$$\dfrac{\begin{array}{c} [\Gamma], ref_1 : \mathsf{Ref}_{id}\ A \vdash (*ref_1) : \&_p(\mathsf{Ref}_{id}\ A)\ \textsc{ref+nec} \\ [\Gamma], ref_2 : \mathsf{Ref}_{id}\ A \vdash (*ref_2) : \&_q(\mathsf{Ref}_{id}\ A)\ \textsc{ref+nec} \end{array}}{[\Gamma], ref_1 : \mathsf{Ref}_{id}\ A, ref_2 : \mathsf{Ref}_{id}\ A \vdash \mathbf{join}\ (*ref_1)\ (*ref_2) : \&_{p+q}(\mathsf{Ref}_{id}\ A)}\textsc{join}$$

Note that the typing in this case restricts t_2 to also have the form $*ref_2$. Then we can reduce by the following rule:

$$\dfrac{ref \# H}{H, ref_1 \mapsto_p id, ref_2 \mapsto_q id, id \mapsto v \vdash \mathbf{join}\ (*ref_1)\ (*ref_2) \rightsquigarrow_s H, ref \mapsto_{(p+q)} id, id \mapsto v \vdash *ref} \rightsquigarrow_{\textsc{joinRef}}$$

b) t_1 could have the form $(*a_1)$. This refines the typing as follows:

$$\frac{\begin{array}{c}[\Gamma], a_1 : \mathsf{Array}_{id}\ \mathbb{F} \vdash (*a_1) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})\ \textsc{ref+nec} \\ [\Gamma], a_2 : \mathsf{Array}_{id}\ \mathbb{F} \vdash (*a_2) : \&_q(\mathsf{Array}_{id}\ \mathbb{F})\ \textsc{ref+nec}\end{array}}{[\Gamma], a_1 : \mathsf{Array}_{id}\ \mathbb{F}, a_2 : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{join}\ (*a_1)\ (*a_2) : \&_{p+q}(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{join}$$

Note that the typing in this case restricts t_2 to also have the form $*a_2$. Then we can reduce by the following rule:

$$\frac{a \# H}{H, a_1 \mapsto_p id, a_2 \mapsto_q id, id \mapsto \mathbf{arr} \vdash \mathbf{join}\ *a_1\ *a_2 \rightsquigarrow_s H, a \mapsto_{(p+q)} id, id \mapsto \mathbf{arr} \vdash *a}\ \rightsquigarrow_{\textsc{joinArr}}$$

c) t_1 could have the form $(*(v_1, w_1))$. There are two possible typings in this case:

$$\frac{\dfrac{\dfrac{\gamma_1 \vdash v_1 : A' \quad \gamma_2 \vdash w_1 : B}{\gamma_1 + \gamma_2 \vdash (v_1, w_1) : A' \otimes B}\otimes_I}{\gamma_1 + \gamma_2 \vdash *(v_1, w_1) : *(A' \otimes B)}\ \textsc{nec} \qquad \dfrac{\dfrac{\gamma_3 \vdash v_2 : A' \quad \gamma_4 \vdash w_2 : B}{\gamma_3 + \gamma_4 \vdash (v_2, w_2) : A' \otimes B}\otimes_I}{\gamma_3 + \gamma_4 \vdash *(v_2, w_2) : *(A' \otimes B)}\ \textsc{nec}}{\gamma_1 + \gamma_2 + \gamma_3 + \gamma_4 \vdash \mathbf{join}\ (*(v_1, w_1))\ (*(v_2, w_2)) : \&_{p+q}(A' \otimes B)}\ \textsc{join}$$

Note that the typing in both situations restricts t_2 to also have the form $(*(v_2, w_2))$. In either instance we can reduce by the following rule:

$$\frac{\begin{array}{c}H \vdash \mathbf{join}\ (*v_1)\ (*v_2) \rightsquigarrow_s H' \vdash *v \\ H' \vdash \mathbf{join}\ (*w_1)\ (*w_2) \rightsquigarrow_s H'' \vdash *w\end{array}}{H \vdash \mathbf{join}\ (*(v_1, w_1))\ (*(v_2, w_2)) \rightsquigarrow_s H'' \vdash *(v, w)}\ \rightsquigarrow_{\textsc{join}\otimes}$$

– If t_2 is not a value, then by induction on the second premise we have that there exists heap H', term t_2' and context Γ' such that $H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{join}\ v\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ v\ t_2'}\ \rightsquigarrow_{\textsc{joinR}}$$

2. If t_1 is not a value, then by induction on the second premise we have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{join}\ v\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ v\ t_2'}\ \rightsquigarrow_{\textsc{joinR}}$$

- (nec)

$$\frac{\gamma \vdash t : A \qquad \neg \mathsf{resourceAllocator}(t)}{0 \cdot \Gamma, \gamma \vdash *t : *A}\ \textsc{nec}$$

By induction on the premise, there are two possibilities.

1. If t is a value, then $*t$ is also a value.
2. Otherwise by induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash *t \rightsquigarrow_s H' \vdash *t'}\ \rightsquigarrow_*$$

- (push)

$$\frac{\Gamma \vdash t : \&_p(A \otimes B)}{\Gamma \vdash \mathbf{push}\, t : (\&_p A) \otimes (\&_p B)}\ \textsc{push}$$

By induction on the premise, there are two possibilities.

1. If t is a value, then by the value lemma and the unique value lemma there are two possibilities for the form of t.
   - t could have the form $*(v_1, v_2)$. Then we can reduce by the following rule:

$$\frac{}{H \vdash \mathbf{push}\,(*(v_1, v_2)) \rightsquigarrow_s H \vdash (*v_1, *v_2)}\ \rightsquigarrow_{\textsc{push}*}$$

2. If t is not a value, by induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{push}\, t \rightsquigarrow_s H' \vdash \mathbf{push}\, t'}\ \rightsquigarrow_{\textsc{push}}$$

- (pull)

$$\frac{\Gamma \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma \vdash \mathbf{pull}\, t : \&_p(A \otimes B)}\ \textsc{pull}$$

By induction on the premise, there are two possibilities.

1. If t is a value, then by the value lemma and the unique value lemma there are two possibilities for the form of t.

– t could have the form $(*v_1, *v_2)$. Then we can reduce by the following rule:

$$\frac{}{H \vdash \mathbf{pull}\,(*v_1, *v_2) \rightsquigarrow_s H \vdash *(v_1, v_2)} \rightsquigarrow_{\text{PULL}*}$$

2. If t is not a value, by induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{pull}\,t \rightsquigarrow_s H' \vdash \mathbf{pull}\,t'} \rightsquigarrow_{\text{PULL}}$$

- (ref)

$$\frac{}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash ref : Res_{id}\ A}\ \textbf{REF}$$

A value.

- (array)

$$\frac{}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash ref : Res_{id}\ A}\ \textbf{REF}$$

A value.

- (unborrow)

$$\frac{\Gamma \vdash t : \&_1 A}{\Gamma \vdash \mathbf{unborrow}\ t : *A}\ \textsc{unborrow}$$

By induction on the premise, there are two possibilities.

1. t is a value, and therefore by the value lemma $t = (*v)$ for some value v. Then we can reduce by the following rule:

$$\frac{}{H \vdash \mathbf{unborrow}\ (*v) \rightsquigarrow_s H \vdash *v} \rightsquigarrow_{\text{UN}\&}$$

2. t is not a value. By induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{unborrow}\ t \rightsquigarrow_s H' \vdash \mathbf{unborrow}\ t'} \rightsquigarrow_{\text{UNBORROW}}$$

- (pack)

$$\frac{\Gamma \vdash t : A \qquad id \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathbf{pack}\ \langle id', t\rangle : \exists id.A[id/id']}\ \textsc{pack}$$

By induction on the premise, there are two possibilities.

1. If t is a value, then **pack** $\langle id', t\rangle$ is also a value.
2. Otherwise by induction on the premise we then have that there exists heap H', term t' and context Γ' such that $H \vdash t \rightsquigarrow_s H' \vdash t'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t \rightsquigarrow_s H \vdash t'}{H \vdash \textbf{pack}\ \langle id, t\rangle \rightsquigarrow_s H \vdash \textbf{pack}\ \langle id, t'\rangle}\ \rightsquigarrow_{\text{PACK}}$$

- (unpack)

$$\frac{\Gamma_1 \vdash t_1 : \exists id.A \qquad \Gamma_2, id, x : A \vdash t_2 : B \qquad id \notin \mathsf{fv}(B)}{\Gamma_1 + \Gamma_2 \vdash \textbf{unpack}\ \langle id, x\rangle = t_1\ \textbf{in}\ t_2 : B}\ \text{UNPACK}$$

By induction on the premise, there are two possibilities.

1. If t_1 is a value, then by the value lemma it has the form **pack** $\langle id', v\rangle$ for some value v. Then we can reduce by the following rule:

$$\frac{y\#\{H, v, t\}}{H \vdash \textbf{unpack}\ \langle id, x\rangle = \textbf{pack}\ \langle id', v\rangle\ \textbf{in}\ t \rightsquigarrow_s H, y{\mapsto_r}v \vdash t[y/x]}\ \rightsquigarrow_{\exists\beta}$$

2. Otherwise by induction on the premise we then have that there exists heap H', term t_1' and context Γ' such that $H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'$. Therefore we can reduce by the following rule:

$$\frac{H \vdash t_1 \rightsquigarrow_s H \vdash t_1'}{H \vdash \textbf{unpack}\ \langle id, x\rangle = t_1\ \textbf{in}\ t_2 \rightsquigarrow_s H \vdash \textbf{unpack}\ \langle id, x\rangle = t_1'\ \textbf{in}\ t_2}\ \rightsquigarrow_{\text{UNPACK}}$$

□

C.3.3 Preservation proof

Lemma 13 (Admissibility of weakening)**.**

$$\Gamma \vdash t : A \implies (0 \cdot \Gamma'), \Gamma \vdash t : A$$

Proof. By induction on the structure of typing and since all 'leaf' nodes of a typing derivation permit weakening (e.g., var rule, primitive rules). □

Lemma 14 (Renaming array refs)**.** *Given an array reference renaming θ (generated from a clone) then $\Gamma \vdash t : A \implies \theta(\Gamma) \vdash \theta(t) : A$.*

Proof. By trivial induction, with the only action happening in the use of the REF runtime typing rule for array references, in which case this acts just like alpha renaming via θ. □

Theorem 8 (Preservation)**.** *For a well-typed term $\Gamma \vdash t : A$, under a restriction that polymorphic reference resources are restricted to non-function types, and for all s, Γ_0, and H such that $H \bowtie (\Gamma_0 + s \cdot \Gamma)$ and a reduction $H \vdash t \rightsquigarrow_s H' \vdash t'$, then we have:*

$$\exists \Gamma', H'.\ \Gamma' \vdash t' : A \ \ \wedge \ \ H' \bowtie (\Gamma_0 + s \cdot \Gamma')$$

Note the caveat to preservation: references $\mathsf{Ref}_{id}\ A$ are restricted such that A cannot be of function type, or some other composite type involving functions. The restriction is needed for preservation since it works at the granularity of a single reduction, and so cannot rule out the possibility that a reference is storing a λ term with free variables. This considerably complicates reasoning about resources and heaps, so we rule it out for this theorem. Importantly, this is not a restriction that needs to be made on the calculus and its implementation as a whole: for deterministic CBV reductions starting from a closed term (i.e., a complete program) then all β-redexes are on closed values and hence this problem does not exist in the context of an overall reduction sequence. However, to make preservation work for a single reduction, on potentially open terms, this minor restriction is needed locally. This does not affect any of the examples discussed in the paper.

Proof. By induction on the structure of typing and reductions.

- (var)

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \textsc{var}$$

and one possible reduction:

$$\frac{\exists r'.\ s + r' \sqsubseteq r}{H, x \mapsto_r v \vdash x \ \rightsquigarrow_s \ H, x \mapsto_r v \vdash v}\ \rightsquigarrow_{\textsc{var}}$$

with incoming heap compatibility:

$$(H, x \mapsto_r v) \bowtie (\Gamma_0 + s \cdot (0 \cdot \Gamma, x : A))$$

with derivation:

$$\frac{H \bowtie (\Gamma_0 + s * 0 \cdot \Gamma + s \cdot \Gamma') \qquad x \notin \mathsf{dom}(H) \qquad \Gamma' \vdash v : A \qquad \exists r''.\ s + r'' \equiv r}{(H, x \mapsto_r v) \bowtie (\Gamma_0 + s * 0 \cdot \Gamma, x : [A]_s)}\ \textsc{ext}$$

Goal 1:

$$\exists \Gamma''.\ \Gamma'' \vdash v : A$$

Which is provided by the third premise of the head compatibility derivation here, but with weakening (Lemma 13) such that we have:

$$\frac{\Gamma' \vdash v : A}{0 \cdot \Gamma, \Gamma', x : [A]_0 \vdash v : A}\ \textsc{Lemma 13}$$

Goal 2

$$(H, x \mapsto_r v) \bowtie (\Gamma_0 + s \cdot (0 \cdot \Gamma, \Gamma', x : [A]_0))$$

given by the following derivation from the premise of the incoming heap compatibility:

$$\frac{\dfrac{H \bowtie (\Gamma_0 + s * 0 \cdot \Gamma + s \cdot \Gamma')}{H \bowtie ((\Gamma_0 + s * 0 \cdot \Gamma + s \cdot \Gamma') + 0 \cdot \Gamma')}\text{o unitality} \quad x \notin \mathrm{dom}(H) \quad \Gamma' \vdash v : A \quad \exists r'''.\ 0 + r''' \equiv r}{H, x \mapsto_r v \bowtie ((\Gamma_0 + s * 0 \cdot \Gamma + s \cdot \Gamma'), x : [A]_0)}\ \textsc{ext}$$

where the premise $\exists r''''.\ 0 + r''' \equiv r$ is fulfilled by $r''' = r$ by unitality and since $0 * s = 0$.

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

Has no reduction so the case is trivial here.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{app}$$

with incoming heap compatibility $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$.

There are four possible general reductions (and then further reductions for the operation of primitives, separated our below).

GENERAL REDUCTIONS (APP)

1. (beta)

$$\frac{y \# \{H, v, t\}}{H \vdash (\lambda x.t)\, v \rightsquigarrow_s H, y \mapsto_s v \vdash t[y/x]}\ \rightsquigarrow_\beta$$

i.e. $t_1 = (\lambda x.t)\, v$ with the refined typing (where $\Gamma = \Gamma_2$):

$$\frac{\dfrac{\Gamma_1, x : A \vdash t : B}{\Gamma_1 \vdash \lambda x.t : A \multimap B}\ \textsc{abs} \qquad \Gamma_2 \vdash v : A}{\Gamma_1 + \Gamma_2 \vdash (\lambda x.t)\, v : B}\ \textsc{app}$$

Therefore the resulting typing judgment is (goal) $\Gamma_1, x : A \vdash t : B$ given by the first premise here.

The goal heap compatibility is: (goal 2) $(H, x{\mapsto_1}v) \bowtie (\Gamma_0 + s \cdot (\Gamma_1, x : A))$

We construct the goal compatibility judgment as follows, using the incoming compatibility assumption:

$$\frac{H \bowtie (\Gamma_0 + s \cdot \Gamma_1 + s \cdot \Gamma_2) \quad x \notin \mathsf{dom}(H) \quad \Gamma_2 \vdash v : A \quad \exists r'.\, s + r' \equiv 1}{H, x{\mapsto_1}v \bowtie (\Gamma_0 + s \cdot \Gamma_1), x : [A]_s}\ \textsc{ext}$$

where $r' = 0$ here.

2. (appL)

$$\frac{H \vdash t_1 \ \rightsquigarrow_s \ H' \vdash t_1'}{H \vdash t_1\, t_2 \ \rightsquigarrow_s \ H' \vdash t_1'\, t_2}\ \rightsquigarrow_{\textsc{appL}}$$

with incoming heap compatibility $H \bowtie (\Gamma_0' + s \cdot (\Gamma_1 + \Gamma_2))$.

Applying induction on the premise reduction with heap compatibility given by the incoming heap compatibility but with $\Gamma_0 = \Gamma_0' + s \cdot \Gamma_2$ then yields:

a) $\exists \Gamma_1'.\, \Gamma_1' \vdash t_1' : A \multimap B$

b) $H' \bowtie (\Gamma_0' + s \cdot \Gamma_2 + s \cdot \Gamma_1')$

(Goal 1) is then given by the reconstructed application type $\Gamma_1' + \Gamma_2 \vdash t_1'\, t_2 : B$

(Goal 2) is $H' \bowtie (\Gamma_0' + s \cdot (\Gamma_1' + \Gamma_2))$ which is given by the second conjunct above by commutativity of + and distributivity of $*$ over $+$.

3. (appR)

$$\frac{H \vdash t_2 \ \rightsquigarrow_s \ H' \vdash t_2'}{H \vdash v\, t_2 \ \rightsquigarrow_s \ H' \vdash v\, t_2'}\ \rightsquigarrow_{\textsc{appR}}$$

Same as (appL) but by induction on the premise with t_2.

4. (prim)

$$\frac{H \vdash t \ \rightsquigarrow_s \ H' \vdash t'}{H \vdash pr\, t \ \rightsquigarrow_s \ H' \vdash pr\, t'}\ \rightsquigarrow_{\textsc{prim}}$$

Same as (appL) but by induction on the premise with t.

PRIMITIVES (APP)

1. (newRef)

$$\frac{ref \# H \qquad id \# H}{H \vdash \mathbf{newRef}\, v \ \rightsquigarrow_s \ H, ref{\mapsto_1}id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{pack}\ \langle id, *ref \rangle}\ \rightsquigarrow_{\textsc{newRef}}$$

Thus typing refines to:

$$\frac{0 \cdot \Gamma_1 \vdash \mathbf{newRef} : A \multimap \exists id.{*}(\mathsf{Ref}_{id}\ A) \quad \Gamma_2 \vdash v : A}{0 \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{newRef}\, v : \exists id.{*}(\mathsf{Ref}_{id}\ A)}\ \textsc{app}$$

with incoming heap compatibility $H \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma_1 + \Gamma_2)$ which is equal to $H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + s \cdot \Gamma_2$ by distributivity of $*$ over $+$ and absorption.

By the closed value lemma (Lemma 11) then $\Gamma_2 = \gamma$ and thus $s \cdot \gamma = \gamma$ since multiplication has no effect on runtime reference type assumptions, therefore incoming heap compatibility is $H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + \gamma$.

Goal 1 (typing) is thus given by:

$$\frac{\dfrac{}{0 \cdot \Gamma_1, ref : \mathsf{Ref}_{id}\ A \vdash *ref : *(\mathsf{Ref}_{id}\ A)}\ {*\textsc{ref}^*}}{0 \cdot \Gamma_1, ref : \mathsf{Ref}_{id}\ A \vdash \mathbf{pack}\ \langle id, *ref \rangle : \exists id.\mathsf{Ref}_{id}\ A}\ \textsc{pack}$$

i.e., we set $\Gamma' = 0 \cdot \Gamma_1, ref : \mathsf{Ref}_{id}\ A$ (which notably has runtime typing of ref).

Goal 2 (compatibility) is $(H, ref \mapsto_1 id, id \mapsto v) \bowtie (\Gamma_0 + s \cdot (0 \cdot \Gamma_1, ref : \mathsf{Ref}_{id}\ A))$ which is equal to: $(H, ref \mapsto_1 id, id \mapsto v) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1, ref : \mathsf{Ref}_{id}\ A)$

We construct this goal as follows (leveraging distributivity of $\cdot$ over $+$):

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + \gamma \quad \dfrac{\gamma \vdash v : A}{\gamma \vdash \mathbf{ref}(v) : \mathsf{Ref}_{id}\ A}\ \textsc{refStore}}{H, ref \mapsto_1 id, id \mapsto \mathbf{ref}(v) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1, ref : \mathsf{Ref}_{id}\ A)}\ \textsc{extRes}$$

2. (swapRef)

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{swapRef}\ (*ref)\ v' \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v') \vdash v}\ \rightsquigarrow\textsc{swapRef}$$

Thus typing refines to

$$\frac{\begin{array}{c} 0 \cdot \Gamma_1 \vdash \mathbf{swapRef} : \&_p(\mathsf{Ref}_{id}\ A) \multimap A \multimap A \otimes \&_p(\mathsf{Ref}_{id}\ A) \\ 0 \cdot \Gamma_2, ref : \mathsf{Ref}_{id}\ A \vdash *ref : *(\mathsf{Ref}_{id}\ A) \qquad \Gamma_3 \vdash v' : A \end{array}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + \Gamma_3, ref : \mathsf{Ref}_{id}\ A \vdash \mathbf{swapRef}\ (*ref)\ v' : A \otimes \&_p(\mathsf{Ref}_{id}\ A)}\ \textsc{app}\times 2$$

i.e. where $p = *$.

By the closed value lemma (Lemma 11) then $\Gamma_3 = \gamma'$ and thus $s \cdot \gamma' = \gamma'$ since multiplication has no effect on runtime reference type assumptions, therefore incoming heap compatibility is:

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + \gamma + \gamma' \quad \dfrac{\gamma \vdash v : A}{\gamma \vdash \mathbf{ref}(v) : \mathsf{Ref}_{id}\ A}\ \textsc{refStore}}{(H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v)) \bowtie (\Gamma_0 + s \cdot (0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + \gamma', ref : \mathsf{Ref}_{id}\ A))}\ \textsc{extRes}$$

Goal 1 (typing) is thus given directly by the heap compatibility's second premise: $\gamma \vdash v : A$

Goal 2 (heap compatibility) is

$$(H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v')) \bowtie (\Gamma_0 + s \cdot \gamma, ref : \mathsf{Ref}_{id}\ A)$$

By absorption then the premise of incoming heap compatibility $H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + \gamma + \gamma'$ is equal to: $H \bowtie \Gamma_0 + \gamma + \gamma'$, and since $s \cdot \gamma = \gamma$ then we can provide this goal by:

$$\frac{H \bowtie \Gamma_0 + s \cdot \gamma + \gamma' \quad \dfrac{\gamma' \vdash v : A}{\gamma' \vdash \mathbf{ref}(v') : \mathsf{Ref}_{id}\ A}\ \textsc{refStore}}{(H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v')) \bowtie (\Gamma_0 + s \cdot (\gamma, ref : \mathsf{Ref}_{id}\ A))}\ \textsc{extRes}$$

3. (freezeRef)

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{freezeRef}\,(*ref) \rightsquigarrow_s H \vdash v}\ \rightsquigarrow\textsc{freezeRef}$$

Thus typing refines to

$$\frac{\begin{array}{c} 0 \cdot \Gamma_1 \vdash \mathbf{freezeRef} : \forall\, id.*(\mathsf{Ref}_{id}\ A) \multimap A \\ 0 \cdot \Gamma_2, ref : \mathsf{Ref}_{id}\ A \vdash *ref : *(\mathsf{Ref}_{id}\ A) \end{array}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, ref : \mathsf{Ref}_{id}\ A \vdash \mathbf{freezeRef}\,(*ref) : A}\ \textsc{app}$$

and we have incoming heap compatibility:

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + \gamma \quad \dfrac{\gamma \vdash v : A}{\gamma \vdash \mathbf{ref}(v) : \mathsf{Ref}_{id}\ A}\ \textsc{refStore}}{(H, a \mapsto_p id, id \mapsto v) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, ref : \mathsf{Ref}_{id}\ A)}\ \textsc{extRes}$$

Goal 1 (typing) is thus given directly by the second premise of heap compatibility: $\gamma \vdash v : A$

Goal 2 (heap compatibility) is then $H \bowtie (\Gamma_0 + s \cdot \gamma)$ given by the premise of incoming heap compatibility since $\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 = \Gamma_0$ and $s \cdot \gamma = \gamma$:

$$\begin{aligned} & H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + \gamma \\ \equiv\ & H \bowtie \Gamma_0 + s \cdot \gamma \end{aligned}$$

4. (readRef)

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_{r+1}) \vdash \mathbf{readRef}\,(*ref) \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_r) \vdash (v, *ref)}\ \rightsquigarrow\textsc{readRef}$$

Thus typing refines to

$$\frac{\begin{array}{c} 0 \cdot \Gamma_1 \vdash \mathbf{readRef} : \&_p(\mathsf{Ref}_{id}\ \square_{r+1} A) \multimap A \otimes \&_p(\mathsf{Ref}_{id}\ \square_r A) \\ 0 \cdot \Gamma_2, ref : \mathsf{Ref}_{id}\ \square_{r+1} A \vdash *ref : *(\mathsf{Ref}_{id}\ \square_{r+1} A) \end{array}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, ref : \mathsf{Ref}_{id}\ A \vdash \mathbf{readRef}\,(*ref) : A \otimes \&_p(\mathsf{Ref}_{id}\ \square_r A)}\ \textsc{app}$$

i.e. where $p = *$ and we have incoming heap compatibility:

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + \gamma \qquad \dfrac{\dfrac{\gamma \vdash v : A}{\gamma \vdash [v]_{r+1} : \Box_{r+1} A}\,\textsc{pr}}{\gamma \vdash \mathbf{ref}([v]_{r+1}) : \mathsf{Ref}_{id}\ (\Box_{r+1} A)}\,\textsc{refStore}}{(H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_{r+1})) \bowtie (\Gamma_0 + s \cdot (0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, ref : \mathsf{Ref}_{id}\ A))}\,\textsc{extRes}$$

where recall $r + 1 \cdot \gamma = \gamma$ and thus compatibility is which is equal to $(H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_{r+1})) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, ref : \mathsf{Ref}_{id}\ A)$ by absorption.

Goal 1 (typing) is thus given by:

$$\frac{\gamma \vdash v : A \quad 0 \cdot \Gamma_2, ref : \mathsf{Ref}_{id}\ \Box_r A \vdash *ref : *(\mathsf{Ref}_{id}\ \Box_r A)}{\gamma + 0 \cdot \Gamma_2, ref : \mathsf{Ref}_{id}\ A \vdash (v, *ref) : A \otimes \&_p(\mathsf{Ref}_{id}\ \Box_r A)}\ \otimes_I$$

Goal 2 (heap compatibility) is

$$(H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_r)) \bowtie (\Gamma_0 + 0 \cdot (\Gamma_1 + \Gamma_2), ref : \mathsf{Ref}_{id}\ A)$$

which is provided exactly by the incoming heap compatibility but re-deriving promotion at the grade r.

5. (newArray)

$$\frac{ref \# H \qquad id \# H}{H \vdash \mathbf{newArray}\, n \rightsquigarrow_s H, ref \mapsto_1 id, id \mapsto \mathsf{init} \vdash \mathbf{pack}\ \langle id, *ref \rangle}\ \rightsquigarrow\textsc{newArray}$$

Thus typing refines to:

$$\frac{0 \cdot \Gamma_1 \vdash \mathbf{newArray} : \mathbb{N} \multimap \exists id.*(\mathsf{Array}_{id}\ \mathbb{F}) \quad 0 \cdot \Gamma_2 \vdash n : \mathbb{N}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 \vdash \mathbf{newArray}\, n : \exists id.*(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{app}$$

with incoming heap compatibility $H \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma_1 + 0 \cdot \Gamma_2)$ which is equal to $H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2$ by distributivity of $*$ over $+$ and absorption.

Goal 1 (typing) is thus given by:

$$\frac{\dfrac{}{0 \cdot (\Gamma_1 + \Gamma_2), a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : *(\mathsf{Array}_{id}\ \mathbb{F})}\ *\textsc{ref}^*}{0 \cdot (\Gamma_1 + \Gamma_2), a : \mathsf{Array}_{id}\ A \vdash \mathbf{pack}\ \langle id, *a \rangle : \exists id.\mathsf{Array}_{id}\ \mathbb{F}}\ \textsc{pack}$$

i.e., we set $\Gamma' = 0 \cdot (\Gamma_1 + \Gamma_2), a : \mathsf{Array}_{id}\ A$ (which notably has runtime typing of a).

Goal 2 (compatibility) is $(H, a \mapsto_1 id, id \mapsto \mathsf{init}) \bowtie (\Gamma_0 + s \cdot (0 \cdot (\Gamma_1 + \Gamma_2), a : \mathsf{Array}_{id}\ A))$

We construct this goal as follows (leveraging distributivity of $\cdot$ over $+$ and absorption and since $s \cdot \gamma = \gamma$):

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 \quad \dfrac{}{\emptyset \vdash \mathsf{init} : \mathsf{Array}_{id}\ \mathbb{F}}\ \textsc{arrayInit}}{H, a \mapsto_1 id, id \mapsto \mathsf{init} \bowtie (\Gamma_0 + 0 \cdot (\Gamma_1 + \Gamma_2), a : \mathsf{Array}_{id}\ A)}\ \textsc{extRes}$$

6. (readArray)

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{arr}[n] = v \vdash \mathbf{readArray}\,(*ref)\,n \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{arr}[n] = v \vdash (v, *ref)}\ \rightsquigarrow_{\text{READARRAY}}$$

Thus typing refines to

$$\frac{\begin{array}{c} 0 \cdot \Gamma_1 \vdash \mathbf{readArray} : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \\ 0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : *(\mathsf{Array}_{id}\ A) \qquad 0 \cdot \Gamma_3 \vdash i : \mathbb{N} \end{array}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{readArray}\,(*a)\,i : \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{APP}\times 2$$

i.e. where $p = *$ and we have incoming heap compatibility (simplified below by absorption since $(\Gamma_0 + s \cdot (0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})) = (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})$):

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3 \qquad \dfrac{\emptyset \vdash \mathbf{arr} : \mathsf{Array}_{id}\ \mathbb{F} \quad \emptyset \vdash v : \mathbb{F}}{\emptyset \vdash \mathbf{arr}[i] = v : \mathsf{Array}_{id}\ \mathbb{F}}\ \text{ARRAYAT}}{(H, a \mapsto_p id, id \mapsto \mathbf{arr}[i] = v) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})}\ \text{EXTRES}$$

Goal 1 (typing) is thus given by:

$$\frac{\begin{array}{c} \emptyset \vdash v : \mathbb{F} \\ 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \end{array}}{0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (v, *a) : \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \otimes_I$$

i.e. $\Gamma' = 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F}$.

Goal 2 (heap compatibility) is

$$(H, a \mapsto_p id, id \mapsto \mathbf{arr}[i] = v) \bowtie (\Gamma_0 + s \cdot 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F})$$

which is provided exactly by the incoming heap compatibility.

7. (writeArray)

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\,(*ref)\,n\,v \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{arr}[n] = v \vdash *ref}\ \rightsquigarrow_{\text{WRITEARRAY}}$$

Thus typing refines to

$$\frac{\begin{array}{c} 0 \cdot \Gamma_1 \vdash \mathbf{writeArray} : \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \\ 0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : \&_p(\mathsf{Array}_{id}\ A) \\ 0 \cdot \Gamma_3 \vdash n : \mathbb{N} \qquad 0 \cdot \Gamma_4 \vdash v : \mathbb{F} \end{array}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3 + 0 \cdot \Gamma_4, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{writeArray}\,(*a)\,n\,v : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{APP}\times 3$$

and we have incoming heap compatibility (simplified by absorption as : $(\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3 + 0 \cdot \Gamma_4, a : \mathsf{Array}_{id}\ \mathbb{F}) = (\Gamma_0 + s \cdot (0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3 + 0 \cdot \Gamma_4, a : \mathsf{Array}_{id}\ \mathbb{F}))$):

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3 + 0 \cdot \Gamma_4 \quad \emptyset \vdash \mathbf{arr} : \mathsf{Array}_{id}\ \mathbb{F}}{(H, a \mapsto_p id, id \mapsto \mathbf{arr}) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3 + 0 \cdot \Gamma_4, a : \mathsf{Array}_{id}\ \mathbb{F})}\ \text{EXTRES}$$

Goal 1 (typing) is thus given by:

$$\frac{0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (*a) : \&_p(\mathsf{Array}_{id}\ A)}{0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (*a) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{Lemma 13}$$

i.e. $\Gamma' = 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F}$ and where we can strengthen $0 \cdot \Gamma_4 \vdash v : \mathbb{F}$ to $\emptyset \vdash v : \mathbb{F}$ by inversion on float values.

Goal 2 (heap compatibility) is

$$(H, a \mapsto_p id, id \mapsto \mathbf{arr}[i] = v) \bowtie (\Gamma_0 + s \cdot 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F})$$

which is provided exactly by the incoming heap compatibility:

$$\frac{H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3 + 0 \cdot \Gamma_4 \qquad \dfrac{\emptyset \vdash \mathbf{arr} : \mathsf{Array}_{id}\ \mathbb{F} \quad \emptyset \vdash v : \mathbb{F}}{\emptyset \vdash \mathbf{arr}[i] = v : \mathsf{Array}_{id}\ \mathbb{F}}\ \textsc{arrayAt}}{(H, a \mapsto_p id, id \mapsto \mathbf{arr}) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{extRes}$$

where $(\Gamma_0 + s \cdot 0 \cdot (\Gamma_1 + \Gamma_2 + \Gamma_3), a : \mathsf{Array}_{id}\ \mathbb{F}) = (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2 + 0 \cdot \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})$.

8. (deleteArray)

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{arr} \vdash \mathbf{deleteArray}\,(*ref) \rightsquigarrow_s H \vdash ()}\ \rightsquigarrow\textsc{deleteArray}$$

Thus typing refines to

$$\frac{\begin{array}{c} 0 \cdot \Gamma_1 \vdash \mathbf{deleteArray} : \forall\, id.*(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathsf{unit} \\ 0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : *(\mathsf{Array}_{id}\ A) \end{array}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{deleteArray}\,(*a) : \mathsf{unit}}\ \textsc{app}$$

and we have incoming heap compatibility:

$$\frac{H \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma_1 + 0 \cdot \Gamma_2) \quad \emptyset \vdash \mathbf{arr} : \mathsf{Array}_{id}\ \mathbb{F}}{(H, a \mapsto_p id, id \mapsto \mathbf{arr}) \bowtie (\Gamma_0 + s \cdot (0 \cdot \Gamma_1 + 0 \cdot \Gamma_2), a : \mathsf{Array}_{id}\ \mathbb{F})}\ \textsc{extRes}$$

Goal 1 (typing) is thus given by:

$$\frac{}{0 \cdot (\Gamma_1 + \Gamma_2) \vdash () : \mathsf{unit}}\ 1_I$$

with $\Gamma' = 0 \cdot (\Gamma_1 + \Gamma_2)$

Goal 2 (heap compatibility) is $H \bowtie (\Gamma_0 + s * 0 \cdot (\Gamma_1 + \Gamma_2))$ given by the premise of incoming heap compatibility.

- (boxElim)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\ \mathbf{in}\ t_2 : B}\ \textsc{elim}$$

And two possible reductions:

1. (betaBoxElim)

$$\frac{y\#\{H,v,t\}}{H \vdash \mathbf{let}\,[x] = [v]_r\,\mathbf{in}\,t \;\rightsquigarrow_s\; H, y{\mapsto_{(s*r)}}v \vdash t[y/x]}\;{\rightsquigarrow}\Box\beta$$

refining the typing to:

$$\frac{\dfrac{[\Gamma_1'] \vdash v : A}{r\cdot\Gamma_1' \vdash [v]_r : \Box_r A}\,\textsc{pr} \qquad \Gamma_2, x : [A]_r \vdash t : B}{r\cdot\Gamma_1' + \Gamma_2 \vdash \mathbf{let}\,[x] = [v]_r\,\mathbf{in}\,t : B}\,\textsc{elim}$$

with incoming heap compatibility:

$$H \bowtie \Gamma_0 + s\cdot(r\cdot\Gamma_1' + \Gamma_2)$$

Goal 1: typing Provided by the premise here as:

$$\Gamma_2, x : [A]_r \vdash t : A$$

under renaming to:

$$\Gamma_2, y : [A]_r \vdash t[y/x] : A$$

i.e., we set the goal $\Gamma' = \Gamma_2, y : [A]_r$.

Goal 2: heap compatibility is $(H, y{\mapsto_{s*r}}v) \bowtie (\Gamma_0 + s\cdot(\Gamma_2, y : [A]_r))$ which refines to: $(H, y{\mapsto_{s*r}}v) \bowtie (\Gamma_0 + s\cdot\Gamma_2), y : [A]_{s*r}$ by the disjointness of x

We construct this goal via the premise heap compatibility by

$$\frac{H \bowtie \Gamma_0 + s*r\cdot\Gamma_1' + s\cdot\Gamma_2}{(H, y{\mapsto_{s*r}}v) \bowtie (\Gamma_0 + s\cdot\Gamma_2), y : [A]_{s*r}}\,\textsc{ext}$$

2. (congBoxElim)

$$\frac{H \vdash t_1 \;\rightsquigarrow_s\; H' \vdash t_1'}{H \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \;\rightsquigarrow_s\; H' \vdash \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2}\;{\rightsquigarrow}_{\textsc{let}\Box}$$

with incoming heap compatibility:

$$H \bowtie \Gamma_0' + \Gamma_1 + \Gamma_2$$

Applying induction on the premise reduction with heap compatibility given by the incoming heap compatibility but with $\Gamma_0 = \Gamma_0' + \Gamma_2$ then yields:

a) $\exists\Gamma_1'.\ \Gamma_1' \vdash t_1' : \Box_r A$

b) $H' \bowtie (\Gamma_0' + s\cdot\Gamma_2 + s\cdot\Gamma_1')$

(Goal 1) is then given by the reconstructed application type $\Gamma_1' + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2 : B$

(Goal 2) is $H' \bowtie (\Gamma_0' + s \cdot (\Gamma_1' + \Gamma_2))$ which is given by the second conjunct above by commutativity of + and distributivity of $*$ over $+$.

- (boxIntro)

$$\frac{\Gamma \vdash t : A \qquad \neg \mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t]_r : \Box_r A}\ \textsc{pr}$$

and one possible reduction

$$\frac{H \vdash t \rightsquigarrow_{s*r} H' \vdash t'}{H \vdash [t]_r \rightsquigarrow_s H' \vdash [t']_r}\ \rightsquigarrow\Box$$

Aside: Here we see the need for the Church-style graded annotation here in the term as otherwise we could have a reduction derivation that uses a different grade in its premise:

$$\frac{H \vdash t \rightsquigarrow_{s*r'} H' \vdash t'}{H \vdash [t] \rightsquigarrow_s H' \vdash [t']}\ \rightsquigarrow\Box\text{-}\textit{alternate-nonChurch}$$

In which case we could not induct on the first premise as we would not have the required heap compatibility $H \bowtie \Gamma_0 + s' * r' \cdot \Gamma$. Thus this motivates the need for the Church style typing here.

Thus, instead we have the incoming heap compatibility:

$$H \bowtie \Gamma_0' + s' * r \cdot \Gamma$$

Applying induction on the premise reduction with $\Gamma_0 = \Gamma_0'$ and $s = s' * r$ then yields:

1. $\exists \Gamma_1'.\ \Gamma_1' \vdash t_1' : A$
2. $H' \bowtie (\Gamma_0' + (s' * r) \cdot \Gamma_1')$

(Goal 1) is then given by the reconstructed application type $r \cdot \Gamma_1' \vdash [t_1'] : \Box_r A$

$$\frac{\Gamma_1' \vdash t_1' : A}{r \cdot \Gamma_1' \vdash [t_1'] : \Box_r A}\ \rightsquigarrow\Box$$

(Goal 2) is $H' \bowtie (\Gamma_0' + s' * r \cdot \Gamma_1')$ which is given by the second conjunct above by commutativity of + and associativity of $*$.

- (der)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

with incoming heap compatibility $H \bowtie \Gamma_0 + s \cdot (\Gamma, x : [A]_1)$ which refines to $H \bowtie \Gamma_0 + s \cdot \Gamma, x : [A]_s$ and a reduction:

$$H \vdash t \rightsquigarrow_s H' \vdash t'$$

Induction requires $H \bowtie \Gamma_0 + s \cdot (\Gamma, x : A)$ which by the definition of scalar multiplication is just $H \bowtie \Gamma_0 + s \cdot \Gamma, x : [A]_s$, thus we can apply induction and get the result of $\Gamma' \vdash t' : B$ and $H' \bowtie \Gamma_0 + s \cdot \Gamma_1'$ satisfying the goal here.

- (tensorIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B} \otimes_I$$

with incoming heap compatibility $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$.

And two possible reductions:

1. (congPairL)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash (t_1, t_2) \rightsquigarrow_s H' \vdash (t_1', t_2)} \rightsquigarrow_{\otimes \mathrm{L}}$$

   By induction on the premise with $\Gamma_0' = \Gamma_0 + s \cdot \Gamma_2$ then we have: (i) $\Gamma_1' \vdash t_1' : A$ and (ii) $H' \bowtie \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1'$. From which we construct the resulting typing, via applying $\otimes_I$ again to get $\Gamma_1' + \Gamma_2 \vdash (t_1', t_2) : A \otimes B$ and with (ii) providing the required heap compatibility (by commutativity of $+$).

2. (congPairR)

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash (v, t_2) \rightsquigarrow_s H' \vdash (v, t_2')} \rightsquigarrow_{\otimes \mathrm{R}}$$

   Essentially the same as the preceding case (congPairL) but by induction on the second premise.

- (tensorElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C} \otimes_E$$

with incoming heap compatibility $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$.

And two possible reductions:

1. (pairBeta)

$$\frac{x' \# \{H, v_1, v_2, t\} \qquad y' \# \{H, v_1, v_2, t\}}{H \vdash \mathbf{let}\,(x, y) = (v_1, v_2)\,\mathbf{in}\,t \rightsquigarrow_s H, x' \mapsto_s v_1, y' \mapsto_s v_2 \vdash t[y'/y][x'/x]} \rightsquigarrow_{\otimes \beta}$$

   with typing $\Gamma_2, x : A, y : B \vdash t_2 : C$ as the result i.e., with $\Gamma' = \Gamma_2, x : A, y : B$ and outgoing heap compatibility

2. (pairElim)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,(x,y) = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,(x,y) = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{LET}\otimes}$$

By induction on the premise with $\Gamma_0' = \Gamma_0 + s \cdot \Gamma_2$ then we have: (i) $\Gamma_1' \vdash t_1' : A \otimes B$ and (ii) $H' \bowtie \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1'$. From which we construct the resulting typing, via applying $\otimes_E$ again to get $\Gamma_1' + \Gamma_2 \vdash \mathbf{let}\,(x,y) = t_1'\,\mathbf{in}\,t_2 : C$ and with (ii) providing the required heap compatibility (by commutativity of $+$).

- (unitIntro)

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}}\ 1_I$$

Trivial case since there is no heap semantics rule as this is already a normal form value.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : B}\ 1_E$$

And two possible reductions:

1. (betaUnit)

$$\frac{}{H \vdash \mathbf{let}\,() = ()\,\mathbf{in}\,t \rightsquigarrow_s H \vdash t}\ \rightsquigarrow_{\beta\text{UNIT}}$$

which refines the typing to:

$$\frac{0 \cdot \Gamma_1 \vdash () : \mathsf{unit} \quad \Gamma_2 \vdash t_2 : B}{0 \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = ()\,\mathbf{in}\,t_2 : B}\ 1_E$$

with incoming heap compatibility $H \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma_1 + \Gamma_2)$ which refines to $H \bowtie \Gamma_0 + 0 \cdot \Gamma_1 + s \cdot \Gamma_2$.

The resulting typing is thus given by:

$$\frac{\Gamma_2 \vdash t_2 : B}{0 \cdot \Gamma_1 + \Gamma_2 \vdash t_2 : B}\ \text{Lemma 13}$$

i.e., $\Gamma' = 0 \cdot \Gamma_1 + \Gamma_2$ and outgoing heap compatibility is provided exactly by the incoming heap compatibility.

2. (congUnitElim)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,() = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{LETUNIT}}$$

By induction on the premise with $\Gamma_0' = \Gamma_0 + s \cdot \Gamma_2$ then we have: (i) $\Gamma_1' \vdash t_1' : \mathsf{unit}$ and (ii) $H' \bowtie \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1'$. From which we construct the resulting typing, via applying 1_E again to get $\Gamma_1' + \Gamma_2 \vdash \mathbf{let}\ () = t_1'\ \mathbf{in}\ t_2 : B$ and with (ii) providing the required heap compatibility (by commutativity of $+$).

- (share)

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathbf{share}\, t : \Box_r A}\ \textsc{share}$$

And two possible reductions:

1. (share)

$$\frac{\mathsf{dom}(H) \equiv \mathsf{refs}(v)}{H, H' \vdash \mathbf{share}\,(*v) \rightsquigarrow_s ([H]_0), H' \vdash [v]}\ \rightsquigarrow_{\textsc{share}\beta}$$

with refined (runtime) typing, and by Lemma 10,

$$\frac{\dfrac{0 \cdot \Gamma_1', \gamma \vdash v : A}{0 \cdot (\Gamma_1', \Gamma_1''), \gamma \vdash *v : *A}\ \textsc{nec}}{0 \cdot (\Gamma_1', \Gamma_1''), \gamma \vdash \mathbf{share}\,(*v) : \Box_r A}\ \textsc{share}$$

and thus incoming heap compatibility $H, H' \bowtie \Gamma_0 + s \cdot (0 \cdot (\Gamma_1', \Gamma_1''), \gamma)$ which refines to: $H, H' \bowtie \Gamma_0 + 0 \cdot (\Gamma_1', \Gamma_1''), \gamma$.

The resulting type is given by:

$$\frac{\dfrac{0 \cdot \Gamma_1', \gamma \vdash v : A}{0 \cdot (\Gamma_1', \Gamma_1''), \gamma \vdash v : A}\ \textsc{Lemma 13}}{r \cdot (0 \cdot (\Gamma_1', \Gamma_1''), \gamma) \vdash [v] : \Box_r A}\ \textsc{pr}$$

where $r \cdot (0 \cdot (\Gamma_1', \Gamma_1''), \gamma) = 0 \cdot (\Gamma_1', \Gamma_1''), \gamma$ and thus $\Gamma' = 0 \cdot (\Gamma_1', \Gamma_1''), \gamma$ and the outgoing heap compatibility is provided by the incoming heap compatibility but where the heap has zeroed out H (which does not affect the heap compatibility; it can be re-derived with different fractions).

2. (congShare)

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{share}\, t \rightsquigarrow_s H' \vdash \mathbf{share}\, t'}\ \rightsquigarrow_{\textsc{share}}$$

By induction on the premise with $\Gamma_0' = \Gamma_0$ then we have: (i) $\Gamma' \vdash t' : *A$ and (ii) $H' \bowtie \Gamma_0 + s \cdot \Gamma'$. From which we construct the resulting typing, via applying SHARE again to get $\Gamma' \vdash \mathbf{share}\, t' : \Box_r A$ and with (ii) providing the required heap compatibility.

- (clone)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : *A \vdash t_2 : \Box_r B \qquad 1 \sqsubseteq r}{\Gamma_1 + \Gamma_2 \vdash \mathbf{clone'}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 : \Box_r B}\ \textsc{clone}'$$

And two possible reductions:

1. (cloneCongr)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{clone}\, t_1\, \mathbf{as}\, x\, \mathbf{in}\, t_2 \rightsquigarrow_s H' \vdash \mathbf{clone}\, t_1'\, \mathbf{as}\, x\, \mathbf{in}\, t_2}\ \rightsquigarrow_{\textsc{clone}}$$

Follows by induction similar to other congruences.

2. (cloneBeta)

$$\frac{\mathsf{dom}(H') \equiv \mathsf{refs}(v) \qquad (H'', \theta, \overline{id}) = \mathsf{copy}(H') \qquad y\#\{H, v, t\}}{H, H' \vdash \mathbf{clone}\, [v]_r\, \mathbf{as}\, x\, \mathbf{in}\, t \rightsquigarrow_s H, H', H'', y \mapsto_s \mathbf{pack}\, \langle \overline{id}, *(\theta(v)) \rangle \vdash t[y/x]}\ \rightsquigarrow_{\textsc{clone}\beta}$$

refining the typing to:

$$\frac{\dfrac{\Gamma_1, \overline{id_1} \vdash v : A}{r \cdot \Gamma_1, \overline{id_1} \vdash [v]_r : \Box_r A}\ \textsc{pr} \qquad \Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id_1}]) \vdash t : \Box_r B \qquad 1 \sqsubseteq r}{(r \cdot \Gamma_1 + \Gamma_2), \overline{id} \vdash \mathbf{clone}\, [v]_r\, \mathbf{as}\, x\, \mathbf{in}\, t : \Box_r B}\ \textsc{clone}'$$

with incoming heap compatibility $H, H' \bowtie \Gamma_0 + s \cdot (r \cdot \Gamma_1 + \Gamma_2)$.

Goal typing is then given by:

$$\Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id_1}]) \vdash t : \Box_r B$$

thus with $\Gamma' = \Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id_1}])$.

The typing of the extended heap is given by the value:

$$\frac{\dfrac{\dfrac{\Gamma_1, \overline{id_1} \vdash v : A}{\theta(\Gamma_1, \overline{id_1}) \vdash \theta(v) : \theta(A)}\ \textsc{Lemma 14}}{\theta(\Gamma_1, \overline{id_1}) \vdash *(\theta(v)) : *\theta(A)}\ \textsc{nec}}{\theta(\Gamma_1), \overline{id} \vdash \mathbf{pack}\, \langle \overline{id}, *(\theta(v)) \rangle : \exists \overline{id'}.*\theta(A)[id'/id]}\ \textsc{pack} \tag{c.1}$$

since θ maps each $\overline{id_1}$ to $\overline{id}$ then $\exists \overline{id'}.*\theta(A)[id'/id] = \exists \overline{id'}.*A[id'/id_1]$.

Goal heap compatibility: $(H, H', H'', x \mapsto_s * (\theta(v))) \bowtie (\Gamma_0 + s \cdot (\Gamma_2, x : *(\#A)))$ given by:

$$\frac{\dfrac{\dfrac{H, H' \bowtie (\Gamma_0 + s \cdot \Gamma_2 + r * s \cdot \Gamma_1)}{H, H', H'' \bowtie (\Gamma_0 + s \cdot \Gamma_2 + r * s \cdot \theta(\Gamma_1))}\ 1 \sqsubseteq r \qquad x \notin \mathsf{dom}(H)}{H, H', H'' \bowtie (\Gamma_0 + s \cdot \Gamma_2 + s \cdot \theta(\Gamma_1))}\ \exists r'.\ r + r' \equiv s \qquad \text{(c.1)}}{(H, H', H'', x \mapsto_s \mathbf{pack}\, \langle \overline{id}, *(\theta(v)) \rangle) \bowtie (\Gamma_0 + s \cdot \Gamma_2, x : [\exists \overline{id'}.*(A[\overline{id'}/\overline{id_1}])]_s)}\ \textsc{ext}$$

- (withBorrow)

$$\frac{\Gamma_1 \vdash t_1 : *A \qquad \Gamma_2 \vdash t_2 : \&_1 A \multimap \&_1 B}{\Gamma_1 + \Gamma_2 \vdash \textbf{withBorrow}\ t_1\ t_2 : *B}\ \textsc{withBorrow}$$

And three possible reductions:

1. (congWithBorrowL)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \textbf{withBorrow}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \textbf{withBorrow}\ t_1'\ t_2}\ \rightsquigarrow_{\textsc{with\&L}}$$

Which follows by induction similar to other inductive cases.

2. (congWithBorrowR)

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \textbf{withBorrow}\ (\lambda x.t_1)\ t_2 \rightsquigarrow_s H' \vdash \textbf{withBorrow}\ (\lambda x.t_1)\ t_2'}\ \rightsquigarrow_{\textsc{with\&R}}$$

Which follows by induction similar to other inductive cases.

3. (withBorrow)

$$\frac{y\#\{H, v, t\}}{H \vdash \textbf{withBorrow}\ (\lambda x.t)\ (*v) \rightsquigarrow_s H, y \mapsto_s (*v) \vdash \textbf{unborrow}\ t[y/x]}\ \rightsquigarrow_{\textsc{with\&}}$$

which refines the typing to:

$$\frac{\dfrac{\Gamma_2, x : \&_1 A \vdash t : \&_1 B}{\Gamma_2 \vdash \lambda x.t : \&_1 A \multimap \&_1 B}\ \textsc{withBorrow} \qquad \dfrac{\Gamma_1 \vdash v : A}{\Gamma_1 \vdash *v : *A}\ \textsc{nec}}{\Gamma_1 + \Gamma_2 \vdash \textbf{withBorrow}\ (\lambda x.t)\ (*v) : *B}\ \textsc{withBorrow}$$

with incoming heap compatibility $H \bowtie (\Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2))$.

We then get the resulting typing as:

$$\frac{\Gamma_2, y : \&_1 A \vdash t[y/x] : \&_1 B}{\Gamma_2, y : \&_1 A \vdash \textbf{unborrow}\ (t[y/x]) : *A}\ \textsc{unborrow}$$

thus: $\Gamma' = \Gamma_2, y : \&_1 A$ and construct the goal heap compatibility as:

$$\frac{H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2) \qquad \Gamma_1 \vdash *v : \&_1 A}{(H, y \mapsto_s (*v)) \bowtie (\Gamma_0 + s \cdot \Gamma_2), y : [\&_1 A]_s}\ \textsc{extLin}$$

- (unborrow)

$$\frac{\Gamma \vdash t : \&_1 A}{\Gamma \vdash \textbf{unborrow}\ t : *A}\ \textsc{unborrow}$$

And two possible reductions:

1. (congUnborrow)

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{unborrow}\ t \rightsquigarrow_s H' \vdash \mathbf{unborrow}\ t'} \rightsquigarrow_{\text{UNBORROW}}$$

   By induction as in the other inductive cases.

2. (unborrowBorrw)

$$\frac{}{H \vdash \mathbf{unborrow}\ (*v) \rightsquigarrow_s H \vdash *v} \rightsquigarrow_{\text{UN\&}}$$

   with the refined typing:

$$\frac{\dfrac{\Gamma \vdash v : A}{\Gamma \vdash (*v) : \&_1 A}\ \text{NEC}}{\Gamma \vdash \mathbf{unborrow}\ (*v) : *A}\ \text{UNBORROW}$$

   and thus heap compatibility is $H \bowtie \Gamma_0 + s \cdot \Gamma$

   The resulting typing matches the goal as:

$$\frac{\Gamma \vdash v : A}{\Gamma \vdash (*v) : *A}\ \text{NEC}$$

   thus $\Gamma' = \Gamma$ and outgoing heap compatibility is provided by the incoming compatibility.

- (push)

$$\frac{\Gamma \vdash t : \&_p(A \otimes B)}{\Gamma \vdash \mathbf{push}\, t : (\&_p A) \otimes (\&_p B)}\ \text{PUSH}$$

  And three possible reductions:

  1. (congPush)

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{push}\, t \rightsquigarrow_s H' \vdash \mathbf{push}\, t'} \rightsquigarrow_{\text{PUSH}}$$

     Inductive case as in other inductive rules.

  2. (pushUnique)

$$\frac{}{H \vdash \mathbf{push}\ (*(v_1, v_2)) \rightsquigarrow_s H \vdash (*v_1, *v_2)} \rightsquigarrow_{\text{PUSH}*}$$

     with the refined typing:

$$\frac{\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : (A \otimes B)}\ \otimes_I}{\Gamma_1 + \Gamma_2 \vdash *(v_1, v_2) : *(A \otimes B)}\ \text{NEC}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{push}\ (*(v_1, v_2)) : (*A) \otimes (*B)}\ \text{PUSH}$$

i.e., in the original typing $p = *$ and thus heap compatibility is $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$

The goal typing is then provided by:

$$\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A}{\Gamma_1 \vdash *v_1 : *A}\,\textsc{nec} \qquad \dfrac{\Gamma_2 \vdash v_2 : B}{\Gamma_2 \vdash *v_2 : *B}\,\textsc{nec}}{\Gamma_1 + \Gamma_2 \vdash (*v_1, *v_2) : (*A \otimes *B)}\,\otimes_I$$

with the goal heap compatibility $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$ then provided by the incoming heap compatibility.

- (pull)

$$\dfrac{\Gamma \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma \vdash \mathbf{pull}\, t : \&_p (A \otimes B)}\,\textsc{pull}$$

And three possible reductions:

1. (congPull)

$$\dfrac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{pull}\, t \rightsquigarrow_s H' \vdash \mathbf{pull}\, t'}\,\rightsquigarrow_{\textsc{pull}}$$

Inductive case as in other inductive rules.

2. (pullUnique)

$$\dfrac{}{H \vdash \mathbf{pull}\, (*v_1, *v_2) \rightsquigarrow_s H \vdash *(v_1, v_2)}\,\rightsquigarrow_{\textsc{pull}*}$$

with the refined typing:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A}{\Gamma_1 \vdash *v_1 : *A}\,\textsc{nec} \qquad \dfrac{\Gamma_2 \vdash v_2 : B}{\Gamma_2 \vdash *v_2 : *B}\,\textsc{nec}}{\Gamma_1 + \Gamma_2 \vdash (*v_1, *v_2) : (*A \otimes *B)}\,\otimes_I}{\Gamma_1 + \Gamma_2 \vdash \mathbf{pull}\, (*v_1, *v_2) : *(A \otimes B)}\,\textsc{pull}$$

thus with $p = *$ in the original typing and thus heap compatibility is $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$

The goal typing is then provided by:

$$\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A \qquad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : (A \otimes B)}\,\otimes_I}{\Gamma_1 + \Gamma_2 \vdash *(v_1, v_2) : *(A \otimes B)}\,\textsc{nec}$$

with the goal heap compatibility $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$ then provided by the incoming heap compatibility.

- (split)

$$\dfrac{\Gamma \vdash t : \&_p A}{\Gamma \vdash \mathbf{split}\, t : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\,\textsc{split}$$

And three possible reductions:

1. (congSplit)

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{split}\ t \rightsquigarrow_s H' \vdash \mathbf{split}\ t'}\ \rightsquigarrow_{\text{SPLIT}}$$

Inductive case as in other inductive rules.

2. (splitArr)

$$\frac{a_1 \# H \qquad a_2 \# H}{H, a \mapsto_p id, id \mapsto \mathbf{arr} \vdash \mathbf{split}\ (*a) \rightsquigarrow_s H, a_1 \mapsto_{\frac{p}{2}} id, a_2 \mapsto_{\frac{p}{2}} id, id \mapsto \mathbf{arr} \vdash (*a_1, *a_2)}\ \rightsquigarrow_{\text{SPLITARR}}$$

with the refined typing:

$$\frac{\dfrac{0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A \vdash a : \mathsf{Array}_{id}\ A}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A \vdash *a : \&_p B}\ \text{NEC}}{0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A \vdash \mathbf{split}\ (*a) : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\ \text{SPLIT}$$

and thus heap compatibility is $(H, id \mapsto \mathbf{arr}) \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A)$

The resulting typing is then given by:

$$\frac{\dfrac{\dfrac{0 \cdot \Gamma_1, a_1 : \mathsf{Array}_{id}\ A \vdash a_1 : \mathsf{Array}_{id}\ A}{0 \cdot \Gamma_1, a_1 : \mathsf{Array}_{id}\ A \vdash *a_1 : \&_p(\mathsf{Array}_{id}\ A)}\ \text{NEC} \quad \dfrac{0 \cdot \Gamma_2, a_2 : \mathsf{Array}_{id'}\ B \vdash a_2 : \mathsf{Array}_{id'}\ B}{0 \cdot \Gamma_2, a_2 : \mathsf{Array}_{id'}\ B \vdash *a_2 : \&_p(*(\mathsf{Array}_{id'}\ B))}\ \text{NEC}}{0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, a_1 : \mathsf{Array}_{id}\ A, a_2 : \mathsf{Array}_{id'}\ B \vdash (*v_1, *v_2) : (\&_p(\mathsf{Array}_{id}\ A) \otimes \&_p(\mathsf{Array}_{id}\ B))}\ \otimes_I$$

Goal compatibility is $(H, id \mapsto \mathbf{arr}, a_1 \mapsto_{\frac{p}{2}} id, a_2 \mapsto_{\frac{p}{2}} id) \bowtie (\Gamma_0 + s \cdot (0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, a_1 : \mathsf{Array}_{id}\ A, a_2 : \mathsf{Array}_{id'}\ B))$ which refines to

$$(H, id \mapsto \mathbf{arr}, a_1 \mapsto_{\frac{p}{2}} id, a_2 \mapsto_{\frac{p}{2}} id) \bowtie (\Gamma_0 + 0 \cdot \Gamma_1 + 0 \cdot \Gamma_2, a_1 : \mathsf{Array}_{id}\ A, a_2 : \mathsf{Array}_{id'}\ B)$$

which is constructed by:

$$\frac{\dfrac{(H, id \mapsto \mathbf{arr}) \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A)}{(H, id \mapsto \mathbf{arr}, a_1 \mapsto_{\frac{p}{2}} id) \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A), a_1 : \mathsf{Array}_{id}\ A}\ \text{EXTRES}}{((H, id \mapsto \mathbf{arr}, a_1 \mapsto_{\frac{p}{2}} id), a_2 \mapsto_{\frac{p}{2}} id) \bowtie \Gamma_0 + s \cdot (0 \cdot \Gamma, a : \mathsf{Array}_{id}\ A), a_1 : \mathsf{Array}_{id}\ A, a_2 : \mathsf{Array}_{id'}\ B}\ \text{EXTRES}$$

satisfying the goal here.

3. (splitPair)

$$\frac{\begin{array}{c} H \vdash \mathbf{split}\ (*v) \rightsquigarrow_s H' \vdash (*v_1, *v_2) \\ H' \vdash \mathbf{split}\ (*w) \rightsquigarrow_s H'' \vdash (*w_1, *w_2) \end{array}}{H \vdash \mathbf{split}\ (*(v, w)) \rightsquigarrow_s H'' \vdash (*(v_1, w_1), *(v_2, w_2))}\ \rightsquigarrow_{\text{SPLIT}\otimes}$$

with the refined typing:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v : A \quad \Gamma_2 \vdash w : B}{\Gamma_1 + \Gamma_2 \vdash (v, w) : A \otimes B}\otimes_I}{\Gamma_1 + \Gamma_2 \vdash *(v, w) : \&_p(A \otimes B)}\textsc{nec}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{split}\ (*(v, w)) : \&_{\frac{p}{2}}(A \otimes B) \otimes \&_{\frac{p}{2}}(A \otimes B)}\textsc{split}$$

and thus heap compatibility is $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$ By induction on the first premise with $\Gamma_0' = \Gamma_0 + s \cdot \Gamma_2$ and second premise with $\Gamma_0'' = \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1' + s \cdot \Gamma_1$, providing:

a) Γ_1' with $\Gamma_1' \vdash (*v_1, *v_2) : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A$ and $H' \bowtie \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1'$

b) Γ_2' with $\Gamma_2' \vdash (*w_1, *w_2) : \&_{\frac{p}{2}} B \otimes \&_{\frac{p}{2}} B$ and $H'' \bowtie \Gamma_0 + s \cdot \Gamma_2 + s \cdot \Gamma_1' + s \cdot \Gamma_1 + s \cdot \Gamma_2'$

The resulting goal type derivation is then:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash v_1 : A \quad \Gamma_1' \vdash w_1 : B}{\Gamma_1 + \Gamma_1' \vdash (v_1, w_1) : A \otimes B}\otimes_I}{\Gamma_1 + \Gamma_1' \vdash *(v_1, w_1) : \&_{\frac{p}{2}}(A \otimes B)}\textsc{nec} \quad \dfrac{\dfrac{\Gamma_2 \vdash v_2 : A \quad \Gamma_2' \vdash w_2 : B}{\Gamma_2 + \Gamma_2' \vdash (v_2, w_2) : A \otimes B}\otimes_I}{\Gamma_2 + \Gamma_2' \vdash *(v_2, w_2) : \&_{\frac{p}{2}}(A \otimes B)}\textsc{nec}}{\Gamma_1 + \Gamma_1' + \Gamma_2 + \Gamma_2' \vdash (*(v_1, w_1), *(v_2, w_2)) : \&_{\frac{p}{2}}(A \otimes B) \otimes \&_{\frac{p}{2}}(A \otimes B)}\otimes_I$$

The goal compatibility is then $H'' \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_1' + \Gamma_2 + \Gamma_2')$ which is provided by the second induction with distributivity and commutativity of $+$.

- (join)

$$\dfrac{\Gamma_1 \vdash t_1 : \&_p A \qquad \Gamma_2 \vdash t_2 : \&_q A \qquad p + q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \mathbf{join}\ t_1\ t_2 : \&_{p+q} A}\textsc{join}$$

And four possible reductions:

1. (congJoinL)

$$\dfrac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{join}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ t_1'\ t_2}\rightsquigarrow_{\textsc{joinL}}$$

Inductive case as in other inductive rules.

2. (congJoinR)

$$\dfrac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{join}\ v\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ v\ t_2'}\rightsquigarrow_{\textsc{joinR}}$$

Inductive case as in other inductive rules.

3. (joinArr)

$$\frac{a\#H}{H, a_1 \mapsto_p id, a_2 \mapsto_q id, id \mapsto \mathbf{arr} \vdash \mathbf{join}\ *a_1\ *a_2 \rightsquigarrow_s H, a \mapsto_{(p+q)} id, id \mapsto \mathbf{arr} \vdash *a}\ \rightsquigarrow_{\text{JOINARR}}$$

Dualising the splitArr cases exactly.

4. (joinPair)

$$\frac{\begin{array}{c} H \vdash \mathbf{join}\ (*v_1)\ (*v_2) \rightsquigarrow_s H' \vdash *v \\ H' \vdash \mathbf{join}\ (*w_1)\ (*w_2) \rightsquigarrow_s H'' \vdash *w \end{array}}{H \vdash \mathbf{join}\ (*(v_1, w_1))\ (*(v_2, w_2)) \rightsquigarrow_s H'' \vdash *(v, w)}\ \rightsquigarrow_{\text{JOIN}\otimes}$$

Dualising the joinPair cases exactly.

□

C.4 UNIQUENESS AND BORROW SAFETY PROOFS

Proposition 1 (Relevant references in a well-typed term are in a compatible heap)**.** *Given* $\Gamma \vdash t : A$ *and* $H \bowtie \Gamma_0 + s \cdot \Gamma$ *then if* $ref \in \mathsf{refs}(t)$ *then* $ref \in \mathsf{dom}(H)$.

Proof. Trivial induction on typing and inversion of heap compatibility; *ref* must receive a (runtime) type in Γ and thus must feature in H for the heap to be compatible. □

Lemma 15 (Irrelevant references are preserved by reduction)**.** *For* $\Gamma \vdash t : A$ *and* Γ_0 *and* H *such that* $H \bowtie (\Gamma_0 + s \cdot \Gamma)$ *and* $H \vdash t \rightsquigarrow_s H' \vdash t'$, *then for all references* $ref \in \mathsf{dom}(H) \wedge ref \notin \mathsf{refs}(t)$ *then* $\forall id.(ref \mapsto_p id \in H \implies ref \mapsto_p id \in H')$.

Proof.

- (var)

$$\frac{\exists r'.\ s + r' \sqsubseteq r}{H, x \mapsto_r v \vdash x \rightsquigarrow_s H, x \mapsto_r v \vdash v}\ \rightsquigarrow_{\text{VAR}}$$

  For all $ref \in \mathsf{dom}(H) \wedge ref \notin \mathsf{refs}(x)$, then assume $ref \mapsto_p id \in (H, x \mapsto_r v)$. Since the output heap is the same as the input heap, then the assumption provides the goal.

- (app)
  - (appR)

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash v\, t_2 \rightsquigarrow_s H' \vdash v\, t_2'}\ \rightsquigarrow_{\text{APPR}}$$

    By induction since $ref \notin \mathsf{refs}(v\, t_2)$ implies $ref \notin \mathsf{refs}(t_2)$, and the heaps of the premise are preserved in the conclusion.

– (appL)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash t_1\, t_2 \rightsquigarrow_s H' \vdash t_1'\, t_2} \rightsquigarrow_{\text{APPL}}$$

By induction since $ref \notin \mathsf{refs}(t_1\, t_2)$ implies $ref \notin \mathsf{refs}(t_2)$, and the heaps of the premise are preserved in the conclusion.

– (beta)

$$\frac{y\#\{H, v, t\}}{H \vdash (\lambda x.t)\, v \rightsquigarrow_s H, y \mapsto_s v \vdash t[y/x]} \rightsquigarrow_{\beta}$$

Trivial since the reduction does not affect any references or identifiers in the heap.

– (congPrim)

$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash pr\, t \rightsquigarrow_s H' \vdash pr\, t'} \rightsquigarrow_{\text{PRIM}}$$

Trivial since $ref \notin \mathsf{refs}(pr\, t_2)$ implies $ref \notin \mathsf{refs}(t_2)$, and the heaps of the premise are preserved in the conclusion

- (existsBeta)

$$\frac{y\#\{H, v, t\}}{H \vdash \textbf{unpack}\ \langle id, x\rangle = \textbf{pack}\ \langle id', v\rangle\ \textbf{in}\ t \rightsquigarrow_s H, y \mapsto_r v \vdash t[y/x]} \rightsquigarrow_{\exists\beta}$$

Then for $ref \in \mathsf{dom}(H) \wedge ref \notin \mathsf{refs}(\textbf{unpack}\ \langle id, x\rangle = \textbf{pack}\ \langle id', v\rangle\ \textbf{in}\ t)$ and with antecedent $ref \mapsto_p id'' \in H$ goal is $ref \mapsto_p id'' \in H'$.

Consider two possibilities:

– $id'' = id$. If this were the case then $ref : Res_{id''}\ A$ in the typing of the term and thus $ref \in \mathsf{refs}(\textbf{unpack}\ \langle id, x\rangle = \textbf{pack}\ \langle id', v\rangle\ \textbf{in}\ t)$, contradicting the premise and so this trivially holds.

– $id'' \neq id$, then the renaming here does not change id'' and thus $ref \mapsto_p id'' \in H[id'/id]$.

- (packCong)

$$\frac{H \vdash t \rightsquigarrow_s H \vdash t'}{H \vdash \textbf{pack}\ \langle id, t\rangle \rightsquigarrow_s H \vdash \textbf{pack}\ \langle id, t'\rangle} \rightsquigarrow_{\text{PACK}}$$

Trivial since $ref \notin \mathsf{refs}(\textbf{pack}\ \langle id, t\rangle)$ implies $ref \notin \mathsf{refs}(t)$, and the heaps of the premise are preserved in the conclusion

- (unpackCong), (congBoxElim), (congPromotion), (congPairL), (congPairR), (congPairElim), (congShare), (congSplit), (congJoinL), (congJoinR), (congWithBorrowL), (congWithBorrowR), (congUnborrow), (cloneCong), (congUnique), (congPush), (congPull)

Trivial like the other congruence rules (see above)

- (betaBox)

$$\frac{y\#\{H, v, t\}}{H \vdash \mathbf{let}\,[x] = [v]_r\,\mathbf{in}\,t \;\leadsto_s\; H, y \overset{}{\mapsto}_{(s*r)} v \vdash t[y/x]}\;{}^{\leadsto\Box\beta}$$

Trivially holds since no new references are added to the outgoing heap.

- (pairBeta)

$$\frac{x'\#\{H, v_1, v_2, t\} \qquad y'\#\{H, v_1, v_2, t\}}{H \vdash \mathbf{let}\,(x, y) = (v_1, v_2)\,\mathbf{in}\,t \;\leadsto_s\; H, x' \mapsto_s v_1, y' \mapsto_s v_2 \vdash t[y'/y][x'/x]}\;{}^{\leadsto\otimes\beta}$$

Trivially holds since no new references are added to the outgoing heap.

- (betaUnit)

$$\frac{}{H \vdash \mathbf{let}\,() = ()\,\mathbf{in}\,t \;\leadsto_s\; H \vdash t}\;{}^{\leadsto\beta\text{UNIT}}$$

Trivially holds since the outgoing heap is the same as incoming heap.

- (share)

$$\frac{\mathsf{dom}(H) \equiv \mathsf{refs}(v)}{H, H' \vdash \mathbf{share}\,(*v) \;\leadsto_s\; ([H]_0), H' \vdash [v]}\;{}^{\leadsto\text{SHARE}\beta}$$

For $ref \in \mathsf{dom}(H)$ and $ref \notin \mathsf{refs}(\mathbf{share}\,(*v))$ then this implies ref in $\mathsf{dom}(H')$ (as otherwise ref would be in H and get zeroed) therefore $\forall id.(ref \mapsto_p id \in H' \implies ref \mapsto_p id \in H')$ providing the goal here.

- (splitRef)

$$\frac{ref_1\#H \qquad ref_2\#H}{H, ref \mapsto_p id, id \mapsto v \vdash \mathbf{split}\,(*ref) \;\leadsto_s\; H, ref_1 \mapsto_{\frac{p}{2}} id, ref_2 \mapsto_{\frac{p}{2}} id, id \mapsto v \vdash (*ref_1, *ref_2)}\;{}^{\leadsto\text{SPLITREF}}$$

Let the incoming heap be $H_0 = H, id \mapsto v, ref \mapsto_p id$ For $ref' \in \mathsf{dom}(H_0)$ and $ref' \notin \mathsf{refs}(\mathbf{split}\,(*ref))$ then this implies $ref' \neq ref$ and thus $ref' \in \mathsf{dom}(H)$, therefore $\forall id.(ref' \mapsto_p id \in H_0 \implies ref' \mapsto_p id \in H_0)$ trivially providing the goal here.

- (joinRef)

$$\frac{ref\#H}{H, ref_1 \mapsto_p id, ref_2 \mapsto_q id, id \mapsto v \vdash \mathbf{join}\,(*ref_1)\,(*ref_2) \;\leadsto_s\; H, ref \mapsto_{(p+q)} id, id \mapsto v \vdash *ref}\;{}^{\leadsto\text{JOINREF}}$$

Let the incoming heap be $H_0 = \mathsf{dom}(H, id \mapsto v, ref_1 \mapsto_p id, ref_2 \mapsto_q id)$. For $ref' \in H_0$ and $ref' \notin \mathsf{refs}(\mathbf{join}\,(*ref_1)\,(*ref_2))$ then this implies $ref' \neq ref$ and thus $ref' \in \mathsf{dom}(H)$, therefore $\forall id.(ref' \mapsto_p id \in H_0 \implies ref' \mapsto_p id \in H_0)$ trivially providing the goal here.

- (splitPair)

$$\frac{H \vdash \mathbf{split}\ (*v) \rightsquigarrow_s H' \vdash (*v_1, *v_2) \qquad H' \vdash \mathbf{split}\ (*w) \rightsquigarrow_s H'' \vdash (*w_1, *w_2)}{H \vdash \mathbf{split}\ (*(v,w)) \rightsquigarrow_s H'' \vdash (*(v_1,w_1), *(v_2,w_2))}\ \rightsquigarrow_{\text{SPLIT}\otimes}$$

  By induction over the first premise, and transitivity with induction on the second premise.

- (joinPair) Similarly to (splitPair).

- (withBorrow)

$$\frac{y\#\{H,v,t\}}{H \vdash \mathbf{withBorrow}\ (\lambda x.t)\ (*v) \rightsquigarrow_s H, y \mapsto_s (*v) \vdash \mathbf{unborrow}\ t[y/x]}\ \rightsquigarrow_{\text{WITH\&}}$$

  Trivially holds since no new references are added to the outgoing heap.

- (unborrowBorrow)

$$\frac{}{H \vdash \mathbf{unborrow}\ (*v) \rightsquigarrow_s H \vdash *v}\ \rightsquigarrow_{\text{UN\&}}$$

  Trivial since the input heap and output heap are the same.

- (newRef)

$$\frac{ref\#H \qquad id\#H}{H \vdash \mathbf{newRef}\, v \rightsquigarrow_s H, ref \mapsto_1 id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{pack}\ \langle id, *ref \rangle}\ \rightsquigarrow_{\text{NEWREF}}$$

  Trivially holds since heap references not changed.

- (swapRef)

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{swapRef}\ (*ref)\, v' \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v') \vdash v}\ \rightsquigarrow_{\text{SWAPREF}}$$

  Trivially holds since heap H whose references are not in the term is preserved into the outgoing heap.

- (freezeRef)

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{freezeRef}\ (*ref) \rightsquigarrow_s H \vdash v}\ \rightsquigarrow_{\text{FREEZEREF}}$$

  Trivially holds since heap H whose references are not in the term is preserved into the outgoing heap.

- (readRef)

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_{r+1}) \vdash \mathbf{readRef}\ (*ref) \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_r) \vdash (v, *ref)}\ \rightsquigarrow_{\text{READREF}}$$

  Trivially holds since heap H whose references are not in the term is preserved into the outgoing heap.

- (newArray), (readArray), (writeArray), (deleteArray) follow in a similar way to the polymorphic reference counterparts above.
- (copyBeta)

$$\frac{\mathsf{dom}(H') \equiv \mathsf{refs}(v) \qquad (H'', \theta, \overline{id}) = \mathsf{copy}(H') \qquad y\#\{H, v, t\}}{H, H' \vdash \mathbf{clone}\,[v]_r\,\mathbf{as}\,x\,\mathbf{in}\,t \;\leadsto_s\; H, H', H'', y{\mapsto_s}\mathbf{pack}\,\langle \overline{id}, *(\theta(v))\rangle \vdash t[y/x]} \leadsto_{\text{CLONE}\beta}$$

  then for $ref \in \mathsf{dom}(H, , H')$ and $ref \notin \mathsf{refs}(\mathbf{clone}\,[v]\,\mathbf{as}\,x\,\mathbf{in}\,t)$ then for id where $(ref{\mapsto_p}id \in H, H'$ then we have $ref{\mapsto_p}id \in (H, H'), H'')$ since the heap is only extended not changed.
- (pushUnique), (pullUnique) Trivial since the incoming and outgoing heaps are the same.

□

Lemma 16 (Borrow safety)**.** *For a well-typed term $\Gamma \vdash t : A$ and all Γ_0, s, and heaps H such that $H \bowtie (\Gamma_0 + s \cdot \Gamma)$, and given a single-step reduction $H \vdash t \leadsto_s H' \vdash t'$ then for all $id \in \mathsf{dom}(H)$:*

$$\sum_{\substack{\forall ref \in \mathsf{refs}(t).\\ ref{\mapsto_p}id \in H}} p \;= 1 \quad\Longrightarrow\quad \sum_{\substack{\forall ref \in \mathsf{refs}(t').\\ ref{\mapsto_{p'}}id \in H'}} p' \;\in \{0, 1\}$$

i.e., for all resources with identifier id in the incoming heap and all references in the term pointing to this resource, if the sum of all permissions pointing to this resource are 1 in the incoming heap then either this is preserved in the outgoing heap or *the total permissions in the output heap is 0, i.e., this resource has now been fully shared and has no ownership tracking.*

Furthermore, any resources in the outgoing heap that did not appear in the initial heap with references in the final term should have permissions summing to 1. That is, for all $id' \in \mathsf{dom}(H') \wedge id' \notin \mathsf{dom}(H)$:

$$\sum_{\substack{\forall ref' \in \mathsf{refs}(t').\\ ref'{\mapsto_q}id' \in H'}} q \;= 1$$

Proof. By induction on typing $\Gamma \vdash t : A$.

- (var)

$$\frac{}{0 \cdot \Gamma, x : A \vdash x : A}\ \text{VAR}$$

  Then we also have a heap H such that $H \bowtie (0 \cdot \Gamma, x : A)$.

  By inversion of heap compatibility, which implies that there exists a subheap H_1 such that $H = H_1, x{\mapsto_r}(\Gamma' \vdash v : A)$ (where there exists some r' such that $r' + 1 \sqsubseteq r$, and since $\downarrow v = \emptyset$):

$$\frac{H_1 \bowtie 0 \cdot \Gamma + \Gamma' + \downarrow v \qquad x \notin \mathsf{dom}(H_1) \qquad \Gamma' \vdash v : A \qquad \exists r'.\, r' + s \equiv r}{(H_1, x{\mapsto_r}(\Gamma' \vdash v : A)) \bowtie (0 \cdot \Gamma, x : [A]_s)}\ \text{EXT}$$

and we have the reduction:

$$\frac{\exists r'.\ r' + s \sqsubseteq r}{H_1, x \mapsto_r v \vdash x \rightsquigarrow_s H_1, x \mapsto_r v \vdash v}\ \rightsquigarrow_{\text{VAR}}$$

For part 1 of the lemma, for all $id \in \mathsf{dom}(H)$ and for all $ref \in \mathsf{refs}(t)$ then we assume the antecedent:

$$\sum_{ref \mapsto_p id \in H} p = 1$$

Since the heap is unchanged by the reduction, $H' = H$ and then we trivially conclude with the antecedent evidence:

$$\sum_{ref \mapsto_p id \in H'} p = 1$$

For part 2, we have:

$$\forall id' \in \mathsf{dom}(H'), ref \in \mathsf{refs}(v).\ \left(ref \notin \mathsf{dom}(H) \implies \sum_{ref \mapsto_q id' \in H'} q = 1 \right)$$

trivially since the premise must always be false by heap compatibility and that the heap is preserved by the reduction here.

- (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textbf{ABS}$$

Is trivial since there are no possible reductions.

- (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textbf{APP}$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and three reductions (beta and application congruence on the left and right) then several possible reductions following from primitive applications:

1.

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash t_1\, t_2 \rightsquigarrow_s H' \vdash t_1'\, t_2}\ \rightsquigarrow_{\text{APPL}}$$

For part 1, for $id \in \mathsf{dom}(H)$ and $ref \in \mathsf{refs}(t_1\, t_2)$, assuming the antecedent

$$\sum_{ref \mapsto_p id \in H} p = 1$$

From this, since $\mathsf{refs}(t_1) \subseteq \mathsf{refs}(t_1\, t_2)$, we induct on the premise reduction to attain that:

(1) $\displaystyle\sum_{ref \mapsto_p id \in H'} p \in \{0, 1\}$

(2) $\forall id' \in \mathsf{dom}(H'), ref \in \mathsf{refs}(t_2').\ \left(ref \notin \mathsf{dom}(H) \implies \displaystyle\sum_{ref \mapsto_q id' \in H'} q = 1 \right)$

To conclude then we use (1) but also need that $\forall ref' \in \mathsf{refs}(t_2). \sum_{ref' \mapsto_p id \in H'} p \in \{0,1\}$. We consider two cases depending on whether $ref' \in \mathsf{refs}(t_1)$ or not:

- $ref' \in \mathsf{refs}(t_1)$, therefore by (1) we also have that $\sum_{a' \mapsto_p id \in H'} p \in \{0,1\}$, satisfying the goal.
- $ref' \notin \mathsf{refs}(t_1)$ therefore $\sum_{ref' \mapsto_p id \in H'} p = 1$ satisfying the goal.

Second goal is that:

$$\forall id' \in \mathsf{dom}(H'), ref \in \mathsf{refs}(t_1\, t_2'). \left(ref \notin \mathsf{dom}(H) \implies \sum_{ref \mapsto_q id' \in H'} q = 1 \right)$$

Thus we can conclude with (2) along with the cases for $\forall ref' \in \mathsf{refs}(t_1)$, for which we discriminate based on whether ref' is contained also in t_2':

a) $ref' \in \mathsf{refs}(t_2')$ therefore we have from (2) that:

$$\left(ref' \notin \mathsf{dom}(H) \implies \sum_{ref' \mapsto_q id' \in H'} q = 1 \right)$$

satisfying the goal.

b) $ref' \notin \mathsf{refs}(t_2')$ then by Proposition 1 with $\Gamma_1 + \Gamma_2 \vdash t_1\, t_2' : B$ (Type preservation) and $ref' \in \mathsf{refs}(t_1\, t_2')$ and $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$ then $ref' \in \mathsf{dom}(H)$ and thus the antecedent of condition (2) is false and so is trivially true.

2.

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash v\, t_2 \rightsquigarrow_s H' \vdash v\, t_2'} \rightsquigarrow_{\text{APPR}}$$

By the same reasoning as above for the (appL) case *mutatis mutandis*: with t_1 being a value and induction happening on the right-hand term t_2's reduction.

3. Alternatively $t_1 = \lambda x.t_1'$ such that typing is:

$$\frac{\dfrac{\Gamma_1, x : A \vdash t' : B}{\Gamma_1 \vdash t' : A \multimap B}\,\textsc{abs} \qquad \Gamma_2 \vdash v : A}{\Gamma_1 + \Gamma_2 \vdash (\lambda x.t')\, v : B}\,\textsc{app}$$

and we have reduction:

$$\frac{y \# \{H, v, t\}}{H \vdash (\lambda x.t)\, v \rightsquigarrow_s H, y \mapsto_s v \vdash t[y/x]} \rightsquigarrow_\beta$$

(Part 1) Then for $id \in \mathsf{dom}(H)$ and $ref \in \mathsf{refs}((\lambda x.t')\, v)$, we assume the antecedent: $\sum_{ref \mapsto_p id \in H} p = 1$.

Since the right-hand side of reduction does not introduce any new references then the goal follows from the antecedent: $\sum_{ref \mapsto_p id \in H'} = 1$.

(Part 2) Trivial since the antecedent is always false as $\mathsf{refs}(t[y/x]) \subseteq \mathsf{refs}((\lambda x.t)\, v)$.

4. $t_1 = \mathbf{newRef}$

   Therefore we induct on the second argument:

   – t_2 is a value and thus the typing is:

   $$\frac{}{0 \cdot \Gamma \vdash \mathbf{newRef} : A \multimap \exists id.*(\mathsf{Ref}_{id}\ A)}\ \textsc{newRef}$$

   with $H \bowtie (\Gamma_0 + \Gamma)$.

   Thus there is a reduction as follows:

   $$\frac{ref \# H \qquad id \# H}{H \vdash \mathbf{newRef}\, v \rightsquigarrow_s H, ref \mapsto_1 id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{pack}\ \langle id, *ref \rangle}\ \rightsquigarrow_{\textsc{newRef}}$$

   Part 1 follows as all id in the input heap and references pointing to them are preserved in the output heap.

   Part 2 follows since where $ref \notin \mathsf{dom}(H)$ and we have that $ref \mapsto_1 id \in H, ref \mapsto_1 id$ satisfying the goal.

   – t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

   $$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{newRef}\, t_2 \rightsquigarrow_s H' \vdash \mathbf{newRef}\, t_2'}\ \rightsquigarrow_{\textsc{prim}}$$

   Then the result holds by induction.

5. $t_1 = \mathbf{swapRef}$

   and there is a reduction:

   $$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{swapRef}\, t_2 \rightsquigarrow_s H' \vdash \mathbf{swapRef}\, t_2'}\ \rightsquigarrow_{\textsc{prim}}$$

   Therefore the borrow safety result holds by induction.

6. $t_1 = \mathbf{swapRef}\ (*ref)$ Case on progress for t_2:

   – t_2 is a value and we have a reduction:

   $$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{swapRef}\ (*ref)\ v' \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v') \vdash v}\ \rightsquigarrow_{\textsc{swapRef}}$$

   (Part 1) Trivially true since the only change to the heap is the value that id is pointing to.

   (Part 2) No new references are created so trivially true.

– t_2 is not a value and therefore has a reduction from which we form the congruence:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{swapRef}\,(*ref)\,t_2 \rightsquigarrow_s H' \vdash \mathbf{swapRef}\,(*ref)\,t_2'} \rightsquigarrow_{\text{PRIM}}$$

7. $t_1 = \mathbf{freezeRef}$ Case on progress for t_2:
   – t_2 is a value therefore by the value lemma $t_2 = *ref$ and we have a reduction:

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}(v) \vdash \mathbf{freezeRef}\,(*ref) \rightsquigarrow_s H \vdash v} \rightsquigarrow_{\text{FREEZEREF}}$$

   (Part 1) Thus, for $id' \in \mathsf{dom}(H), ref' \in \mathsf{refs}(t)$ with assumption $\sum_{ref' \mapsto_p id' \in H} p = 1$.
   If $id' = id$ and $ref' = ref$ then we can conclude with: $\sum_{ref' \mapsto_p id' \in H'} p = 0$ since the reference and identifier assignments are removed from the output heap. Otherwise, the heap is preserved so $\sum_{ref' \mapsto_p id' \in H'} p = 1$
   (Part 2) No new references are created so trivially true.
   – t_2 is not a value and therefore has a reduction from which we form the congruence:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{freezeRef}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{freezeRef}\,t_2'} \rightsquigarrow_{\text{PRIM}}$$

8. $t_1 = \mathbf{readRef}$ Case on progress for t_2:
   – t_2 is a value therefore by the value lemma $t_2 = *ref$ and we have a reduction:

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_{r+1}) \vdash \mathbf{readRef}\,(*ref) \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{ref}([v]_r) \vdash (v, *ref)} \rightsquigarrow_{\text{READREF}}$$

   (Part 1) Trivially true since the heap is preserved.
   (Part 2) No new references are created so trivially true.
   – t_2 is not a value and therefore has a reduction from which we form the congruence:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{readRef}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{readRef}\,t_2'} \rightsquigarrow_{\text{PRIM}}$$

9. $t_1 = \mathbf{newArray}$ therefore $A = \mathbb{N}$
   Therefore we induct on the second argument:
   – t_2 is a value and therefore by the value lemma (Lemma 10) $t_2 = n$ and thus the typing is:

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathbf{newArray}\,n : *(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{NEWARRAY}$$

with $H \bowtie (\Gamma_0 + s \cdot \Gamma)$.

Thus there is a reduction as follows:

$$\frac{ref \# H \qquad id \# H}{H \vdash \mathbf{newArray}\, n \rightsquigarrow_s H, ref \mapsto_1 id, id \mapsto \mathsf{init} \vdash \mathbf{pack}\, \langle id, *ref \rangle} \rightsquigarrow_{\text{NEWARRAY}}$$

Part 1 follows as all id in the input heap and references pointing to them are preserved in the output heap.

Part 2 follows since where $ref \notin \mathsf{dom}(H)$ and we have that $ref \mapsto_1 id \in H, ref \mapsto_1 id$ satisfying the goal.

– t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{newArray}\, t_2 \rightsquigarrow_s H' \vdash \mathbf{newArray}\, t_2'} \rightsquigarrow_{\text{PRIM}}$$

Then the result holds by induction.

10. $t_1 = \mathbf{readArray}$ therefore $A = \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \otimes \&_p(\mathsf{Array}_{id}\ \mathbb{F})$

and there is a reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{readArray}\, t_2 \rightsquigarrow_s H' \vdash \mathbf{readArray}\, t_2'} \rightsquigarrow_{\text{PRIM}}$$

Therefore the borrow safety result holds by induction.

11. $t_1 = \mathbf{readArray}\,(*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \otimes *(\mathsf{Array}_{id}\ \mathbb{F})$

– t_2 is a value and therefore by the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{N}$ (Lemma 10) implies $t_2 = n$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \text{REF}}{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash *a : *(\mathsf{Array}_{id}\ \mathbb{F})}\ *\text{REF}^* \qquad \Gamma_2 \vdash n : \mathbb{N}}{[\Gamma_1] + \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{readArray}\,(*a)\, n : \mathbb{F} \otimes *(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{READARRAY}$$

with $H' \bowtie (\Gamma_0 + s \cdot ([\Gamma_1] + \Gamma_2, a : \mathsf{Array}_{id}\ \mathbb{F}))$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_p id, id \mapsto \mathbf{arr}$.

Then there is a reduction as follows:

$$\frac{}{H, ref \mapsto_p id, id \mapsto \mathbf{arr}[n] = v \vdash \mathbf{readArray}\,(*ref)\, n \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{arr}[n] = v \vdash (v, *ref)} \rightsquigarrow_{\text{READARRAY}}$$

As the entirety of the heap is preserved (including the array reference a being read from here), the goal follows trivially.

– t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{readArray}\,(*a)\,t_2 \rightsquigarrow_s H' \vdash \mathbf{readArray}\,(*a)\,t_2'} \rightsquigarrow_{\text{PRIM}}$$

And therefore the borrow safety result holds by induction.

12. $t_1 = \mathbf{writeArray}$ therefore $A = \&_p(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathbb{N} \multimap \mathbb{F} \multimap \&_p(\mathsf{Array}_{id}\ \mathbb{F})$

with reduction

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{writeArray}\,t_2'} \rightsquigarrow_{\text{PRIM}}$$

Therefore the borrow safety result holds by induction.

13. $t_1 = \mathbf{writeArray}\,(*a)$ therefore $A = \mathbb{N} \multimap \mathbb{F} \multimap *(\mathsf{Array}_{id}\ \mathbb{F})$

with reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{writeArray}\,(*a)\,t_2 \rightsquigarrow_s H' \vdash \mathbf{writeArray}\,(*a)\,t_2'} \rightsquigarrow_{\text{PRIM}}$$

Therefore the borrow safety result holds by induction.

14. $t_1 = \mathbf{writeArray}\,(*a)\,i$ therefore $A = \mathbb{F} \otimes \&_1(\mathsf{Array}_{id}\ \mathbb{F})$

Then we case on progress for t_2:

– t_2 is a value and therefore by the value lemma on $\Gamma_2 \vdash t_2 : \mathbb{F}$ (Lemma 10) implies $t_2 = f$ and thus the typing is refined at runtime as follows:

$$\frac{\dfrac{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash a : (\mathsf{Array}_{id}\ \mathbb{F})\ \text{REF}}{[\Gamma_1], a : \mathsf{Array}_{id}\ \mathbb{F} \vdash (*a) : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{NEC} \qquad \Gamma_2 \vdash i : \mathbb{N} \quad \Gamma_3 \vdash f : \mathbb{F}}{[\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F} \vdash \mathbf{writeArray}\,(*a)\,i\,f : \&_p(\mathsf{Array}_{id}\ \mathbb{F})}\ \text{WRITEARRAY}$$

with $H' \bowtie (\Gamma_0 + s \cdot [\Gamma_1] + \Gamma_2 + \Gamma_3, a : \mathsf{Array}_{id}\ \mathbb{F})$, and by the heap compatibility rule for array references there exists some H such that $H' = H, a \mapsto_p id, id \mapsto \mathbf{arr}$.

Then there is a reduction as follows:

$$\frac{}{\begin{array}{l} H, ref \mapsto_p id, id \mapsto \mathbf{arr} \vdash \mathbf{writeArray}\,(*ref)\,n\,v \\ \quad \rightsquigarrow_s H, ref \mapsto_p id, id \mapsto \mathbf{arr}[n] = v \vdash *ref \end{array}} \rightsquigarrow_{\text{WRITEARRAY}}$$

Here, the only change in the heap is to the array value that id is pointing to. As the remainder of the heap is preserved, both parts of the goal follow trivially.

– t_2 is not a value and thus has a reduction, therefore we can build the compound reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \textbf{writeArray}\,(*a)\,i\,t_2 \rightsquigarrow_s H' \vdash \textbf{writeArray}\,(*a)\,i\,t_2'}\ \rightsquigarrow_{\text{PRIM}}$$

Therefore the borrow safety result holds by induction.

15. $t_1 = \textbf{deleteArray}$ therefore $A = *(\mathsf{Array}_{id}\ \mathbb{F}) \multimap \mathsf{unit}$

Case on progress for t_2:

– t_2 is a value therefore by the value lemma $t_2 = *ref$ and we have a reduction:

$$\frac{}{H, ref \mapsto_p id, id \mapsto \textbf{arr} \vdash \textbf{deleteArray}\,(*ref) \rightsquigarrow_s H \vdash ()}\ \rightsquigarrow_{\text{DELETEARRAY}}$$

(Part 1) Thus, for $id' \in \mathsf{dom}(H), ref' \in \mathsf{refs}(t)$ with assumption $\sum_{ref' \mapsto_p id' \in H} p = 1$.

If $id' = id$ and $ref' = ref$ then we can conclude with: $\sum_{ref' \mapsto_p id' \in H'} p = 0$ since the reference and identifier assignments are removed from the output heap. Otherwise, the heap is preserved so $\sum_{ref' \mapsto_p id' \in H'} p = 1$

(Part 2) No new references are created so trivially true.

– t_2 is not a value and therefore has a reduction from which we form the congruence:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \textbf{deleteArray}\,t_2 \rightsquigarrow_s H' \vdash \textbf{deleteArray}\,t_2'}\ \rightsquigarrow_{\text{PRIM}}$$

- (pr)

$$\frac{\Gamma \vdash t : A \qquad \neg\mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t]_r : \square_r A}\ \text{PR}$$

with a heap H such that $H \bowtie (\Gamma_0 + s \cdot (r \cdot \Gamma))$ and reduction:

$$H \vdash [t]_r \rightsquigarrow_s H' \vdash t''$$

which has only one possible derivation:

$$\frac{H \vdash t \rightsquigarrow_{s*r} H' \vdash t'}{H \vdash [t]_r \rightsquigarrow_s H' \vdash [t']_r}\ \rightsquigarrow_{\square}$$

Thus, induction provides the goal, where the incoming heap compatibility by provides the inductive heap compatibility as $H \bowtie \Gamma_0 + (s * r) \cdot \Gamma_0$ by associativity of $*$.

- (elim)

$$\frac{\Gamma_1 \vdash t_1 : \square_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\,[x] = t_1\,\textbf{in}\,t_2 : B}\ \text{ELIM}$$

with a heap H such that $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash t'$$

which has two possible derivations:

1. $$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2}\ \rightsquigarrow_{\text{LET}\Box}$$

   Then induction provides the goal following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\text{APPL}}$ case.

2. Alternatively $t_1 = [v]$ such that the typing is:

   $$\frac{\dfrac{\Gamma_1 \vdash v : A}{r \cdot \Gamma_1 \vdash [v]_r : \Box_r A}\ \text{PR} \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{r \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = [v]_r\,\mathbf{in}\,t_2 : B}\ \text{ELIM}$$

   and we have reduction:

   $$\frac{y\#\{H, v, t\}}{H \vdash \mathbf{let}\,[x] = [v]_r\,\mathbf{in}\,t \rightsquigarrow_s H, y \mapsto_{(s*r)} v \vdash t[y/x]}\ \rightsquigarrow\Box\beta$$

   (where $\Gamma = r \cdot \Gamma_1$ in the above.

   Which trivially satisfies the goal since all references are then in H and we get the conditions trivially (since no references are manipulated) similar to the β proof above.

- (der)

  $$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \text{DER}$$

  with a heap H such that $H \bowtie \Gamma_0 + s \cdot (\Gamma, x : [A]_1)$ and reduction:

  $$H \vdash t \rightsquigarrow_s H' \vdash t'$$

  As the term t is unchanged from the premise, the goal holds by induction regardless of how this reduction is derived.

- (approx)

  $$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \text{APPROX}$$

  with a heap H such that $H \bowtie (\Gamma, x : [A]_s, \Gamma')$ and reduction:

  $$H \vdash t \rightsquigarrow_s H' \vdash t'$$

  As the term t is unchanged from the premise, the goal holds by induction regardless of how this reduction is derived.

- (pairIntro)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B} \otimes_I$$

with a heap H such that $H \bowtie \Gamma_0 + s \cdot (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash (t_1, t_2) \rightsquigarrow_s H' \vdash t''$$

which has two possible derivations:

1.
$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash (t_1, t_2) \rightsquigarrow_s H' \vdash (t_1', t_2)} \rightsquigarrow_{\otimes \mathrm{L}}$$

   Here, induction on t_1 provides the goal following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\mathrm{APPL}}$ case.

2. Otherwise, $t_1 = v$ for some value v and we can perform the reduction:

$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash (v, t_2) \rightsquigarrow_s H' \vdash (v, t_2')} \rightsquigarrow_{\otimes \mathrm{R}}$$

   Here, induction on t_2 provides the goal, following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\mathrm{APPL}}$ case.

- (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 : C} \otimes_E$$

Two possible reductions:

1.
$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{let}\,(x, y) = t_1\,\mathbf{in}\,t_2 \rightsquigarrow_s H' \vdash \mathbf{let}\,(x, y) = t_1'\,\mathbf{in}\,t_2} \rightsquigarrow_{\mathrm{LET}\otimes}$$

   and induction provides the goal, following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\mathrm{APPL}}$ case.

2. otherwise $t_1 = (v_1, v_2)$ such that the typing is:

$$\frac{\dfrac{\Gamma_3 \vdash v_1 : A \quad \Gamma_4 \vdash v_2 : B}{\Gamma_3 + \Gamma_4 \vdash (v_1, v_2) : (A \otimes B)} \otimes_I \qquad \Gamma_2, x : A, y : B \vdash t_2 : C}{\Gamma_3 + \Gamma_4 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = (v_1, v_2)\,\mathbf{in}\,t_2 : C} \otimes_E$$

   and we have reduction:

$$\frac{\Gamma_1 \vdash t_1' : A \quad \Gamma_2 \vdash t_1'' : B}{H \vdash \mathbf{let}\,(x, y) = (t_1', t_1'')\,\mathbf{in}\,t_3 \rightsquigarrow_s H, x \overset{s}{\mapsto} (\Gamma_1 \vdash t_1' : A), y \overset{s}{\mapsto} (\Gamma_2 \vdash t_1'' : B) \vdash t_3} \rightsquigarrow_{\otimes\beta}$$

Then (part 1) for all $id \in \mathsf{refs}(H)$ and all $ref \in \mathsf{refs}(t_3)$ we have:

$$\sum_{\substack{\forall ref \in \mathsf{refs}(t). \\ ref \mapsto_p id \in H}} p \quad = \quad \sum_{\substack{\forall ref \in \mathsf{refs}(t'). \\ ref \mapsto_{p'} id \in H'}} p'$$

satisfying the goal and (part 2) for all $id' \in \mathsf{dom}(H') \wedge id' \notin \mathsf{dom}(H)$:

$$\sum_{\substack{\forall ref' \in \mathsf{refs}(t'). \\ ref' \mapsto_q id' \in H'}} q \quad = 1$$

since no references are introduced.

- (unitIntro)

$$\frac{}{0 \cdot \Gamma \vdash () : \mathsf{unit}} \; 1_I$$

The result type here is 1, so cannot contain any types of the form $\&_p A$, and therefore the result holds trivially.

- (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ () = t_1\ \mathbf{in}\ t_2 : B} \; 1_E$$

Following essentially the same structure as the tensor proof where array reference counting is not used so induction provides the goal.

- (share)

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \mathbf{share}\, t : \Box_r A} \; \textsc{share}$$

with a heap H such that $H \bowtie \Gamma_0 + s \cdot \Gamma$ and reduction:

$$H \vdash \mathbf{share}\, t \rightsquigarrow_s H' \vdash t''$$

which has two possible derivations:

1. $$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \mathbf{share}\, t \rightsquigarrow_s H' \vdash \mathbf{share}\, t'} \; \rightsquigarrow_{\textsc{share}}$$

   Here, induction provides the goal the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\textsc{appl}}$ case.

2. t has the form $*v$ for some value v, and we can reduce:

   $$\frac{\mathsf{dom}(H) \equiv \mathsf{refs}(v)}{H, H' \vdash \mathbf{share}\,(*v) \rightsquigarrow_s ([H]_0), H' \vdash [v]} \; \rightsquigarrow_{\textsc{share}\beta}$$

   (Part 1) Then for $id \in \mathsf{dom}(H, H'), ref \in \mathsf{refs}(\mathbf{share}\,(*v))$, we assume the antecedent $\sum_{ref \mapsto_p id \in H, H'} p = 1$.

   Then we have two cases:

- $ref \in \mathsf{dom}(H)$ therefore in the output heap $\sum_{ref \mapsto_p id \in ([H]_0), H'} p = 0$ by the zeroing of H here (thus $p = 0$), satisfying the goal.
- $ref \in \mathsf{dom}(H')$ then $\sum_{ref \mapsto_p id \in ([H]_0), H'} p = 1$ following from the antecedent.

(Part 2) No new references are introduced in the output term, so this trivially holds.

- (clone)

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : *A \vdash t_2 : \Box_r B \qquad 1 \sqsubseteq r}{\Gamma_1 + \Gamma_2 \vdash \textbf{clone'}\ t_1\ \textbf{as}\ x\ \textbf{in}\ t_2 : \Box_r B}\ \textsc{clone}'$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash \textbf{clone}\, t_1\, \textbf{as}\, x\, \textbf{in}\, t_2 \rightsquigarrow_s H' \vdash t''$$

which has two possible derivations:

1. $$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \textbf{clone}\, t_1\, \textbf{as}\, x\, \textbf{in}\, t_2 \rightsquigarrow_s H' \vdash \textbf{clone}\, t_1'\, \textbf{as}\, x\, \textbf{in}\, t_2}\ \rightsquigarrow_{\textsc{clone}}$$

   Here, induction provides the goal, following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\textsc{appl}}$ case.

2. t_1 has the form $[v]$ for some value v, and we can reduce:

$$\frac{\mathsf{dom}(H') \equiv \mathsf{refs}(v) \qquad (H'', \theta, \overline{id}) = \mathsf{copy}(H') \qquad y\#\{H, v, t\}}{H, H' \vdash \textbf{clone}\, [v]_r\, \textbf{as}\, x\, \textbf{in}\, t \rightsquigarrow_s H, H', H'', y \mapsto_s \textbf{pack}\, \langle \overline{id}, *(\theta(v)) \rangle \vdash t[y/x]}\ \rightsquigarrow_{\textsc{clone}\beta}$$

   Here, all of the references from the original heap H (separated out as H') are copied and given a new identifier, to form H''. In other words, for every $ref \mapsto_p id, id \mapsto v$ in H, there now exists a new ref' and id' in H'' such that $ref' \mapsto_p id', id' \mapsto v$.

   Therefore, for all $id \in \mathsf{dom}(H)$ we have:

$$\sum_{\substack{\forall ref \in \mathsf{refs}(t). \\ ref \mapsto_p id \in H}} p = 1 \implies \sum_{\substack{\forall ref \in \mathsf{refs}(t'). \\ ref \mapsto_{p'} id \in H'}} p' = 1$$

   because none of the references in the preexisting heap have been modified in the new heap, and there are no new references with the same identifiers as any references appearing in H'' have fresh identifiers.

We also have that for all $id' \in \mathsf{dom}(H') \wedge id' \notin \mathsf{dom}(H)$:

$$\sum_{\substack{\forall ref' \in \mathsf{refs}(t'). \\ ref' \mapsto_q id' \in H'}} q \quad = 1$$

because any $ref' \notin \mathsf{dom}(H)$ must be in H'', and so it matches up with another reference ref appearing in H, for which we have $\sum_{\substack{\forall ref \in \mathsf{refs}(t). \\ ref \mapsto_p id \in H}} p \quad = 1$ by the above argument.

- (withBorrow)

$$\frac{\Gamma_1 \vdash t_1 : *A \qquad \Gamma_2 \vdash t_2 : \&_1 A \multimap \&_1 B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{withBorrow}\ t_1\ t_2 : *B}\ \textsc{withBorrow}$$

with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:

$$H \vdash \mathbf{withBorrow}\ f\ t \rightsquigarrow_s H' \vdash t''$$

which has three possible derivations:

1.
$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{withBorrow}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \mathbf{withBorrow}\ t_1'\ t_2}\ \rightsquigarrow_{\textsc{with\&L}}$$
Here, induction on t_1 provides the goal, following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\textsc{appL}}$ case.

2. f has the form $(\lambda x.t_1)$, and we can reduce:
$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{withBorrow}\ (\lambda x.t_1)\ t_2 \rightsquigarrow_s H' \vdash \mathbf{withBorrow}\ (\lambda x.t_1)\ t_2'}\ \rightsquigarrow_{\textsc{with\&R}}$$
Here, induction on t_2 provides the goal, following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\textsc{appL}}$ case.

3. As above, but t also has the form $(*v)$, and we can reduce:
$$\frac{y\#\{H, v, t\}}{H \vdash \mathbf{withBorrow}\ (\lambda x.t)\ (*v) \rightsquigarrow_s H, y \mapsto_s (*v) \vdash \mathbf{unborrow}\ t[y/x]}\ \rightsquigarrow_{\textsc{with\&}}$$
Here, the goal is trivial since the heap H is preserved modulo new variable bindings.

- (split)

$$\frac{\Gamma \vdash t : \&_p A}{\Gamma \vdash \mathbf{split}\ t : \&_{\frac{p}{2}} A \otimes \&_{\frac{p}{2}} A}\ \textsc{split}$$

with a heap H such that $H \bowtie \Gamma$ and reduction:

$$H \vdash \mathbf{split}\ t \rightsquigarrow_s H' \vdash t''$$

which has four possible derivations:

1.
$$\frac{H \vdash t \rightsquigarrow_s H' \vdash t'}{H \vdash \textbf{split}\ t \rightsquigarrow_s H' \vdash \textbf{split}\ t'} \rightsquigarrow_{\text{SPLIT}}$$
Here, induction provides the goal, following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\text{APPL}}$ case.

2. t has the form $(*ref)$ such that the typing is:
$$\frac{[\Gamma], ref : \mathsf{Ref}_{id}\ A \vdash (*ref) : \&_p(\mathsf{Ref}_{id}\ A)\ *\text{REF}^*}{[\Gamma], ref : \mathsf{Ref}_{id}\ A \vdash \textbf{split}\ (*ref) : \&_{\frac{p}{2}}(\mathsf{Ref}_{id}\ A) \otimes \&_{\frac{p}{2}}(\mathsf{Ref}_{id}\ A)}\ \text{SPLIT}$$
and we have reduction:
$$\frac{ref_1 \# H \qquad ref_2 \# H}{H, ref \mapsto_p id, id \mapsto v \vdash \textbf{split}\ (*ref) \rightsquigarrow_s H, ref_1 \mapsto_{\frac{p}{2}} id, ref_2 \mapsto_{\frac{p}{2}} id, id \mapsto v \vdash (*ref_1, *ref_2)} \rightsquigarrow_{\text{SPLITREF}}$$
Thus for $id' \in \mathsf{dom}(H, id \mapsto v, ref \mapsto_p id)$ and $ref' \in \mathsf{refs}(\textbf{split}\ (*ref))$ we assume the antecedent $\sum_{ref' \mapsto_{p'} id' \in H, id \mapsto v, ref \mapsto_p id} p = 1$.

Since there is only one reference in the term then we have $ref' = ref$ and $id' = id$ and $p' = p$. However in the output heap we now have $ref_1 \mapsto_{\frac{p}{2}} id$ and $ref_2 \mapsto_{\frac{p}{2}} id$ in the output heap and thus:
$$\sum_{ref \mapsto_p id \in H'} p = \frac{p}{2} + \frac{p}{2} = p = 1$$
Satisfying the goal.

3. t has the form $(*(v, w))$ with reduction:
$$\frac{\begin{array}{c} H \vdash \textbf{split}\ (*v) \rightsquigarrow_s H' \vdash (*v_1, *v_2) \\ H' \vdash \textbf{split}\ (*w) \rightsquigarrow_s H'' \vdash (*w_1, *w_2) \end{array}}{H \vdash \textbf{split}\ (*(v, w)) \rightsquigarrow_s H'' \vdash (*(v_1, w_1), *(v_2, w_2))} \rightsquigarrow_{\text{SPLIT}\otimes}$$
Induction over the two premises gives us the result here, by threading the theorem's implications from H to H' via the first premise and then H' to H'' via the second.

- (join)
$$\frac{\Gamma_1 \vdash t_1 : \&_p A \qquad \Gamma_2 \vdash t_2 : \&_q A \qquad p + q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \textbf{join}\ t_1\ t_2 : \&_{p+q} A}\ \text{JOIN}$$
with a heap H such that $H \bowtie (\Gamma_1 + \Gamma_2)$ and reduction:
$$H \vdash \textbf{join}\ t_1\ t_2 \rightsquigarrow_s H' \vdash t''$$
which has four possible derivations:

1.
$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_1'}{H \vdash \mathbf{join}\ t_1\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ t_1'\ t_2}\ \rightsquigarrow_{\text{JOINL}}$$
Here, induction over t_1 provides the goal, following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\text{APPL}}$ case.

2. $t_1 = v$ for some value v, and we can reduce:
$$\frac{H \vdash t_2 \rightsquigarrow_s H' \vdash t_2'}{H \vdash \mathbf{join}\ v\ t_2 \rightsquigarrow_s H' \vdash \mathbf{join}\ v\ t_2'}\ \rightsquigarrow_{\text{JOINR}}$$
Here, induction over t_2 provides the goal, following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\text{APPL}}$ case.

3. As above, but t_2 has the form $(*ref_2)$, which restricts v to have the form $(*ref_1)$ such that the typing is:
$$\frac{\begin{array}{c}[\Gamma], ref_1 : \mathsf{Ref}_{id}\ A \vdash (*ref_1) : \&_p(\mathsf{Ref}_{id}\ A)\ \textsc{ref+nec} \\ [\Gamma], ref_2 : \mathsf{Ref}_{id}\ A \vdash (*ref_2) : \&_q(\mathsf{Ref}_{id}\ A)\ \textsc{ref+nec}\end{array}}{[\Gamma], ref_1 : \mathsf{Ref}_{id}\ A, ref_2 : \mathsf{Ref}_{id}\ A \vdash \mathbf{join}\ (*ref_1)\ (*ref_2) : \&_{p+q}(\mathsf{Ref}_{id}\ A)}\ \textsc{join}$$
and we have reduction:
$$\frac{ref \# H}{H, ref_1 \mapsto_p id, ref_2 \mapsto_q id, id \mapsto v \vdash \mathbf{join}\ (*ref_1)\ (*ref_2) \rightsquigarrow_s H, ref \mapsto_{(p+q)} id, id \mapsto v \vdash *ref}\ \rightsquigarrow_{\text{JOINREF}}$$
Here, for all $id \in \mathsf{dom}(H, id \mapsto v, ref_1 \mapsto_p id, ref_2 \mapsto_q id)$ and all $ref' \in \mathsf{refs}(\mathbf{join}\ (*ref_1)\ (*ref_2))$ then we assume the antecedent $\sum_{ref' \mapsto_p id \in H} p = 1$ then we have:
$$\sum_{\substack{\forall ref \in \mathsf{refs}(t'). \\ ref \mapsto_{p'} id \in H'}} = p + q = 1$$
Part 2 then follows trivially as there are no new resources created.

4. t_2 has the form $(*(v_2, w_2))$, which restricts t_1 to have the form $(*(v_1, w_1))$, allowing the reduction:
$$\frac{\begin{array}{c}H \vdash \mathbf{join}\ (*v_1)\ (*v_2) \rightsquigarrow_s H' \vdash *v \\ H' \vdash \mathbf{join}\ (*w_1)\ (*w_2) \rightsquigarrow_s H'' \vdash *w\end{array}}{H \vdash \mathbf{join}\ (*(v_1, w_1))\ (*(v_2, w_2)) \rightsquigarrow_s H'' \vdash *(v, w)}\ \rightsquigarrow_{\text{JOIN}\otimes}$$
Induction over the two premises gives us the result here, by threading the theorem's implications from H to H' via the first premise and then H' to H'' via the second.

- (push)

$$\frac{\Gamma \vdash t : \&_p(A \otimes B)}{\Gamma \vdash \mathbf{push}\, t : (\&_p A) \otimes (\&_p B)}\ \textsc{push}$$

with a heap H such that $H \bowtie \Gamma$ and reduction:

$$H \vdash \mathbf{push}\, t \ \rightsquigarrow_s\ H' \vdash t''$$

which has three possible derivations:

1.
$$\frac{H \vdash t \ \rightsquigarrow_s\ H' \vdash t'}{H \vdash \mathbf{push}\, t \ \rightsquigarrow_s\ H' \vdash \mathbf{push}\, t'}\ \rightsquigarrow_{\textsc{push}}$$
Here, induction provides the goal, following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\textsc{appl}}$ case.

2. t has the form $*(v_1, v_2)$ and we can reduce:
$$\frac{}{H \vdash \mathbf{push}\,(*(v_1, v_2)) \ \rightsquigarrow_s\ H \vdash (*v_1, *v_2)}\ \rightsquigarrow_{\textsc{push}*}$$
Here, the goal is trivial since the heap H is preserved.

- (pull)

$$\frac{\Gamma \vdash t : (\&_p A) \otimes (\&_p B)}{\Gamma \vdash \mathbf{pull}\, t : \&_p(A \otimes B)}\ \textsc{pull}$$

with a heap H such that $H \bowtie \Gamma$ and reduction:

$$H \vdash \mathbf{pull}\, t \ \rightsquigarrow_s\ H' \vdash t''$$

which has three possible derivations:

1.
$$\frac{H \vdash t \ \rightsquigarrow_s\ H' \vdash t'}{H \vdash \mathbf{pull}\, t \ \rightsquigarrow_s\ H' \vdash \mathbf{pull}\, t'}\ \rightsquigarrow_{\textsc{pull}}$$
Here, induction provides the goal, following the same scheme of generalising the inductive evidence as for the $\rightsquigarrow_{\textsc{appl}}$ case.

2. t has the form $(*v_1, *v_2)$ and we can reduce:
$$\frac{}{H \vdash \mathbf{pull}\,(*v_1, *v_2) \ \rightsquigarrow_s\ H \vdash *(v_1, v_2)}\ \rightsquigarrow_{\textsc{pull}*}$$
Here, the goal is trivial since the heap H is preserved.

- (newArray) (readArray) (writeArray) (deleteArray) (newRef) (swapRef) (freezeRef) (readRef)

All trivial as they have no reductions.

- (pack)

$$\frac{\Gamma \vdash t : A \qquad id \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathbf{pack}\ \langle id', t\rangle : \exists id.A[id/id']}\ \textsc{pack}$$

with a heap H such that $H \bowtie \Gamma$ and reduction:

$$H \vdash \mathbf{pack}\ \langle id, t\rangle \rightsquigarrow_s H' \vdash t''$$

which has one possible derivation:

$$\frac{H \vdash t \rightsquigarrow_s H \vdash t'}{H \vdash \mathbf{pack}\ \langle id, t\rangle \rightsquigarrow_s H \vdash \mathbf{pack}\ \langle id, t'\rangle}\ \rightsquigarrow_{\textsc{pack}}$$

Here, the goal is trivial since the heap H is preserved.

- (unpack)

$$\frac{\begin{array}{l}\Gamma_1 \vdash t_1 : \exists id.A \\ \Gamma_2, id, x : A \vdash t_2 : B \qquad id \notin \mathsf{fv}(B)\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ \langle id, x\rangle = t_1\ \mathbf{in}\ t_2 : B}\ \textsc{unpack}$$

with a heap H such that $H \bowtie \Gamma$ and reduction:

$$H \vdash \mathbf{unpack}\ \langle id, x\rangle = t_1\ \mathbf{in}\ t_2 \rightsquigarrow_s H' \vdash t''$$

which has two possible derivations:

1.

$$\frac{y\#\{H, v, t\}}{H \vdash \mathbf{unpack}\ \langle id, x\rangle = \mathbf{pack}\ \langle id', v\rangle\ \mathbf{in}\ t \rightsquigarrow_s H, y \mapsto_r v \vdash t[y/x]}\ \rightsquigarrow_{\exists\beta}$$

Here, the heap is preserved with the exception of y. Since no references are affected, the goal is achieved directly.

2.

$$\frac{H \vdash t_1 \rightsquigarrow_s H \vdash t_1'}{H \vdash \mathbf{unpack}\ \langle id, x\rangle = t_1\ \mathbf{in}\ t_2 \rightsquigarrow_s H \vdash \mathbf{unpack}\ \langle id, x\rangle = t_1'\ \mathbf{in}\ t_2}\ \rightsquigarrow_{\textsc{unpack}}$$

Here, the goal is trivial since the heap H is preserved.

□

Theorem 9 (Multi-reduction borrow safety)**.** *For a well-typed term $\Gamma \vdash t : A$ and all Γ_0, s, and H such that $H \bowtie (\Gamma_0 + s \cdot \Gamma)$, and multi-step reduction $H \vdash t \Rightarrow_s H' \vdash v$, then for all $id \in \mathsf{dom}(H)$:*

$$\sum_{\substack{\forall ref \in \mathsf{refs}(t).\\ ref \mapsto_p id \in H}} p = 1 \implies \exists! ref'.ref' \mapsto_1 id \in H'$$

i.e., for all resources with identifier id in the incoming heap and all references in the term pointing to this resource, if the sum of all permissions pointing to this resource are 1 *in the incoming heap then their total permission of* 1 *is preserved from the incoming heap to the resulting term, with this permission now contained in a single reference ref′.*

Furthermore, any new references in the final term should uniquely point to an identifier, and thus have permission 1*. That is, for all* $id' \in \mathsf{dom}(H') \wedge id' \notin \mathsf{dom}(H)$ *then:*

$$\forall ref \in \mathsf{refs}(v).\ \exists! ref'.\ ref' \mapsto_1 id' \in H'$$

Proof. By induction on the structure of the multi-reduction $H \vdash t_1 \Rightarrow_s H' \vdash v$.

- (refl)

$$\frac{}{H \vdash v \Rightarrow_s H \vdash v}\ \textsc{refl}$$

Since v is a value which we know has type $*A$, then by the unique value lemma there are two possibilities for the form of v.

1. $A = Res_{id}\ A'$, and so v has the form $*ref$. This restricts the typing as follows:

$$\frac{}{0 \cdot \Gamma, ref : Res_{id}\ A \vdash *ref : *(Res_{id}\ A)}\ *\textsc{ref}^*$$

Then we also have a heap H such that $H \bowtie (0 \cdot \Gamma, ref : Res_{id}\ A)$ which by inversion of heap-compatibility for references implies that there exists a subheap H_1 such that $H = H_1, ref \mapsto_1 id, id \mapsto v$.

As we have a single reference in the heap annotated with 1, both the premise and the goal of the implication in the theorem hold.

2. $A = A' \otimes B$, and so v has the form (v_1, v_2). This restricts the typing to two possible derivations:

$$\frac{\dfrac{\gamma_1 \vdash v_1 : A' \quad \gamma_2 \vdash v_2 : B}{\gamma_1 + \gamma_2 \vdash (v_1, v_2) : A' \otimes B}\otimes_I}{\gamma_1 + \gamma_2 \vdash *(v_1, v_2) : *(A' \otimes B)}\ \textsc{nec} \qquad \frac{\dfrac{\dfrac{\gamma_1 \vdash v_1 : A'}{\gamma_1 \vdash *v_1 : *A'}\ \textsc{nec} \quad \dfrac{\gamma_2 \vdash v_2 : B}{\gamma_2 \vdash *v_2 : *B}\ \textsc{nec}}{\gamma_1 + \gamma_2 \vdash (*v_1, *v_2) : *A' \otimes *B}\otimes_I}{\gamma_1 + \gamma_2 \vdash *(v_1, v_2) : *(A' \otimes B)}\ \textsc{pull}$$

In either case, by the unique value lemma we know that v_1 and v_2 are both restricted: they can either be of the form a, meaning that one of the types in the product is $Res_{id}\ A$, or they can be of the form (v_3, v_4), meaning that one of the types is $A'' \otimes B'$.

Proceed by inspecting each of v_1 and v_2 in turn. If the value is of the form ref, then as above by inversion of heap

compatibility the heap must contain a unique reference $ref \mapsto_1 id, id \mapsto v$, which satisfies the theorem as it is annotated with 1.

If the value is of the form (v_3, v_4), then we can restrict the typing of this subpart of the derivation by exactly the above argument, and then inspect the form of v_3 and v_4 using the unique value lemma following the same logic. This proceeds inductively; as typing derivations are finite trees, this must eventually terminate in an reference which satisfies the theorem at every leaf of the tree.

- (ext)

$$\frac{H \vdash t_1 \rightsquigarrow_s H' \vdash t_2 \qquad H' \vdash t_2 \Rightarrow_s H'' \vdash t_3}{H \vdash t_1 \Rightarrow_s H'' \vdash t_3}\ \textsc{ext}$$

First, consider the first premise, which is a single-step reduction of the form $H \vdash t_1 \rightsquigarrow_s H' \vdash t_2$.

From Theorem 16, we know that for all Γ_0 and heaps H such that $H \bowtie (\Gamma_0 + s \cdot \Gamma)$ then for all $id \in \mathsf{dom}(H)$:

$$\sum_{\substack{\forall ref \in \mathsf{refs}(t). \\ ref \mapsto_p id \in H}} p = 1 \implies \sum_{\substack{\forall ref \in \mathsf{refs}(t). \\ ref \mapsto_p id \in H'}} p \in \{0, 1\}$$

and also that for all $id' \in \mathsf{dom}(H') \wedge id' \notin \mathsf{dom}(H)$ (new resources) we have:

$$\sum_{\substack{\forall ref' \in \mathsf{refs}(t'). \\ ref' \mapsto_q id' \in H'}} q = 1$$

Now, we induct over the second premise, which is a multi-reduction of the form $H' \vdash t_2 \Rightarrow_s H'' \vdash t_3$. Induction using the present theorem tells us that for all Γ_0 and H such that $H \bowtie (\Gamma_0 + s \cdot \Gamma)$, then for all $id \in \mathsf{dom}(H)$ we have:

$$\sum_{\substack{\forall ref \in \mathsf{refs}(t). \\ ref \mapsto_p id \in H'}} p = 1 \implies \exists! ref'. ref' \mapsto_1 id \in H''$$

From our knowledge about the first premise (the single-step reduction), there are two cases.

- $\sum_{\substack{\forall ref \in \mathsf{refs}(t). \\ ref \mapsto_p id \in H}} p = \sum_{\substack{\forall ref \in \mathsf{refs}(t). \\ ref \mapsto_p id \in H'}} p = 1.$

Then we know that fractions must sum to 1 for both references preserved from t_1 and also new references preserved from t_2, but

via the implication we obtained from induction on the second premise, we know $\exists! ref'.ref' \mapsto_1 id \in H''$, which is exactly the required result.

- $\sum_{\substack{\forall ref \in \mathsf{refs}(t). \\ ref \mapsto_p id \in H'}} p = 0.$

  Then we only need to concern ourselves with new references preserved from t_2, but via the same implication we obtained via induction on the second premise, we know $\exists! ref'.ref' \mapsto_1 id \in H''$, again exactly as required.

□

Corollary 2 (Uniqueness). *For a well-typed term $\Gamma \vdash t : *A$ and all Γ_0, s, and H such that $H \bowtie (\Gamma_0 + s \cdot \Gamma)$ and multi-reduction to a value $H \vdash t \Rightarrow_s H' \vdash *v$, for all $id \in \mathsf{dom}(H)$ then:*

$$\forall ref \in \mathsf{refs}(t).(ref \mapsto_1 id \in H \implies ref \mapsto_1 id \in H')$$
$$\wedge \quad \forall id' \in \mathsf{dom}(H') \wedge id' \notin \mathsf{dom}(H).\ \forall ref \in \mathsf{refs}(v).\ \exists! ref'.ref \mapsto_1 id' \in H'$$

i.e., any references contributing to the final term that are unique in the incoming heap stay unique in the resulting term, and any new references contributing to the final term are also unique.

Proof. Follows directly from Lemma 9, in the subcase where only one reference exists in the initial heap (since $\sum_{ref \mapsto_p id \in H} p = 1$ must hold if there exists a single reference such that $ref \mapsto_1 id \in H$). □

C.5 SOUNDNESS OF HEAP MODEL WRT. EQUATIONAL THEORY

Theorem 10 (Soundness with respect to the equational theory). *For all t_1, t_2 such that $\Gamma \vdash t_1 : A$ and $\Gamma \vdash t_2 : A$ and $t_1 \equiv t_2$ and given H such that $H \bowtie \Gamma$, there exist multi-reductions to values that are equal under full β-reduction and evaluating any references to the value they point to in the resulting heaps (written $H'(v_1)$ and $H'(v_2)$):*

$$H \vdash t_1 \Rightarrow_1 H' \vdash v_1 \quad \wedge \quad H \vdash t_2 \Rightarrow_1 H'' \vdash v_2 \quad \wedge \quad H'(v_1) \equiv H''(v_2)$$

Proof. • (β_*)

$$\frac{}{\mathbf{clone}\,(\mathbf{share}\,v)\,\mathbf{as}\,x\,\mathbf{in}\,t' \equiv t'[\mathbf{pack}\,\langle \overline{id}, v\rangle / x]}\ \beta_*$$

With typing derivation for the LHS:

$$\frac{\dfrac{\Gamma_1, \overline{id} \vdash v : *A}{\Gamma_1, \overline{id} \vdash \mathbf{share}\,v : \square_r A}\ \textsc{share} \qquad \mathsf{noIDs}(\Gamma_1) \quad \Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id}]) \vdash t : \square_r B \quad 1 \sqsubseteq r}{(\Gamma_1 + \Gamma_2), \overline{id} \vdash \mathbf{clone}\,(\mathbf{share}\,v)\,\mathbf{as}\,x\,\mathbf{in}\,t : \square_r B}\ \textsc{clone}$$

And typing derivation for the RHS:

$$(\Gamma_1 + \Gamma_2), \overline{id} \vdash t[\mathbf{pack}\ \langle \overline{id}, v\rangle / x] : \Box_r B$$

By the value lemma (Lemma 10) we know that $v = *v'$ therefore we know that we can perform the following reduction (where we elide the output binding context and usage context as they are not needed in the proof):

$$\begin{array}{rcl} & & H_1, H_2 \vdash \mathbf{clone}\ (\mathbf{share}\ (*v'))\ \mathbf{as}\ x\ \mathbf{in}\ t \\ \rightsquigarrow_{\text{CLONE}} + \rightsquigarrow_{\text{SHARE}\beta} & \to & [H_1]_0, H_2 \vdash \mathbf{clone}\ [v']\ \mathbf{as}\ x\ \mathbf{in}\ t \\ \rightsquigarrow_{\text{CLONE}\beta} & \rightsquigarrow & [H_1]_0, H_2, H_3, x \mapsto_r \mathbf{pack}\ \langle \overline{id}, \theta(v)\rangle \vdash t \end{array}$$

where $\mathsf{dom}(H_1) \equiv \mathsf{refs}(v)$ and $(H_3, \theta, \overline{id}) = \mathsf{copy}(H_1)$

- (∗assoc)

$$\frac{x \# t_3}{\mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ (\mathbf{clone}\ t_2\ \mathbf{as}\ y\ \mathbf{in}\ t_3) \equiv \mathbf{clone}\ (\mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2)\ \mathbf{as}\ y\ \mathbf{in}\ t_3}\ *\text{ASSOC}$$

With typing for the LHS:

$$\dfrac{\Gamma_1, \overline{id_1} \vdash t_1 : \Box_{r_1} A_1 \quad \dfrac{\Gamma_2, \overline{id_2}, x : \exists \overline{id_1'}.*(A_1[\overline{id_1'}/\overline{id_1}]) \vdash t_2 : \Box_{r_2} A_2 \quad \Gamma_3, y : \exists \overline{id_2'}.*(A_2[\overline{id_2'}/\overline{id_2}]) \vdash t_3 : \Box_{r_3} A_3}{\Gamma_2, x : \exists \overline{id_1'}.*(A_1[\overline{id_1'}/\overline{id_1}]) + \Gamma_3 \vdash \mathbf{clone}\ t_2\ \mathbf{as}\ y\ \mathbf{in}\ t_3 : \Box_{r_3} A_3}\ \text{CLONE}}{(\Gamma_1 + \Gamma_2 + \Gamma_3), \overline{id_1}, \overline{id_2} \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ (\mathbf{clone}\ t_2\ \mathbf{as}\ y\ \mathbf{in}\ t_3) : \Box_{r_3} A_3}\ \text{CLONE}$$

where $\mathsf{noIDs}(\Gamma_1)$ and $\mathsf{noIDs}(\Gamma_2)$ and $1 \sqsubseteq r_1$ and $1 \sqsubseteq r_2$.

And with RHS typing, with the same conditions:

$$\dfrac{\dfrac{\Gamma_1, \overline{id_1} \vdash t_1 : \Box_{r_1} A_1 \quad \Gamma_2, \overline{id_2}, x : \exists \overline{id_1'}.*(A_1[\overline{id_1'}/\overline{id_1}]) \vdash t_2 : \Box_{r_2} A_2}{(\Gamma_1 + \Gamma_2), \overline{id_1}, \overline{id_2} \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2 : \Box_{r_2} A_2}\ \text{CLONE} \quad \Gamma_3, y : \exists \overline{id_2'}.*(A_2[\overline{id_2'}/\overline{id_2}]) \vdash t_3 : \Box_{r_3} A_3}{(\Gamma_1 + \Gamma_2 + \Gamma_3), \overline{id_1}, \overline{id_2} \vdash \mathbf{clone}\ (\mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2)\ \mathbf{as}\ y\ \mathbf{in}\ t_3 : \Box_{r_3} A_3}\ \text{CLONE}$$

There are four possibilities depending on the reduction of t_1 and t_2.

1. Divergence: $t_1 \to^\omega$ (i.e., t_1 diverges). In which case then $H \vdash \mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ (\mathbf{clone}\ t_2\ \mathbf{as}\ y\ \mathbf{in}\ t_3) \rightsquigarrow_{\text{CLONE}}{}^\omega$ (diverging) and also $H \vdash \mathbf{clone}\ (\mathbf{clone}\ t_1\ \mathbf{as}\ x\ \mathbf{in}\ t_2)\ \mathbf{as}\ y\ \mathbf{in}\ t_3 \rightsquigarrow_{\text{CLONE}}{}^\omega$ and so the the equation is trivially satisfied as both sides diverge.

   If t_1 converges, but t_2 diverges then both sides diverge by similar reasoning. Similarly if both diverge then overall both sides diverge.

2. Convergence: t_1 reduces to a value v_1 and t_2 reduces to a value v_2. By the typing and the value lemma (Lemma 10) then $\exists v_1'.\ v_1 = [v_1']$ and then $\exists v_2'.\ v_2 = [v_2']$.

Then we can reduce as follows on the LHS:

$$
\begin{array}{rll}
 & & H \vdash \mathbf{clone}\, t_1\, \mathbf{as}\, x\, \mathbf{in}\, (\mathbf{clone}\, t_2\, \mathbf{as}\, y\, \mathbf{in}\, t_3) \\
 & \rightsquigarrow * & H' \vdash \mathbf{clone}\, [v_1']\, \mathbf{as}\, x\, \mathbf{in}\, (\mathbf{clone}\, t_2\, \mathbf{as}\, y\, \mathbf{in}\, t_3) \\
\rightsquigarrow_{\text{CLONE}\beta} & \rightsquigarrow & H_1, H_1', H_1'', x \mapsto_r \mathbf{pack}\, \langle \overline{id_1}, *\theta(v_1) \rangle \vdash \mathbf{clone}\, t_2\, \mathbf{as}\, y\, \mathbf{in}\, t_3 \\
 & & \mathsf{dom}(H_1') \equiv \mathsf{refs}(v_1) \ \wedge\ (H_1'', \theta, \overline{id_1}) = \mathsf{copy}(H_1') \\
 & \rightsquigarrow * & H'' \vdash \mathbf{clone}\, [v_2]\, \mathbf{as}\, y\, \mathbf{in}\, t_3 \\
\rightsquigarrow_{\text{CLONE}\beta} & \rightsquigarrow & H_2, H_2', H_2'', y \mapsto_r \mathbf{pack}\, \langle \overline{id_2}, *\theta'(v_2) \rangle \vdash t_3 \\
 & & \mathsf{dom}(H_2') \equiv \mathsf{refs}(v_2) \ \wedge\ (H_2'', \theta, \overline{id_2}) = \mathsf{copy}(H_2')
\end{array}
$$

and on the RHS:

$$
\begin{array}{rl}
 & H \vdash \mathbf{clone}\, (\mathbf{clone}\, t_1\, \mathbf{as}\, x\, \mathbf{in}\, t_2)\, \mathbf{as}\, y\, \mathbf{in}\, t_3 \\
\rightsquigarrow * & H' \vdash \mathbf{clone}\, (\mathbf{clone}\, [v_1]\, \mathbf{as}\, x\, \mathbf{in}\, t_2)\, \mathbf{as}\, y\, \mathbf{in}\, t_3 \\
\rightsquigarrow & H_1, H_1', H_1'', x \mapsto_r \mathbf{pack}\, \langle \overline{id_1}, *\theta(v_1) \rangle \vdash \mathbf{clone}\, t_2\, \mathbf{as}\, y\, \mathbf{in}\, t_3 \\
 & \mathsf{dom}(H_1') \equiv \mathsf{refs}(v_1) \ \wedge\ (H_1'', \theta, \overline{id_1'}) = \mathsf{copy}(H_1') \\
\rightsquigarrow * & H'' \vdash \mathbf{clone}\, [v_2]\, \mathbf{as}\, y\, \mathbf{in}\, t_3 \\
\rightsquigarrow_{\text{CLONE}\beta} & H_2, H_2', H_2'', y \mapsto_r \mathbf{pack}\, \langle \overline{id_2}, *\theta'(v_2) \rangle \vdash t_3 \\
 & \mathsf{dom}(H_2') \equiv \mathsf{refs}(v_2) \ \wedge\ (H_2'', \theta, \overline{id_2'}) = \mathsf{copy}(H_2')
\end{array}
$$

matching in both sides.

- (&unit)

$$\frac{}{\mathbf{withBorrow}\ (\lambda x.x)\ t \equiv t}\ \text{\&UNIT}$$

With typing derivation for the LHS:

$$\frac{\Gamma_1 \vdash t : *A \qquad \dfrac{\dfrac{}{0 \cdot \Gamma_2, x : \&_1 A \vdash x : \&_1 A}\ \text{VAR}}{0 \cdot \Gamma_2 \vdash (\lambda x.x) : \&_1 A \multimap \&_1 A}\ \text{ABS}}{\Gamma_1 + 0 \cdot \Gamma_2 \vdash \mathbf{withBorrow}\ (\lambda x.x)\ t : *A}\ \text{WITHBORROW}$$

and typing derivation for the RHS:

$$\Gamma_1 + 0 \cdot \Gamma_2 \vdash t : *A$$

There are two possibilities depending on the reduction of t.

1. If reduction of t diverges, then reduction on the LHS also diverges, since we repeatedly apply the $\rightsquigarrow_{\text{WITH\&R}}$ rule. Hence, the equation is trivially satisfied as both sides diverge.
2. Otherwise, t reduces to a value v. By the typing and the value lemma (Lemma 10) then $\exists v'.\ v = *v'$.

Then we can reduce as follows on the LHS:

$$\begin{array}{rll} & & H \vdash \textbf{withBorrow}\ (\lambda x.x)\ t \\ & \rightsquigarrow * & H' \vdash \textbf{withBorrow}\ (\lambda x.x)\ (*v') \\ \rightsquigarrow_{\text{WITH\&}} & \rightsquigarrow & H', y \mapsto_r * v' \vdash \textbf{unborrow}\ (x[y/x]) \\ & \rightsquigarrow * & H', y \mapsto_r * v' \vdash \textbf{unborrow}\ y \\ \rightsquigarrow_{\text{UN\&}} & \rightsquigarrow & H', y \mapsto_r * v' \vdash y \\ & \rightsquigarrow * & H', y \mapsto_r * v' \vdash *v' \end{array}$$

and as follows on the RHS:

$$\begin{array}{rll} & & H \vdash t \\ & \rightsquigarrow * & H' \vdash *v' \end{array}$$

- (&assoc)

$$\frac{}{\textbf{withBorrow}\ (\lambda x.(f\ (g\ x)))\ t \equiv \textbf{withBorrow}\ f\ (\textbf{withBorrow}\ g\ t)}\ \&\text{ASSOC}$$

If any of the terms f, g or t diverge, then the LHS and RHS both diverge, and so the equation is trivially satisfied.

Otherwise, all three terms must reduce to values v_1, v_2 and v_3.

By typing and the value lemma (Lemma 10) then $v_1 = \lambda x_1.t_1$, $v_2 = \lambda x_2.t_2$, and $\exists v_3'.\ v_3 = *v_3'$.

Then we can reduce as follows on the LHS:

$$\begin{array}{rll} & & H \vdash \textbf{withBorrow}\ (\lambda x.(f\ (g\ x)))\ t \\ & \rightsquigarrow * & H' \vdash \textbf{withBorrow}\ (\lambda x.(f\ (g\ x)))\ (*v_3') \\ \rightsquigarrow_{\text{WITH\&}} & \rightsquigarrow & H', y \mapsto_r * v_3' \vdash \textbf{unborrow}\ ((f\ (g\ x))[y/x]) \\ & \rightsquigarrow * & H', y \mapsto_r * v_3' \vdash \textbf{unborrow}\ (((\lambda x_1.t_1)\ ((\lambda x_2.t_2)\ y))) \\ \rightsquigarrow_{\beta} & \rightsquigarrow & H', y \mapsto_r * v_3', x_2 \mapsto_s (*v_3') \vdash \textbf{unborrow}\ (((\lambda x_1.t_1)\ t_2)) \end{array}$$

If t_2 diverges, then again both sides diverge. Otherwise, t_2 must reduce to a value v_4, and by the value lemma $\exists v_4'.\ v_4 = (*v_4')$. Then we can continue reducing as follows:

$$\begin{array}{rll} & \rightsquigarrow * & H'', y \mapsto_r * v_3', x_2 \mapsto_s (*v_3') \vdash \textbf{unborrow}\ (((\lambda x_1.t_1)\ (*v_4'))) \\ \rightsquigarrow_{\beta} & \rightsquigarrow & H'', y \mapsto_r * v_3', x_2 \mapsto_s (*v_3'), x_1 \mapsto_{s'} (*v_4') \vdash \textbf{unborrow}\ (t_1) \end{array}$$

If t_1 diverges, then again both sides diverge. Otherwise, t_1 must reduce to a value v_5, and by the value lemma $\exists v_5'.\ v_5 = (*v_5')$. Then we can continue reducing as follows:

$$\begin{array}{rll} & \rightsquigarrow * & H''', y \mapsto_r * v_3', x_2 \mapsto_s (*v_3'), x_1 \mapsto_{s'} ((*v_4')) \vdash \textbf{unborrow}\ ((*v_5')) \\ \rightsquigarrow_{\text{UN\&}} & \rightsquigarrow & H''', y \mapsto_r * v_3', x_2 \mapsto_s (*v_3'), x_1 \mapsto_{s'} (*v_4') \vdash *v_5' \end{array}$$

and on the RHS:

$$\begin{array}{rcl} & & H \vdash \textbf{withBorrow}\ f\ (\textbf{withBorrow}\ g\ t) \\ & \rightsquigarrow * & H' \vdash \textbf{withBorrow}\ (\lambda x_1.t_1)\ (\textbf{withBorrow}\ (\lambda x_2.t_2)\ (*v_3')) \\ \rightsquigarrow_{\text{WITH\&}} & \rightsquigarrow & H', y \mapsto_r * v_3' \vdash \textbf{withBorrow}\ (\lambda x_1.t_1)\ (\textbf{unborrow}\ (t_2[y/x_2])) \\ & \sim & H', y \mapsto_r * v_3', x_2 \mapsto_s (*v_3') \vdash \textbf{withBorrow}\ (\lambda x_1.t_1)\ (\textbf{unborrow}\ (t_2)) \end{array}$$

If t_2 diverges, then again both sides diverge. Otherwise, t_2 must reduce to a value v_4, and by the value lemma $\exists v_4'.\ v_4 = (*v_4')$. Then we can continue reducing as follows:

$$\begin{array}{rcl} & \rightsquigarrow * & H'', y \mapsto_r * v_3', x_2 \mapsto_s (*v_3') \vdash \textbf{withBorrow}\ (\lambda x_1.t_1)\ (\textbf{unborrow}\ ((*v_4'))) \\ \rightsquigarrow_{\text{UN\&}} & \rightsquigarrow & H'', y \mapsto_r * v_3, x_2 \mapsto_s (*v_3') \vdash \textbf{withBorrow}\ (\lambda x_1.t_1)\ (*v_4') \\ \rightsquigarrow_{\text{WITH\&}} & \rightsquigarrow & H'', y \mapsto_r * v_3', x_2 \mapsto_s (*v_3'), z \mapsto_{r'} * v_4' \vdash \textbf{unborrow}\ (t_1[z/x_1]) \\ & \sim & H'', y \mapsto_r * v_3', x_2 \mapsto_s (*v_3'), z \mapsto_{r'} * v_4', x_1 \mapsto_{s'} ((*v_4')) \vdash \textbf{unborrow}\ (t_1) \end{array}$$

If t_1 diverges, then again both sides diverge. Otherwise, t_1 must reduce to a value v_5, and by the value lemma $\exists v_5'.\ v_5 = (*v_5')$. Then we can continue reducing as follows:

$$\begin{array}{rcl} & \rightsquigarrow * & H'', y \mapsto_r * v_3', x_2 \mapsto_s (*v_3'), z \mapsto_{r'} * v_4', x_1 \mapsto_{s'} ((*v_4')) \vdash \textbf{unborrow}\ (*v_5') \\ \rightsquigarrow_{\text{UN\&}} & \rightsquigarrow & H'', y \mapsto_r * v_3', x_2 \mapsto_s (*v_3'), z \mapsto_{r'} * v_4', x_1 \mapsto_{s'} ((*v_4')) \vdash *v_5' \end{array}$$

- $(\textbf{let}\,(x, y) = (\textbf{split}\ t)\ \textbf{in}\ (\textbf{join}\ x\ y)) \equiv t$

  If t diverges, then both sides diverge.

  Otherwise t reduces to a value v which by (Lemma 10) is over the form $*ref$.

  Thus, we reduce by:

$$\begin{array}{rcl} & & H \vdash \textbf{let}\,(x, y) = \textbf{split}\ t\ \textbf{in}\ (\textbf{join}\ x\ y) \\ & \rightsquigarrow * & H', id \mapsto v, ref \mapsto_p id \vdash \textbf{let}\,(x, y) = \textbf{split}\ (*ref)\ \textbf{in}\ (\textbf{join}\ x\ y) \\ \rightsquigarrow_{\text{LET}\otimes} + \rightsquigarrow_{\text{SPLITREF}} & \rightsquigarrow^* & H', id \mapsto v, ref_1 \mapsto_{\frac{p}{2}} id, ref_2 \mapsto_{\frac{p}{2}} id \vdash \textbf{let}\,(x, y) = (*ref_1, *ref_2)\ \textbf{in}\ (\textbf{join}\ x\ y) \\ \rightsquigarrow_{\otimes\beta} & \rightsquigarrow & H', id \mapsto v, ref_1 \mapsto_{\frac{p}{2}} id, ref_2 \mapsto_{\frac{p}{2}} id, x' \mapsto_1 * ref_1, y' \mapsto_1 * ref_2 \vdash \textbf{join}\ x'\ y' \\ (\rightsquigarrow_{\text{PRIM}} + \rightsquigarrow_{\text{VAR}})^{*2} & \rightsquigarrow^* & H', id \mapsto v, ref_1 \mapsto_{\frac{p}{2}} id, ref_2 \mapsto_{\frac{p}{2}} id, x' \mapsto_1 * ref_1, y' \mapsto_1 * ref_2 \vdash \textbf{join}\ (*ref_1')\ (*ref_2') \\ \rightsquigarrow_{\text{JOINREF}} & \rightsquigarrow^* & H', id \mapsto v, ref' \mapsto_p id, x' \mapsto_1 * ref_1, y' \mapsto_1 * ref_2 \vdash *ref' \end{array}$$

  Evaluating under the heap $(H', id \mapsto v, ref' \mapsto_p id, x' \mapsto_1 * ref_1, y' \mapsto_1 * ref_2)(*ref') = v$.

  The RHS then reduces to: $H', id \mapsto v, ref \mapsto_p id \vdash (*ref)$

  Evaluating under the heap $(H', id \mapsto v, ref \mapsto_p id)(ref) = v$, satisfying the goal.

- $\textbf{split}\ (\textbf{join}\ t_1\ t_2) \equiv (t_1, t_2)$

If either t_1 or t_2 diverge then both sides diverge. Otherwise, we assume reduction to values, using the value lemma to ascertain their form:

$$
\begin{array}{rcl}
 & & H \vdash \textbf{split}\ (\textbf{join}\ t_1\ t_2) \\
 & \rightsquigarrow * & H', id \mapsto v, ref_1 \mapsto_p id, ref_2 \mapsto_q id \vdash \textbf{split}\ (\textbf{join}\ (*ref_1)\ (*ref_2)) \\
\rightsquigarrow_{\textsc{joinRef}} & \rightsquigarrow & H', id \mapsto v, ref \mapsto_{(p+q)} id \vdash \textbf{split}\ (*ref) \\
\rightsquigarrow_{\textsc{splitRef}} & \rightsquigarrow & H', id \mapsto v, ref_1' \mapsto_{\frac{(p+q)}{2}} id, ref_2 \mapsto_{\frac{(p+q)}{2}} id \vdash (*ref_1', *ref_2')
\end{array}
$$

Evaluating under the heap $(H', id \mapsto v, ref_1' \mapsto_{\frac{(p+q)}{2}} id, ref_2 \mapsto_{\frac{(p+q)}{2}} id)((*ref_1', *ref_2')) = (v, v)$.

The RHS then reduces to $H', id \mapsto v, ref_1 \mapsto_p id, ref_2 \mapsto_q id \vdash (*ref_1, *ref_2)$.

Evaluating under the heap $(H', id \mapsto v, ref_1 \mapsto_p id, ref_2 \mapsto_q id)((*ref_1, *ref_2)) = (v, v)$, satisfying the goal.

□

d APPENDIX FOR CHAPTERS 6/7

D.1 COLLECTED RULES

D.1.1 Typing

$$\frac{}{x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \multimap_i \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \multimap_e$$

$$\frac{\Gamma_1 \vdash t_1 : A \quad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_i \qquad \frac{\Gamma_1 \vdash t_1 : A \otimes A' \quad \Gamma_2, x : A, y : A' \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2 : B}\ \otimes_e$$

$$\frac{}{\varnothing \vdash \mathbf{unit} : \mathsf{unit}}\ \textsc{unit}_i \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \quad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ \mathbf{unit} = t_1\ \mathbf{in}\ t_2 : B}\ \textsc{unit}_e$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der} \qquad \frac{\neg \mathsf{resourceAllocator}(t) \quad [\Gamma] \vdash t : A}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr} \qquad \frac{\Gamma \vdash t : B}{\Gamma, 0 \cdot \Gamma' \vdash t : B}\ \textsc{weak}$$

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \quad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx}$$

Definition 21 (Graded contexts). $[\Gamma]$ *classifies those contexts which contain only graded variables:*

$$\frac{}{[\varnothing]} \qquad \frac{[\Gamma]}{[\Gamma, x : [A]_r]}$$

Definition 22 (Resource allocating terms). *Predicate definition:*

$$\frac{}{\mathsf{resourceAllocator}(\mathsf{forkLinear})} \qquad \frac{}{\mathsf{resourceAllocator}(\mathsf{forkNonLinear})}$$

$$\frac{}{\mathsf{resourceAllocator}(\mathsf{forkReplicate})} \qquad \frac{}{\mathsf{resourceAllocator}(\mathsf{forkMulticast})}$$

$$\frac{\mathsf{resourceAllocator}(t)}{\mathsf{resourceAllocator}([t])}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(t_1\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}((\lambda x.t_1)\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ [x]\ = t_1\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ [x]\ = t_1\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ \mathbf{unit} = t_1\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ \mathbf{unit} = t_1\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ (x,y) = t_1\ \textbf{\textit{in}}\ t_2)} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}(\textbf{\textit{let}}\ (x,y) = t_1\ \textbf{\textit{in}}\ t_2)}$$

$$\frac{\mathsf{resourceAllocator}(t_1)}{\mathsf{resourceAllocator}((t_1,t_2))} \qquad \frac{\mathsf{resourceAllocator}(t_2)}{\mathsf{resourceAllocator}((t_1,t_2))}$$

PRIMITIVES

$$\frac{}{\emptyset \vdash \mathsf{send} : A \otimes \mathbf{Chan}\,(\mathbf{Send}\ A\ P) \multimap \mathbf{Chan}\,P}\ \textsc{send}$$

$$\frac{}{\emptyset \vdash \mathsf{receive} : \mathbf{Chan}\,(\mathbf{Recv}\ A\ P) \multimap (A \otimes \mathbf{Chan}\,P)}\ \textsc{recv}$$

$$\frac{}{\emptyset \vdash \mathsf{forkLinear} : (\mathbf{Chan}\,P \multimap \mathsf{unit}) \multimap \mathbf{Chan}\,\overline{P}}\ \textsc{fork}$$

$$\frac{}{\emptyset \vdash \mathsf{close} : \mathbf{Chan}\,\mathbf{End} \multimap \mathsf{unit}}\ \textsc{close}$$

$$\frac{\mathsf{SingleAction}(P) \qquad \mathsf{ExactSemiring}(S) \qquad r : S}{\emptyset \vdash \mathsf{forkNonLinear} : (\square_r \mathbf{Chan}\,P \multimap \mathsf{unit}) \multimap \square_r(\mathbf{Chan}\,\overline{P})}\ \textsc{forkNonLin}$$

$$\frac{\mathsf{ReceivePrefix}(P)}{\emptyset \vdash \mathsf{forkReplicate} : \square_{0..n}(\mathbf{Chan}\,P \multimap \mathsf{unit}) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\,n\,(\square_{0..1}(\mathbf{Chan}\,\overline{P}))}\ \textsc{forkRep}$$

$$\frac{\mathsf{Sends}(P)}{\emptyset \vdash \mathsf{forkMulticast} : (\mathbf{Chan}\,(\mathsf{Graded}_n P) \multimap \mathsf{unit}) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})}\ \textsc{forkMulti}$$

RUNTIME TYPING

$$\frac{}{c : \mathbf{Chan}\,P \vdash c : \mathbf{Chan}\,P}\ \textsc{var} \qquad \frac{}{c :^* \mathbf{Chan}\,P \vdash c : \mathbf{Chan}\,P}\ \textsc{varChanS}$$

$$\frac{\Gamma \vdash t : B}{\Gamma, 0 \cdot \Gamma', c :^* \mathbf{Chan}\,P \vdash t : B}\ \textsc{weakChanS}$$

In typing derivations in the appendix, we abstract over whether a channel is linear or shared by writing $c :^{r} \mathbf{Chan}\, P$.

D.1.2 Global configuration reduction rules

$$\frac{t \rightsquigarrow t'}{(\mathcal{P} \mid t^{\mathcal{R}};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid t'^{\mathcal{R}};\ \mathcal{C})}\ \textsc{reduce}$$

$$\frac{(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')}{(\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t;\ \mathcal{C}')}\ \textsc{par}$$

$$\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{send}\, c\, v\};\ \mathcal{C}, \overline{c} = \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{c\};\ \mathcal{C}, \overline{c} = \vec{v} : v)}\ \textsc{send}$$

$$\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{receive}\, c\};\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{(v, c)\};\ \mathcal{C}, c = \vec{v})}\ \textsc{recv}$$

$$\frac{\#c}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkLinear}\, (\lambda x.t)\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^{\circ} \mid \mathit{Ctxt}\{\overline{c}\};\ \mathcal{C}, \overline{c} = \langle\rangle, c = \langle\rangle)}\ \textsc{forkLinear}$$

$$\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{close}\, c\};\ \mathcal{C}, c^{*} = \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{\mathbf{unit}\};\ \mathcal{C}, c^{*} = \vec{v})}\ \textsc{sharedClose}$$

$$\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{close}\, c\};\ \mathcal{C}, c^{1} = \langle\rangle) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{\mathbf{unit}\};\ \mathcal{C})}\ \textsc{close}$$

$$\frac{\#c}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkNonLinear}\, (\lambda x.t)\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{[\overline{c}]\} \mid t[[c]/x]^{\circ};\ \mathcal{C}, \overline{c}^{*} = \langle\rangle, c^{*} = \langle\rangle)}\ \textsc{forkNonLinear}$$

$$\frac{\#c_1 \ldots \#c_n}{\begin{array}{ll} & (\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkReplicate}\, [\lambda x.t]\, n\};\ \mathcal{C}) \\ \rightsquigarrow & (\mathcal{P} \mid \mathit{Ctxt}\{\langle [\overline{c_1}], .., [\overline{c_n}] \rangle\} \mid \mathsf{Wch}\, (c_1 \mapsto \lambda x.t) \mid .. \mid \mathsf{Wch}\, (c_n \mapsto \lambda x.t) \\ & \quad ;\ \mathcal{C}, {c_1}^{*} = \langle\rangle, .., {c_n}^{*} = \langle\rangle, \overline{c_1}^{*} = \langle\rangle, .., \overline{c_n}^{*} = \langle\rangle) \end{array}}\ \textsc{forkReplicate}$$

$$\frac{\#c \qquad \#c_1 \ldots \#c_n}{\begin{array}{ll} & (\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkMulticast}\, (\lambda x.t)\, n\};\ \mathcal{C}) \\ \rightsquigarrow & (\mathcal{P} \mid \mathit{Ctxt}\{\langle \overline{c_1}, .., \overline{c_n} \rangle\} \mid t[c/x]^{\circ} \mid \mathsf{Fwd}\, (c \mapsto c_1, .., c_n) \\ & \quad ;\ \mathcal{C}, c = \langle\rangle, \overline{c} = \langle\rangle, c_1 = \langle\rangle, .., c_n = \langle\rangle, \overline{c_1} = \langle\rangle, .., \overline{c_n} = \langle\rangle) \end{array}}\ \textsc{forkMulticast}$$

D.1.3 Standard small-step reduction rules

$$\frac{}{(\lambda x.t)\, v \rightsquigarrow t[v/x]}\ \rightsquigarrow_{\beta} \qquad \frac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_2}\ \rightsquigarrow_{\text{APP}} \qquad \frac{t \rightsquigarrow t'}{v\, t \rightsquigarrow v\, t'}\ \rightsquigarrow_{\text{prim}}$$

$$\frac{}{\textbf{let}\ (x,y) = (v_1, v_2)\ \textbf{in}\ t_3 \rightsquigarrow t_3[v_1/x][v_2/y]}\ \rightsquigarrow_{\otimes\beta}$$

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let}\ (x,y) = t_1\ \textbf{in}\ t_2 \rightsquigarrow \textbf{let}\ (x,y) = t_1'\ \textbf{in}\ t_2}\ \rightsquigarrow_{\text{LET}\otimes}$$

$$\frac{t_1 \rightsquigarrow t_1'}{(t_1, t_2) \rightsquigarrow (t_1', t_2)}\ \rightsquigarrow_{\otimes\text{reduceL}} \qquad \frac{t_2 \rightsquigarrow t_2'}{(v, t_2) \rightsquigarrow (v, t_2')}\ \rightsquigarrow_{\otimes\text{reduceR}}$$

$$\frac{}{\textbf{let unit} = \textbf{unit}\ \textbf{in}\ t \rightsquigarrow t}\ \rightsquigarrow_{\beta\text{unit}} \qquad \frac{t_1 \rightsquigarrow t_1'}{\textbf{let unit} = t_1\ \textbf{in}\ t_2 \rightsquigarrow \textbf{let unit} = t_1'\ \textbf{in}\ t_2}\ \rightsquigarrow_{\text{LET}\text{unit}}$$

$$\frac{}{\textbf{let}\ [x]\ = [v]\ \textbf{in}\ t \rightsquigarrow t[v/x]}\ \rightsquigarrow_{\Box\beta} \qquad \frac{t_1 \rightsquigarrow t_1'}{\textbf{let}\ [x]\ = t_1\ \textbf{in}\ t_2 \rightsquigarrow \textbf{let}\ [x]\ = t_1'\ \textbf{in}\ t_2}\ \rightsquigarrow_{\text{LET}\Box}$$

$$\frac{t \rightsquigarrow t'}{[t] \rightsquigarrow [t']}\ \rightsquigarrow_{\Box\text{reduce}}$$

D.2 AUXILIARY RESULTS

Linear and graded substitution lemmas have been previously proved by Orchard et al. [103] for the core calculus used here:

Lemma 17 (Linear substitution). *[103] Given* $\Gamma_1, x : A \vdash t_1 : B$ *and* $\Gamma_2 \vdash t_2 : A$ *then* $\Gamma_1 + \Gamma_2 \vdash t_1[t_2/x] : B$.

Lemma 18 (Graded substitution). *[103] Given* $\Gamma_1, x : [A]_r \vdash t_1 : B$ *and* $[\Gamma_2] \vdash t_2 : A$ *and* $\neg\mathsf{resourceAllocator}(t_2)$ *then* $\Gamma_1 + r \cdot \Gamma_2 \vdash t_1[t_2/x] : B$.

Proof. Proof as in Orchard et al. [103], but where the additional premise of $\neg\mathsf{resourceAllocator}(t_2)$ is deployed in the promotion case. □

Definition 23 (Evaluation contexts)**.**

$$
\begin{array}{lcll}
Ctxt & ::= & & \textit{Contexts} \\
 & | & - & \\
 & | & Ctxt\, t & \\
 & | & v\, Ctxt & \\
 & | & [Ctxt] & \\
 & | & \mathbf{let}\, [x] = Ctxt\, \mathbf{in}\, t & \\
 & | & \mathbf{let}\, \mathbf{unit} = Ctxt\, \mathbf{in}\, t & \\
 & | & \mathbf{let}\, (x, y) = Ctxt\, \mathbf{in}\, t & \\
 & | & (Ctxt, t) & \\
 & | & (v, Ctxt) &
\end{array}
$$

Lemma 19 (Non-resource allocator context)**.** $\neg\mathsf{resourceAllocator}(Ctxt\{t\}) \implies \neg\mathsf{resourceAllocator}(Ctxt\{x\})$

Proof. Trivial by the definition of $\mathsf{resourceAllocator}(t)$ and since x is a variable and not a resource allocator. □

Lemma 20 (Non-resource allocator composition)**.** $\neg\mathsf{resourceAllocator}(Ctxt\{x\}) \wedge \neg\mathsf{resourceAllocator}(t) \implies \neg\mathsf{resourceAllocator}(Ctxt\{t\})$

Proof. By induction on the definition of contexts and the definition of $\mathsf{resourceAllocator}(t)$. □

Lemma 21 (Contexts preserve resoure allocator property)**.**

$$\mathsf{resourceAllocator}(t) \implies \mathsf{resourceAllocator}(Ctxt\{t\})$$

Proof. By induction on the grammar of context terms: the structure of context terms matches the inductive term structure of resourceAllocator.

- $Ctxt = -$ by identity;
- $Ctxt = Ctxt'\, t'$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}(Ctxt'\{t\}\, t')$ by the definition.
- $Ctxt = v\, Ctxt'$. By induction on the second premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}(v\, Ctxt'\{t\})$ by the definition.
- $Ctxt = [Ctxt']$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}([Ctxt'\{t\}])$ by the definition.
- $Ctxt = \mathbf{let}\, [x] = Ctxt'\, \mathbf{in}\, t_2$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}(\mathbf{let}\, [x] = Ctxt'\{t\}\, \mathbf{in}\, t_2)$ by the definition.

- $Ctxt = \mathbf{let}\,\mathbf{unit} = Ctxt'\,\mathbf{in}\,t_2$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}(\mathbf{let}\,\mathbf{unit} = Ctxt'\{t\}\,\mathbf{in}\,t_2)$ by the definition.
- $Ctxt = \mathbf{let}\,(x,y) = Ctxt'\,\mathbf{in}\,t_2$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}(\mathbf{let}\,(x,y) = Ctxt'\{t\}\,\mathbf{in}\,t_2)$ by the definition.
- $Ctxt = (Ctxt', t')$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}((Ctxt'\{t\}, t'))$ by the definition.
- $Ctxt = (v, Ctxt')$. By induction on the first premise and modus ponens with the antecedent then $\mathsf{resourceAllocator}(Ctxt'\{t\})$ and thus $\mathsf{resourceAllocator}((v, Ctxt'\{t\}))$ by the definition.

□

Corollary 3 (Non-resource-allocating contextual term)**.**

$$\neg\mathsf{resourceAllocator}(Ctxt\{t\}) \implies \neg\mathsf{resourceAllocator}(t)$$

Proof. Dual of Lemma 21 (modus tollens) □

Lemma 22 (Context factorisation)**.**

$$\begin{aligned}
& \Gamma \vdash Ctxt\{t\} : B \\
\implies \quad & (\exists \Gamma_1, \Gamma_2, A. \quad \Gamma_1, x : A \vdash Ctxt\{x\} : B \\
& \qquad\qquad \wedge \quad \Gamma_2 \vdash t : A \\
& \qquad\qquad \wedge \quad \Gamma = \Gamma_1 + \Gamma_2) \\
\vee \quad & (\exists \Gamma_1, \Gamma_2, A, r. \quad \neg\mathsf{resourceAllocator}(t) \\
& \qquad\qquad \wedge \quad \Gamma_1, x : [A]_r \vdash Ctxt\{x\} : B \\
& \qquad\qquad \wedge \quad [\Gamma_2] \vdash t : A \\
& \qquad\qquad \wedge \quad \Gamma = \Gamma_1 + r \cdot \Gamma_2)
\end{aligned}$$

Proof. By induction on $Ctxt$:

- $Ctxt = -$, i.e., the premise is that $\Gamma \vdash t : B$ and thus the result is given by $\Gamma_1 = \cdot$ and $\Gamma_2 = \Gamma$ with derivation:

$$\textsc{var}\ x : B \vdash x : B \qquad \text{(ctxt)}$$

  and the substitutee derivation given by the premise. Thus, we prove this case by giving the first disjunct.

- $Ctxt = Ctxt'\, t'$ where $Ctxt\{t\} = Ctxt'\{t\}\, t'$ i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash Ctxt'\{t\} : A' \multimap B \qquad \Gamma_2' \vdash t' : A'}{\Gamma \vdash Ctxt'\{t\}\, t' : B} \multimap_e$$

  where $\Gamma = \Gamma_1' + \Gamma_2'$.

  By induction on the first premise, then there are two cases:

  - (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash Ctxt'\{x\} : A' \multimap B$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + \Gamma_2''$.

    We conclude with derivation:

$$\frac{\Gamma_1'', x : A \vdash Ctxt'\{x\} : A' \multimap B \qquad \Gamma_2' \vdash t' : A'}{(\Gamma_1'' + \Gamma_2'), x : A \vdash Ctxt'\{x\}\, t' : B} \multimap_e \quad \text{(ctxt)}$$

    i.e., where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ where $\Gamma = \Gamma_1 + \Gamma_2 = \Gamma_1'' + \Gamma_2' + \Gamma_2'' = \Gamma_1' + \Gamma_2'$ from the induction and by commutativity and associativity of $+$.

    The concluding substitutee $\Gamma_2'' \vdash t : A$ is from the induction.

  - (second disjunct) We have Γ_1', Γ_2'' such that $\neg \mathsf{resourceAllocator}(t)$ and $\Gamma_1'', x : [A]_r \vdash Ctxt'\{x\} : A' \multimap B$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + r \cdot \Gamma_2''$.

    We conclude with derivation:

$$\frac{\Gamma_1'', x : [A]_r \vdash Ctxt'\{x\} : A' \multimap B \qquad \Gamma_2' \vdash t' : A'}{(\Gamma_1'' + \Gamma_2'), x : A \vdash Ctxt'\{x\}\, t' : B} \multimap_e \quad \text{(ctxt)}$$

    i.e., where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and where $\Gamma_2 = [\Gamma_2'']$ where:

$$\begin{array}{rll} \Gamma = & \Gamma_1' + \Gamma_2' & \{\text{premise }\} \\ = & \Gamma_1'' + r \cdot \Gamma_2'' + \Gamma_2' & \{\text{induction } \Gamma_1'\ \} \\ = & \Gamma_1'' + \Gamma_2' + r \cdot \Gamma_2'' & \{+ \text{ comm. and assoc.}\} \\ = & \Gamma_1 + r \cdot \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

    The concluding substitutee $[\Gamma_2''] \vdash t : A$ is from the induction and that $\neg \mathsf{resourceAllocator}(t)$.

- $Ctxt = v\, Ctxt'$ where $Ctxt\{t\} = v\, Ctxt'\{t\}$ i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash v : A' \multimap B \qquad \Gamma_2' \vdash Ctxt'\{t\} : A'}{\Gamma \vdash v\, Ctxt'\{t\} : B} \multimap_e$$

  where $\Gamma = \Gamma_1' + \Gamma_2'$.

  By induction on the second premise, then there are two cases:

– (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash Ctxt'\{x\} : A'$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_2' = \Gamma_1'' + \Gamma_2''$. We can then conclude with:

$$\frac{\Gamma_1' \vdash v : A' \multimap B \qquad \Gamma_1'', x : A \vdash Ctxt'\{x\} : A'}{\Gamma_1' + \Gamma_1'', x : A \vdash v\, Ctxt'\{x\} : B} \multimap_e \qquad \text{(ctxt)}$$

and substitutee $\Gamma_2'' \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and $\Gamma_2 = \Gamma_2''$ where $\Gamma = \Gamma_1 + \Gamma_2 = \Gamma_1' + \Gamma_1'' + \Gamma_2''$ from the induction and by commutativity and associativity of $+$.

– (second disjunct) We have Γ_1'', Γ_2'' such that $\neg\mathsf{resourceAllocator}(t)$ and $\Gamma_1'', x : [A]_r \vdash Ctxt'\{x\} : A'$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_2' = \Gamma_1'' + r \cdot \Gamma_2''$.

We can then conclude with:

$$\frac{\Gamma_1' \vdash v : A' \multimap B \qquad \Gamma_1'', x : [A]_r \vdash Ctxt'\{x\} : A'}{\Gamma_1' + \Gamma_1'', x : [A]_r \vdash v\, Ctxt'\{x\} : B} \multimap_e \qquad \text{(ctxt)}$$

and $\neg\mathsf{resourceAllocator}(t)$ (from the induction) and substitutee $[\Gamma_2''] \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and $\Gamma_2 = \Gamma_2''$ where:

$$\begin{array}{rll} \Gamma = & \Gamma_1' + \Gamma_2' & \{\text{premise }\} \\ = & \Gamma_1' + \Gamma_1'' + r \cdot \Gamma_2'' & \{\text{induction } \Gamma_1'\} \\ = & \Gamma_1 + r \cdot \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

- $Ctxt = [Ctxt']$ where $Ctxt\{t\} = [Ctxt'\{t\}]$ i.e., with premise derivation:

$$\frac{\neg\mathsf{resourceAllocator}(Ctxt'\{t\}) \qquad [\Gamma_1'] \vdash Ctxt'\{t\} : A'}{\Gamma \vdash [Ctxt'\{t\}] : \Box_r A'}\ \text{PR}'$$

where $\Gamma = r \cdot \Gamma_1'$.

By induction on the second premise, there are two cases:

– (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash Ctxt'\{x\} : A'$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + \Gamma_2''$, and we know that $[\Gamma_1'']$ since the premise derivation gives us that $[\Gamma_1']$.

We can then conclude with the derivation (giving the second disjunct):

$$\frac{\dfrac{\textit{Lemma 19}}{\neg\mathsf{resourceAllocator}(Ctxt'\{x\})} \qquad \dfrac{\dfrac{\text{ind.hyp.}}{[\Gamma_1''], x : A \vdash Ctxt'\{x\} : A'}}{[\Gamma_1''], x : [A]_1 \vdash Ctxt'\{x\} : A'}\ \text{DER}}{r \cdot \Gamma_1'', x : [A]_r \vdash [Ctxt'\{x\}] : \Box_r A'}\ \text{PR}$$

(ctxt)

with substitutee $\Gamma_2'' \vdash t : A$ (from the induction) and with $\Gamma_1 = r \cdot \Gamma_1''$ and $\Gamma_2 = \Gamma_2''$ where:

$$\begin{array}{lll} \Gamma = & r \cdot \Gamma_1' & \{\text{premise }\} \\ = & r \cdot (\Gamma_1'' + \Gamma_2'') & \{\text{induction } \Gamma_1' \} \\ = & r \cdot \Gamma_1'' + r \cdot \Gamma_2'' & \{* \text{ dist. }\} \\ = & \Gamma_1 + r \cdot \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

From the premise derivation $\neg\mathsf{resourceAllocator}(Ctxt'\{t\})$, by Corollary 3, we then have that $\neg\mathsf{resourceAllocator}(t)$ as needed here.

– (second disjunct) We have $\neg\mathsf{resourceAllocator}(t)$ and Γ_1'', Γ_2'' such that $\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : A'$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + s \cdot \Gamma_2''$, and we know that $[\Gamma_1'']$ since the premise derivation gives us that $[\Gamma_1']$.

We can then conclude with the derivation (giving the second disjunct):

$$\dfrac{\dfrac{\textit{Lemma 19}}{\neg\mathsf{resourceAllocator}(Ctxt'\{x\})} \quad \dfrac{\text{ind.hyp.}}{[\Gamma_1''], x : [A]_s \vdash Ctxt'\{x\} : A'}}{r \cdot \Gamma_1'', x : [A]_{r*s} \vdash [Ctxt'\{x\}] : \Box_r A'}\ \textsc{pr}$$

(ctxt)

with substitutee $[\Gamma_2''] \vdash t : A$ (from the induction) and with $\Gamma_1 = r \cdot \Gamma_1''$ and $\Gamma_2 = \Gamma_2''$ where:

$$\begin{array}{lll} \Gamma = & r \cdot \Gamma_1' & \{\text{premise }\} \\ = & r \cdot (\Gamma_1'' + s \cdot \Gamma_2'') & \{\text{induction } \Gamma_1' \} \\ = & r \cdot \Gamma_1'' + r * s \cdot \Gamma_2'' & \{* \text{ dist. }\} \\ = & \Gamma_1 + r * s \cdot \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

and $\neg\mathsf{resourceAllocator}(t)$ from the induction, as required.

- $Ctxt = \mathbf{let}\,[y] = Ctxt'\,\mathbf{in}\,t_2$ where $Ctxt\{t\} = \mathbf{let}\,[y] = Ctxt'\{t\}\,\mathbf{in}\,t_2$. i.e., with premise derivation:

$$\dfrac{\Gamma_1' \vdash Ctxt'\{t\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{\Gamma_1' + \Gamma_2' \vdash \mathbf{let}\,[y] = Ctxt'\{t\}\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

where $\Gamma = \Gamma_1' + \Gamma_2'$.

By induction on the first premise, then there are two case:

– (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash Ctxt'\{x\} : A'$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + \Gamma_2''$.

We can then conclude with:

$$\dfrac{\Gamma_1'', x : A \vdash Ctxt'\{x\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{(\Gamma_1'' + \Gamma_2'), x : A \vdash \mathbf{let}\,[y] = Ctxt'\{x\}\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

and substitutee $\Gamma_2'' \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{aligned}\Gamma = &\ \Gamma_1' + \Gamma_2' && \{\text{premise }\}\\ = &\ \Gamma_1'' + \Gamma_2'' + \Gamma_2' && \{\text{induction } \Gamma_1'\ \}\\ = &\ \Gamma_1'' + \Gamma_2' + \Gamma_2'' && \{+ \text{ comm. and assoc.}\}\\ = &\ \Gamma_1 + \Gamma_2 && \{\text{concluding contexts}\}\end{aligned}$$

– (second disjunct) We have $\neg\mathsf{resourceAllocator}(t)$ and Γ_1'', Γ_2'' such that $\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : A'$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + s \cdot \Gamma_2''$.

We can then conclude with:

$$\frac{\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{(\Gamma_1'' + \Gamma_2'), x : [A]_s \vdash \mathbf{let}\ [y]\ = Ctxt'\{x\}\ \mathbf{in}\ t_2 : B}\ \textsc{elim}$$

and substitutee $[\Gamma_2''] \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{aligned}\Gamma = &\ \Gamma_1' + \Gamma_2' && \{\text{premise }\}\\ = &\ \Gamma_1'' + s \cdot \Gamma_2'' + \Gamma_2' && \{\text{induction } \Gamma_1'\ \}\\ = &\ \Gamma_1'' + \Gamma_2' + s \cdot \Gamma_2'' && \{+ \text{ comm. and assoc.}\}\\ = &\ \Gamma_1 + s \cdot \Gamma_2 && \{\text{concluding contexts}\}\end{aligned}$$

and $\neg\mathsf{resourceAllocator}(t)$ as required, from the induction.

- $Ctxt = \mathbf{let\ unit}\ = Ctxt'\ \mathbf{in}\ t_2$ where $Ctxt\{t\} = \mathbf{let\ unit} = Ctxt'\{t\}\ \mathbf{in}\ t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash Ctxt'\{t\} : \mathsf{unit} \qquad \Gamma_2' \vdash t_2 : B}{\Gamma_1' + \Gamma_2' \vdash \mathbf{let\ unit} = Ctxt'\{t\}\ \mathbf{in}\ t_2 : B}\ \textsc{elim}$$

where $\Gamma = \Gamma_1' + \Gamma_2'$.

By induction on the first premise, then there are two cases:

– (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash Ctxt'\{x\} : \mathsf{unit}$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + \Gamma_2''$.

We can then conclude with:

$$\frac{\Gamma_1'', x : A \vdash Ctxt'\{x\} : \mathsf{unit} \qquad \Gamma_2' \vdash t_2 : B}{(\Gamma_1'' + \Gamma_2'), x : A \vdash \mathbf{let}\ [y]\ = Ctxt'\{x\}\ \mathbf{in}\ t_2 : B}\ \textsc{elim}$$

and substitutee $\Gamma_2'' \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{aligned}\Gamma = &\ \Gamma_1' + \Gamma_2' && \{\text{premise }\}\\ = &\ \Gamma_1'' + \Gamma_2'' + \Gamma_2' && \{\text{induction } \Gamma_1'\ \}\\ = &\ \Gamma_1'' + \Gamma_2' + \Gamma_2'' && \{+ \text{ comm. and assoc.}\}\\ = &\ \Gamma_1 + \Gamma_2 && \{\text{concluding contexts}\}\end{aligned}$$

– (second disjunct) We have $\neg\mathsf{resourceAllocator}(t)$ and Γ_1'', Γ_2'' such that $\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : \mathsf{unit}$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + s \cdot \Gamma_2''$.

We can then conclude with:

$$\frac{\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : \mathsf{unit} \qquad \Gamma_2' \vdash t_2 : B}{(\Gamma_1'' + \Gamma_2'), x : [A]_s \vdash \mathbf{let}\ [y]\ = Ctxt'\{x\}\ \mathbf{in}\ t_2 : B}\ \textsc{elim}$$

and substitutee $[\Gamma_2''] \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{array}{rll} \Gamma = & \Gamma_1' + \Gamma_2' & \{\text{premise }\} \\ = & \Gamma_1'' + s \cdot \Gamma_2'' + \Gamma_2' & \{\text{induction } \Gamma_1'\ \} \\ = & \Gamma_1'' + \Gamma_2' + s \cdot \Gamma_2'' & \{+ \text{ comm. and assoc.}\} \\ = & \Gamma_1 + s \cdot \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

and $\neg\mathsf{resourceAllocator}(t)$ as required, from the induction.

- $Ctxt = \mathbf{let}\,(x, y) = Ctxt\,\mathbf{in}\,t_2$ the same as above, modulo the additional free variables for x and y in the scope of t_2;
- $Ctxt = (Ctxt', t')$ where $Ctxt\{t\} = (Ctxt'\{t\}, t')$. i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash Ctxt'\{t\} : A' \qquad \Gamma_2' \vdash t' : B'}{\Gamma_1' + \Gamma_2' \vdash (Ctxt'\{t\}, t') : A' \otimes B'}\ \otimes_i$$

where $\Gamma = \Gamma_1' + \Gamma_2'$.

By induction on the first premise, then there are two case:

– (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash Ctxt'\{x\} : A'$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + \Gamma_2''$.

We can then conclude with:

$$\frac{\Gamma_1'', x : A \vdash Ctxt'\{x\} : A' \qquad \Gamma_2' \vdash t' : B'}{\Gamma_1'' + \Gamma_2', x : A \vdash (Ctxt'\{t\}, t') : A' \otimes B'}\ \otimes_i$$

and substitutee $\Gamma_2'' \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{array}{rll} \Gamma = & \Gamma_1' + \Gamma_2' & \{\text{premise }\} \\ = & \Gamma_1'' + \Gamma_2'' + \Gamma_2' & \{\text{induction } \Gamma_1'\ \} \\ = & \Gamma_1'' + \Gamma_2' + \Gamma_2'' & \{+ \text{ comm. and assoc.}\} \\ = & \Gamma_1 + \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

– (second disjunct) We have $\neg\mathsf{resourceAllocator}(t)$ and Γ_1'', Γ_2'' such that $\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : A'$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_1' = \Gamma_1'' + s \cdot \Gamma_2''$.

We can then conclude with:

$$\frac{\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : A' \qquad \Gamma_2' \vdash t' : B'}{\Gamma_1'' + \Gamma_2', x : [A]_s \vdash (Ctxt'\{t\}, t') : A' \otimes B'} \otimes_i$$

and substitutee $[\Gamma_2''] \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1'' + \Gamma_2'$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{array}{rll} \Gamma = & \Gamma_1' + \Gamma_2' & \{\text{premise }\} \\ = & \Gamma_1'' + s \cdot \Gamma_2'' + \Gamma_2' & \{\text{induction } \Gamma_1' \} \\ = & \Gamma_1'' + \Gamma_2' + s \cdot \Gamma_2'' & \{+ \text{ comm. and assoc.}\} \\ = & \Gamma_1 + s \cdot \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

and $\neg\mathsf{resourceAllocator}(t)$ as required, from the induction.

- $Ctxt = (v, Ctxt')$ where $Ctxt\{t\} = (v, Ctxt'\{t\})$. i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash v : A' \qquad \Gamma_2' \vdash Ctxt'\{t\} : B'}{\Gamma_1' + \Gamma_2' \vdash (v, Ctxt'\{t\}) : A' \otimes B'} \otimes_i$$

  where $\Gamma = \Gamma_1' + \Gamma_2'$.

  By induction on the second premise, then there are two case:

  - (first disjunct) We have Γ_1'', Γ_2'' such that $\Gamma_1'', x : A \vdash Ctxt'\{x\} : A'$ and $\Gamma_2'' \vdash t : A$ such that $\Gamma_2' = \Gamma_1'' + \Gamma_2''$.

    We can then conclude with:

$$\frac{\Gamma_1' \vdash v : A' \qquad \Gamma_1'', x : A \vdash Ctxt'\{x\} : B'}{(\Gamma_1' + \Gamma_1''), x : A \vdash (v, Ctxt'\{x\}) : A' \otimes B'} \otimes_i$$

    and substitutee $\Gamma_2'' \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{array}{rll} \Gamma = & \Gamma_1' + \Gamma_2' & \{\text{premise }\} \\ = & \Gamma_1' + \Gamma_1'' + \Gamma_2'' & \{\text{induction } \Gamma_1' \} \\ = & \Gamma_1 + \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

  - (second disjunct) We have $\neg\mathsf{resourceAllocator}(t)$ and Γ_1'', Γ_2'' such that $\Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : A'$ and $[\Gamma_2''] \vdash t : A$ such that $\Gamma_2' = \Gamma_1'' + s \cdot \Gamma_2''$.

    We can then conclude with:

$$\frac{\Gamma_1' \vdash v : A' \qquad \Gamma_1'', x : [A]_s \vdash Ctxt'\{x\} : B'}{(\Gamma_1' + \Gamma_1''), x : [A]_s \vdash (v, Ctxt'\{x\}) : A' \otimes B'} \otimes_i$$

    and substitutee $[\Gamma_2''] \vdash t : A$ (from the induction) where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and $\Gamma_2 = \Gamma_2''$ therefore:

$$\begin{array}{rll} \Gamma = & \Gamma_1' + \Gamma_2' & \{\text{premise }\} \\ = & \Gamma_1' + \Gamma_1'' + s \cdot \Gamma_2'' & \{\text{induction } \Gamma_1' \} \\ = & \Gamma_1 + s \cdot \Gamma_2 & \{\text{concluding contexts}\} \end{array}$$

    and $\neg\mathsf{resourceAllocator}(t)$ as required, from the induction.

□

Lemma 23 (Context application (linear))**.**

$$\begin{aligned} & \Gamma_1, x : A \vdash Ctxt\{x\} : B \\ \wedge\ & \Gamma_2 \vdash t : A \\ \Longrightarrow\ & \Gamma_1 + \Gamma_2 \vdash Ctxt\{t\} : B \end{aligned}$$

Proof. By induction on the structure of evaluation contexts

- $Ctxt = -$, i.e., the premise is:

$$\frac{}{x : A \vdash x : A}\ \textsc{var} \qquad \text{(ctxt)}$$

  (i.e., $\Gamma_1 = \cdot$ and $A = B$).

  Then the conclusion follows by $Ctxt\{t\} = t$ therefore the second premise provides the conclusion.

- $Ctxt = Ctxt'\, t'$ where $Ctxt\{x\} = Ctxt'\{x\}\, t'$ i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : A' \multimap B \qquad \Gamma_2' \vdash t' : A'}{(\Gamma_1' + \Gamma_2'), x : A \vdash Ctxt'\{x\}\, t' : B}\ \multimap_e$$

  By induction on the first premise, yielding: $\Gamma_1' + \Gamma_2 \vdash Ctxt'\{t\} : A' \multimap B$ therefore the conclusion can be constructed as:

$$\frac{\Gamma_1' + \Gamma_2 \vdash Ctxt'\{t\} : A' \multimap B \qquad \Gamma_2' \vdash t' : A'}{(\Gamma_1' + \Gamma_2) + \Gamma_2' \vdash Ctxt'\{t\}\, t' : B}\ \multimap_e$$

  which by commutativity and associativity of $+$ matches the conclusion.

- $Ctxt = v\, Ctxt'$ where $Ctxt\{x\} = v\, Ctxt'\{x\}$ i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash v : A' \multimap B \qquad \Gamma_2', x : A \vdash Ctxt'\{x\} : A'}{(\Gamma_1' + \Gamma_2'), x : A \vdash v\, Ctxt'\{x\} : B}\ \multimap_e$$

  By induction on the second premise, yielding: $\Gamma_2' + \Gamma_2 \vdash Ctxt'\{t\} : A'$ therefore the conclusion can be constructed as:

$$\frac{\Gamma_1' \vdash v : A' \multimap B \qquad \Gamma_2' + \Gamma_2 \vdash Ctxt'\{t\} : A'}{(\Gamma_1' + \Gamma_2') + \Gamma_2 \vdash v\, Ctxt'\{t\} : B}\ \multimap_e$$

  satisfying the goal of the lemma.

- $Ctxt = [Ctxt']$ where $Ctxt\{x\} = [Ctxt'\{x\}]$ i.e., with premise derivation:

$$\frac{[\Gamma_1'] \vdash Ctxt'\{x\} : A'}{r \cdot \Gamma_1' \vdash [Ctxt'\{x\}] : \Box_r A'}\ \textsc{pr}$$

  where Γ_1' must contain x. However, this leads to a contradiction since there cannot possibly be a linear assumption x in the conclusion due to the form of the typing. Therefore, the premise is false and this case is trivial.

- $Ctxt = \mathbf{let}\,[y] = Ctxt'\,\mathbf{in}\,t_2$ where $Ctxt\{x\} = \mathbf{let}\,[y] = Ctxt'\{x\}\,\mathbf{in}\,t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{(\Gamma_1' + \Gamma_2'), x : A \vdash \mathbf{let}\,[y] = Ctxt'\{x\}\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

  By induction on the first premise, yielding: $\Gamma_1' + \Gamma_2 \vdash Ctxt'\{t\} : \Box_r A'$. Therefore the conclusion can be constructed as:

$$\frac{\Gamma_1' + \Gamma_2 \vdash Ctxt'\{t\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{(\Gamma_1' + \Gamma_2) + \Gamma_2' \vdash \mathbf{let}\,[y] = Ctxt'\{t\}\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

  which by commutativity and associaitivity of $+$ matches the goal.

- $Ctxt = \mathbf{let}\,\mathbf{unit} = Ctxt'\,\mathbf{in}\,t_2$ where $Ctxt\{x\} = \mathbf{let}\,\mathbf{unit} = Ctxt'\{x\}\,\mathbf{in}\,t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : \mathsf{unit} \qquad \Gamma_2' \vdash t_2 : B}{(\Gamma_1' + \Gamma_2'), x : A \vdash \mathbf{let}\,[y] = Ctxt'\{x\}\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

  By induction on the first premise, yielding: $\Gamma_1' + \Gamma_2 \vdash Ctxt'\{t\} : \Box_r A'$. Therefore the conclusion can be constructed as:

$$\frac{\Gamma_1' + \Gamma_2 \vdash Ctxt'\{t\} : \mathsf{unit} \qquad \Gamma_2' \vdash t_2 : B}{(\Gamma_1' + \Gamma_2) + \Gamma_2' \vdash \mathbf{let}\,[y] = Ctxt'\{t\}\,\mathbf{in}\,t_2 : B}\ \textsc{elim}$$

  which by commutativity and associaitivity of $+$ matches the goal.

- $Ctxt = \mathbf{let}\,(x, y) = Ctxt\,\mathbf{in}\,t_2$ the same as above, modulo the additional free variables for x and y in the scope of t_2;

- $Ctxt = (Ctxt', t')$ where $Ctxt\{x\} = (Ctxt'\{x\}, t')$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : A' \qquad \Gamma_2' \vdash t' : B'}{(\Gamma_1' + \Gamma_2'), x : A \vdash (Ctxt'\{x\}, t') : A' \otimes B'}\ \otimes_i$$

By induction on the first premise then $\Gamma_1' + \Gamma_2 \vdash Ctxt'\{t\} : A'$. Therefore the goal can be constructed as:

$$\frac{\Gamma_1' + \Gamma_2 \vdash Ctxt'\{t\} : A' \qquad \Gamma_2' \vdash t' : B'}{(\Gamma_1' + \Gamma_2) + \Gamma_2' \vdash (Ctxt'\{x\}, t') : A' \otimes B'} \otimes_i$$

satisfying the goal by commutativity and associativity of $+$.

- $Ctxt = (v, Ctxt')$ where $Ctxt\{x\} = (v, Ctxt'\{x\})$. i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash v : A' \qquad \Gamma_2', x : A \vdash Ctxt'\{x\} : B'}{(\Gamma_1' + \Gamma_2'), x : A \vdash (v, Ctxt'\{x\}) : A' \otimes B'} \otimes_i$$

By induction on the second premise we have $\Gamma_2' + \Gamma_2 \vdash Ctxt'\{t\} : B'$. Therefore the goal can be constructed as:

$$\frac{\Gamma_1' \vdash v : A' \qquad \Gamma_2' + \Gamma_2 \vdash Ctxt'\{t\} : B'}{\Gamma_1' + (\Gamma_2' + \Gamma_2) \vdash (v, Ctxt'\{x\}) : A' \otimes B'} \otimes_i$$

satisfying the goal by associativity of $+$.

□

Lemma 24 (Context application (graded))**.**

$$\begin{aligned} & \Gamma_1, x : [A]_r \vdash Ctxt\{x\} : B \\ \wedge\ & [\Gamma_2] \vdash t : A \\ \wedge\ & \neg \mathsf{resourceAllocator}(t) \\ \implies\ & \Gamma_1 + r \cdot \Gamma_2 \vdash Ctxt\{t\} : B \end{aligned}$$

Proof. By induction on the structure of evaluation contexts and their typing.

Note that for all possible contexts, it is possible that typing is by (non-syntax directed) (approx) rule and thus we have:

$$\frac{\Gamma_1, x : [A]_s \vdash t' : B \qquad s \sqsubseteq r}{\Gamma_1, x : [A]_r \vdash t' : B}\ \textsc{approx}$$

Then by induction on the premise we have $\Gamma_1 + s \cdot \Gamma_2 \vdash t' : B$.

Let $\Gamma_2 = x_i : [A_i]_{s_i}$ for $1 \leq i \leq |\Gamma_2|$ since by the premise every assumption is graded in the context. Then we can then construct the conclusion as follows:

$$\frac{\Gamma_1 + s \cdot \Gamma_2 \vdash t' : B \qquad \dfrac{\dfrac{}{s_i \sqsubseteq s_i}\ \textsc{refl} \qquad s \sqsubseteq r}{s * s_i \sqsubseteq r * s_i}\ \textsc{monoton.+unit}}{\Gamma_1 + r \cdot \Gamma_2 \vdash t' : B}\ \textsc{approx}^*$$

The rest is by induction now we have considered this possibility at each context term:

- $Ctxt = -$.

  There are therefore two possibilies for the derivation of the the premise:

  - (var/der combination)

$$\dfrac{\dfrac{}{x : A \vdash x : A}\,\textsc{var}}{x : [A]_1 \vdash x : A}\,\textsc{der}$$

    (i.e., $\Gamma_1 = \cdot$ and $A = B$ and $r = 1$).

    Since $1 \cdot \Gamma_2 = \Gamma_2$ and since $Ctxt\{t_2\} = t_2$ then we can conclude with the second premise $\Gamma_2 \vdash t_2 : B$

  - (var/der/approx combination)

$$\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}\,\textsc{var}}{x : [A]_1 \vdash x : A}\,\textsc{der} \qquad 1 \sqsubseteq r}{x : [A]_r \vdash x : A}\,\textsc{approx}$$

    (i.e., $\Gamma_1 = \cdot$ and $A = B$).

    Let $\Gamma_2 = x_i : [A]_{s_i}$ (since by the premise every assumption is graded in the context). Then we can then construct the conclusion as follows:

$$\dfrac{\dfrac{}{\Gamma_2 \vdash t : A}\,(\textsc{second premise}) \qquad \dfrac{\dfrac{}{s_i \sqsubseteq s_i}\,\textsc{refl} \qquad 1 \sqsubseteq r}{s_i \sqsubseteq r * s_i}\,\textsc{monoton.+unit}}{r \cdot \Gamma_2 \vdash t : A}\,\textsc{approx}^*$$

- $Ctxt = Ctxt'\, t'$ where $Ctxt\{x\} = Ctxt'\{x\}\, t'$. i.e., with premise derivation:

$$\dfrac{\Gamma_1', x : [A]_{s_1} \vdash Ctxt'\{x\} : A' \multimap B \qquad \Gamma_2', x : [A]_{s_2} \vdash t' : A'}{(\Gamma_1' + \Gamma_2'), x : [A]_{s_1 + s_2} \vdash Ctxt'\{x\}\, t' : B} \multimap_e$$

  i.e., where $r = s_1 + s_2$. This derivation applies without loss of generality here for application since we can set $s2 = 0$ in the case that $x \notin \mathsf{FV}(t')$.

  By induction on both premise we attain:

$$\Gamma_1' + s_1 \cdot \Gamma_2 \vdash Ctxt'\{t\} : A' \multimap B \qquad \text{(ih1)}$$
$$\Gamma_2' + s_2 \cdot \Gamma_2 \vdash t' : A' \qquad \text{(ih2)}$$

From which we construct the concluding judgment:

$$\frac{\Gamma_1' + s_1 \cdot \Gamma_2 \vdash Ctxt'\{t\} : A' \multimap B \qquad \Gamma_2' + s_2 \cdot \Gamma_2 \vdash t' : A'}{\Gamma_1' + s_1 \cdot \Gamma_2 + \Gamma_2' + s_2 \cdot \Gamma_2 \vdash Ctxt'\{t\}\, t' : B} \multimap_e$$

which since $r = s_1 + s_2$, and by distributivity of $*$ over $+$, and commutativity and associativity of $+$ yields: $\Gamma_1' + \Gamma_2' + r \cdot \Gamma_2 \vdash Ctxt'\{t\}\, t' : B$.

- $Ctxt = v\, Ctxt'$ where $Ctxt\{x\} = v\, Ctxt'\{x\}$ i.e., with premise derivation:

$$\frac{\Gamma_1', x : [A]_{s_1} \vdash v : A' \multimap B \qquad \Gamma_2', x : [A]_{s_2} \vdash Ctxt'\{x\} : A'}{(\Gamma_1' + \Gamma_2'), x : [A]_{s_1 + s_2} \vdash v\, Ctxt'\{x\} : B} \multimap_e$$

This case proceeds as above by induction on both premises.

- $Ctxt = [Ctxt']$ where $Ctxt\{x\} = [Ctxt'\{x\}]$ i.e., with premise derivation:

$$\frac{\neg \mathsf{resourceAllocator}(Ctxt'\{x\}) \qquad [\Gamma_1'], x : [A]_{s_2} \vdash Ctxt'\{x\} : A'}{s_1 \cdot (\Gamma_1', x : [A]_{s_2}) \vdash [Ctxt'\{x\}] : \Box_r A'} \text{PR}'$$

i.e., where $r = s_1 * s_2$ and $\Gamma_1 = s_1 \cdot \Gamma_1'$

By induction on the second premise we attain:

$$\Gamma_1' + s_2 \cdot \Gamma_2 \vdash Ctxt'\{t\} : A' \qquad \text{(ih)}$$

From which we construct the concluding judgment:

$$\frac{\dfrac{\textit{Lemma 20}}{\neg \mathsf{resourceAllocator}(Ctxt'\{t\})} \qquad \Gamma_1' + s_2 \cdot \Gamma_2 \vdash Ctxt'\{t\} : A'}{s_1 \cdot (\Gamma_1' + s_2 \cdot \Gamma_2) \vdash [Ctxt'\{t\}] : \Box_r A'} \text{PR}'$$

which by distributivity of $*$ over $+$ and associativity of $*$ yields: $s_1 \cdot \Gamma_1' + (s_1 * s_2) \cdot \Gamma_2 \vdash [Ctxt'\{t\}] : \Box_r A'$. and by $r = s_1 * s_2$ and $\Gamma_1 = s_1 \cdot \Gamma_1'$ yields the goal: $\Gamma_1 + r \cdot \Gamma_2 \vdash [Ctxt'\{t\}] : A'$

- $Ctxt = \mathbf{let}\, [y] = Ctxt'\, \mathbf{in}\, t_2$ where $Ctxt\{x\} = \mathbf{let}\, [y] = Ctxt'\{x\}\, \mathbf{in}\, t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : \Box_r A' \qquad \Gamma_2', y : [A']_r \vdash t_2 : B}{(\Gamma_1' + \Gamma_2'), x : A \vdash \mathbf{let}\, [y] = Ctxt'\{x\}\, \mathbf{in}\, t_2 : B} \text{ELIM}$$

Proceeds by induction similarly to the (App-L) case, with no extra context maniulation necessary.

- $Ctxt = \mathbf{let}\, \mathbf{unit} = Ctxt'\, \mathbf{in}\, t_2$ where $Ctxt\{x\} = \mathbf{let}\, \mathbf{unit} = Ctxt'\{x\}\, \mathbf{in}\, t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : \mathsf{unit} \qquad \Gamma_2' \vdash t_2 : B}{(\Gamma_1' + \Gamma_2'), x : A \vdash \mathbf{let}\, [y] = Ctxt'\{x\}\, \mathbf{in}\, t_2 : B} \text{ELIM}$$

Proceeds by induction similarly to the (App-L) case, with no extra context maniulation necessary.

- $Ctxt = \mathbf{let}\,(y,z) = Ctxt'\,\mathbf{in}\,t_2$ where $Ctxt\{x\} = \mathbf{let}\ (y,z) = Ctxt'\{x\}\ \mathbf{in}\ t_2$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : A_1 \otimes A_2 \qquad \Gamma_2', y : A_1, z : A_2 \vdash t_2 : B}{(\Gamma_1' + \Gamma_2'), x : A \vdash \mathbf{let}\ (y,z) = Ctxt'\{x\}\ \mathbf{in}\ t_2 : B}\ \textsc{elim}$$

  Proceeds by induction similarly to the (App-L) case, with no extra context maniulation necessary.

- $Ctxt = (Ctxt', t')$ where $Ctxt\{x\} = (Ctxt'\{x\}, t')$. i.e., with premise derivation:

$$\frac{\Gamma_1', x : A \vdash Ctxt'\{x\} : A' \qquad \Gamma_2' \vdash t' : B'}{(\Gamma_1' + \Gamma_2'), x : A \vdash (Ctxt'\{x\}, t') : A' \otimes B'}\ \otimes_i$$

  Proceeds by induction similarly to the (App-L) case, with no extra context maniulation necessary.

- $Ctxt = (v, Ctxt')$ where $Ctxt\{x\} = (v, Ctxt'\{x\})$. i.e., with premise derivation:

$$\frac{\Gamma_1' \vdash v : A' \qquad \Gamma_2', x : A \vdash Ctxt'\{x\} : B'}{(\Gamma_1' + \Gamma_2'), x : A \vdash (v, Ctxt'\{x\}) : A' \otimes B'}\ \otimes_i$$

  Proceeds by induction similarly to the (App-R) case, with no extra context maniulation necessary.

□

Lemma 25. *Value lemma*
Given $\Gamma \vdash v : A$ then, depending on the type, the shape of v can be inferred:

- *$A = A' \multimap B$ then $v = \lambda x.v'$ or a partially applied primitive term p.*
- *$A = \Box_r A'$ then $v = [v']$.*
- *$A = A' \otimes B$ then $v = (v_1, v_2)$.*
- *$A = \mathsf{unit}$ then $v = \mathbf{unit}$.*
- *$A = \mathbf{Chan}\,P$ then $v = c$.*

Proof. Recall that the value terms sub-grammar is:

$$v ::= (v_1, v_2) \mid \mathbf{unit} \mid [v] \mid \lambda x.v \mid c \mid p \quad \text{(value terms sub-grammar)}$$

where p are partially-applied primitives. In other words,

$$\begin{aligned} p ::= {} & \mathsf{close} \mid \mathsf{send} \mid \mathsf{send}\,c \mid \mathsf{receive} \mid \mathsf{forkLinear} \mid \mathsf{forkNonLinear} \\ & \mid \mathsf{forkReplicate} \mid \mathsf{forkReplicate}\,c \\ & \mid \mathsf{forkMulticast} \mid \mathsf{forkMulticast}\,c \end{aligned}$$

We then proceed by case analysis on the type A to match the structure of the lemma. In each case we must consider what possible values can be assigned the type A and by which rules.

In all cases, there exist additional derivations based on dereliction, weakening and approximation, e.g., for the case where $A = A' \multimap B$:

$$\frac{\Gamma, x : A' \vdash t : A' \multimap B}{\Gamma, x : \square_1 A' \vdash t : A' \multimap B}\ \textsc{der}$$

$$\frac{\Gamma, x : A' \vdash t : A' \multimap B}{\Gamma, x : \square_1 A', 0 \cdot \gamma \vdash t : A' \multimap B}\ \textsc{der}$$

$$\frac{\Gamma, y : \square_r A'', \Gamma', x : A' \vdash t : A' \multimap B \quad r \sqsubseteq s}{\Gamma, y : \square_s A'', \Gamma', x : A' \vdash t : A' \multimap B}\ \textsc{der}$$

In all of these cases we can apply induction on the premise to get the result since the term is preserved between the premise and the conclusion.

We elide handling this separately each time in the cases that follow as the reasoning through dereliction is the same each time.

- $A = A' \multimap B$ then there are two classes of possible typing:
  - Abstract term:

    $$\frac{\Gamma, x : A' \vdash v' : B}{\Gamma \vdash \lambda x.v' : A' \multimap B}\ \multimap_i$$

    thus $v = \lambda x.v'$ as in the lemma statement.
  - Primitive term p formed by an application of zero or more values to a primitive operation, of which there are then ten possibilities:
    1. $$\frac{}{\varnothing \vdash \mathsf{close} : \mathbf{Chan}\ \mathbf{End} \multimap \mathsf{unit}}\ \textsc{close}$$

       then $v = \mathsf{close}$.
    2. $$\frac{}{\varnothing \vdash \mathsf{send} : A \otimes \mathbf{Chan}\ (\mathbf{Send}\ A\ P) \multimap \mathbf{Chan}\ P}\ \textsc{send}$$

       then $v = \mathsf{send}$.
    3. $$\frac{\dfrac{}{c : \mathbf{Chan}\ P \vdash c : \mathbf{Chan}\ P}\ \textsc{var} \qquad \dfrac{}{\varnothing \vdash \mathsf{send} : A \otimes \mathbf{Chan}\ (\mathbf{Send}\ A\ P) \multimap \mathbf{Chan}\ P}\ \textsc{send}}{c : \mathbf{Chan}\ P \vdash \mathsf{send}\ c : A \multimap \mathbf{Chan}\ P}\ \multimap_e$$

       then $v = \mathsf{send}\ c$.

4.

$$\frac{}{\emptyset \vdash \mathsf{receive} : \mathbf{Chan}\,(\mathbf{Recv}\ A\ P) \multimap (A \otimes \mathbf{Chan}\,P)}\ \textsc{recv}$$

then $v = \mathsf{receive}$.

5.

$$\frac{}{\emptyset \vdash \mathsf{forkLinear} : (\mathbf{Chan}\,P \multimap \mathsf{unit}) \multimap \mathbf{Chan}\,\overline{P}}\ \textsc{fork}$$

then $v = \mathsf{forkLinear}$.

6.

$$\frac{}{\emptyset \vdash \mathsf{forkNonLinear} : (\square_r \mathbf{Chan}\,P \multimap \mathsf{unit}) \multimap \square_r(\mathbf{Chan}\,\overline{P})}\ \textsc{forkNonLin}$$

then $v = \mathsf{forkNonLinear}$.

7.

$$\frac{\mathsf{ReceivePrefix}(P)}{\emptyset \vdash \mathsf{forkReplicate} : \square_{0..n}(\mathbf{Chan}\,P \multimap \mathsf{unit}) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\,n\,(\square_{0..1}(\mathbf{Chan}\,\overline{P}))}\ \textsc{forkRep}$$

then $v = \mathsf{forkReplicate}$.

8.

$$\frac{\textsc{Var} + \textsc{Pr} \quad \dfrac{\mathsf{ReceivePrefix}(P)}{\emptyset \vdash \mathsf{forkReplicate} : \square_{0..n}(\mathbf{Chan}\,P \multimap \mathsf{unit}) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\,n\,(\square_{0..1}(\mathbf{Chan}\,\overline{P}))}\ \textsc{forkRep}}{c : \square_{0..1}\mathbf{Chan}\,P \vdash \mathsf{forkReplicate}\,[c] : \mathbf{N}\ n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})}\ \multimap_e$$

then $v = \mathsf{forkReplicate}\,[c]$.

9.

$$\frac{\mathsf{Sends}(P)}{\emptyset \vdash \mathsf{forkMulticast} : (\mathbf{Chan}\,(\mathsf{Graded}_n P) \multimap \mathsf{unit}) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})}\ \textsc{forkMulti}$$

then $v = \mathsf{forkMulticast}$.

10.

$$\frac{\textsc{Var} \quad \dfrac{\mathsf{Sends}(P)}{\emptyset \vdash \mathsf{forkMulticast} : (\mathbf{Chan}\,(\mathsf{Graded}_n P) \multimap \mathsf{unit}) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})}\ \textsc{forkMulti}}{c : \mathbf{Chan}\,(\mathsf{Graded}_n P) \vdash \mathsf{forkMulticast}\,c : \mathbf{N}\ n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})}\ \multimap_e$$

then $v = \mathsf{forkMulticast}\,c$.

- $A = \square_r A'$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{\Gamma \vdash v' : A'}{r \cdot \Gamma \vdash [v'] : \square_r A'}\ \textsc{pr}$$

thus $v = [v']$ as in the lemma statement.

- $A = A' \otimes B$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{\Gamma_1 \vdash v_1 : A' \quad \Gamma_2 \vdash v_2 : B}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : A' \otimes B} \otimes_i$$

  thus $v = (v_1, v_2)$ as in the lemma statement.

- $A = \mathsf{unit}$ then there is only one possible non-dereliction typing of a value at that type:

$$\frac{}{\varnothing \vdash \mathbf{unit} : \mathsf{unit}} \textsc{unit}_i$$

  thus $v = \mathbf{unit}$ as in the lemma statement.

- $A = \mathbf{Chan}\, P$ then the only possible typing that is a value is given by:

$$\frac{}{c : \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P} \textsc{var}$$

  thus $v = c$ as in the lemma statement, regardless of the protocol P.

□

D.3 TYPE SAFETY

D.3.1 Progress proof

Theorem 11 (Progress terms)**.**

$$\begin{aligned} & \gamma \vdash t : A \\ \wedge\ & \gamma \vdash \mathcal{C} \\ \wedge\ & \mathsf{buffered}(\mathcal{C}, \Delta) \\ \wedge\ & \gamma \vdash \mathcal{P} : B \\ \implies\ & \mathsf{value}(t) \vee \exists t', \mathcal{C}', \mathcal{P}'.\ (\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t';\ \mathcal{C}') \end{aligned}$$

Proof. We prove this by induction on the three judgments forming the assumptions.

First, we case split on the typing of process configurations: $\gamma \vdash \mathcal{P} : B$.

- (cons)

$$\frac{\begin{array}{c}\Gamma_1, \gamma_1 \vdash \mathcal{P} : A \\ \Gamma_2, \gamma_2 \vdash t : B\end{array}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2 \vdash \mathcal{P} \mid t^{\mathcal{R}} : A} \textsc{cons}\mathcal{R}$$

In this case, we can write the process configuration $\mathcal{P}$ as $\mathcal{P}' \mid t''$, or in other words as another process configuration $\mathcal{P}'$ extended with some term t''.

By induction, there are two possibilities here: either t'' is itself a value, or there exists some reduction $(\mathcal{P}';\ \mathcal{C}) \rightsquigarrow (\mathcal{P}'';\ \mathcal{C}')$.

If there is such a reduction, then we can make a global reduction using the following rule:

$$\frac{(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')}{(\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t;\ \mathcal{C}')}\ \textsc{par}$$

and the overall progress theorem is satisified.

Otherwise, we must consider the possible reductions for the additional term t.

- (empty)

$$\frac{}{\emptyset \vdash \cdot : \bot}\ \textsc{empty}$$

Since the process configuration is empty in this case, it cannot make any reductions, so again we only need to consider the possible reductions originating from the term t.

We then induct on the last step of the typing derivation $\gamma_1 \vdash t : A$. Note that in some cases the process and channel configurations do not come into play, in which case the proof proceeds roughly the same as the standard progress theorem on terms; either t is a value, in which case the theorem holds directly by induction, or we can take a step $t \rightsquigarrow t'$, in which case we will be able to take a global step via the Reduce rule. In these cases we can generally ignore the presence of the runtime context γ_1.

  - (var)

$$\frac{}{x : A \vdash x : A}\ \textsc{var}$$

    Trivial since the variable x cannot be reduced and so is necessarily a value.

  - (abs)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \multimap_i$$

    Trivial since $(\lambda x.t)$ is already a value.

  - (app)

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \multimap_e$$

    By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'.

In the first case, by the value lemma there are eleven cases to consider.

1. $t_1 = \lambda x.t_1'$. Here, there are two possibilities: either $t_2 = v$ for some value v or $t_2 \rightsquigarrow t_2'$ for some t_2'.

   In the first case, we can reduce by REDUCE using the reduction:

   $$\frac{}{(\lambda x.t)\, v \rightsquigarrow t[v/x]} \rightsquigarrow_\beta$$

   In the second case, since $\lambda x.t_1'$ is a value then by induction on the premise we can reduce by REDUCE using the reduction:

   $$\frac{t \rightsquigarrow t'}{v\, t \rightsquigarrow v\, t'} \rightsquigarrow_{\mathsf{prim}}$$

2. $t_1 = \mathsf{close}$, where $A = \mathbf{Chan}\,\mathbf{End}$. Therefore, we induct on the second argument. There are two cases.

   * t_2 is a value. Then, by the value lemma, $t_2 = c$ and the runtime typing derivation has two possible (consequential) specialisations:

     a)

     $$\frac{\dfrac{}{c : \mathbf{Chan}\,\mathbf{End} \vdash c : \mathbf{Chan}\,\mathbf{End}}\ \textsc{var} \qquad \dfrac{}{\emptyset \vdash \mathsf{close} : \mathbf{Chan}\,\mathbf{End} \multimap \mathsf{unit}}\ \textsc{close}}{c : \mathbf{Chan}\,\mathbf{End} \vdash \mathsf{close}\, c : \mathsf{unit}} \multimap_e$$

     Therefore, by Lemma 23 our overall succedent is of the form $\gamma_2', c : \mathbf{Chan}\,\mathbf{End} \vdash \mathit{Ctxt}\{\mathsf{close}\, c\} : A$.

     By inversion on the typing of channel configurations, we necessarily have:

     $$\frac{\gamma \vdash \mathcal{C}}{\gamma, c :^r \mathbf{Chan}\,\mathbf{End} \vdash \mathcal{C}, c = \langle\,\rangle}\ \textsc{empty}$$

     Therefore, there is a reduction as follows:

     $$\frac{}{(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{close}\, c\};\ \mathcal{C}, c^1 = \langle\,\rangle) \rightsquigarrow (\mathcal{P} \mid \mathit{Ctxt}\{\mathbf{unit}\};\ \mathcal{C})}\ \textsc{close}$$

     .

     b)

     $$\frac{\dfrac{}{c :^* \mathbf{Chan}\,\mathbf{End} \vdash c : \mathbf{Chan}\,\mathbf{End}}\ \textsc{varS} \qquad \dfrac{}{\emptyset \vdash \mathsf{close} : \mathbf{Chan}\,\mathbf{End} \multimap \mathsf{unit}}\ \textsc{close}}{c :^* \mathbf{Chan}\,\mathbf{End} \vdash \mathsf{close}\, c : \mathsf{unit}} \multimap_e$$

     Therefore, by Lemma 23 our overall succedent is of the form $\gamma_2', c :^* \mathbf{Chan}\,\mathbf{End} \vdash \mathit{Ctxt}\{\mathsf{close}\, c\} : A$.

There is then a reduction as follows:

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{close}\, c\};\ \mathcal{C}, c^* = \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{\mathbf{unit}\};\ \mathcal{C}, c^* = \vec{v})}\ \textsc{sharedClose}$$

.

* t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by Reduce using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{close}\, t_2 \rightsquigarrow \mathsf{close}\, t_2'}\ \rightsquigarrow_{\mathsf{prim}}$$

3. $t_1 = \mathsf{send}$, where $A = \mathbf{Chan}\,(\mathbf{Send}\ A'\ P) \multimap A' \multimap \mathbf{Chan}\, P$. Therefore, we induct on the second argument. There are two cases.

   * t_2 is a value. Then, by the value lemma, $t_2 = c$ and so $t_1\, t_2 = \mathsf{send}\, c$ which is also a value.
   * t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by Reduce using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{send}\, t_2 \rightsquigarrow \mathsf{send}\, t_2'}\ \rightsquigarrow_{\mathsf{prim}}$$

4. $t_1 = \mathsf{send}\, c$, where $A = A' \multimap \mathbf{Chan}\, P$. Therefore, we induct on the second argument. There are two cases.

   * $t_2 = x$ for some value x. The runtime typing derivation specialises to:

$$\frac{\dfrac{\cdots}{c :^r \mathbf{Chan}\,(\mathbf{Send}\ A'\ P) \vdash \mathsf{send}\, c : A' \multimap \mathbf{Chan}\, P}\ \multimap_e \quad \dfrac{\cdot}{\emptyset \vdash v : A'}\ \textsc{var}}{c :^r \mathbf{Chan}\,(\mathbf{Send}\ A'\ P) \vdash \mathsf{send}\, c\, x : \mathbf{Chan}\, P}\ \multimap_e$$

   Therefore, by Lemma 23 our overall succedent is of the form $\gamma_2', c :^r \mathbf{Chan}\,(\mathbf{Send}\ A'\ P) \vdash Ctxt\{\mathsf{send}\, c\, x\} : A$.

   By inversion, channel configuration typing must be of the form:

$$\frac{\begin{array}{c}\gamma_1 \vdash v : A \\ \gamma_1, \gamma_2, c :^r \mathbf{Chan}\, P \vdash \mathcal{C}, \bar{c} = \vec{v} : v\end{array}}{\gamma_1, \gamma_2, c :^r \mathbf{Chan}\,(\mathbf{Send}\ A\ P) \vdash \mathcal{C}, \bar{c} = \vec{v}}\ \textsc{send}$$

   with $\gamma = \gamma_1, \gamma_2$.

   In this case, there is then a reduction as follows:

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{send}\, c\, v\};\ \mathcal{C}, \bar{c} = \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{c\};\ \mathcal{C}, \bar{c} = \vec{v} : v)}\ \textsc{send}$$

* t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by REDUCE using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{send}\, c\, t_2 \rightsquigarrow \mathsf{send}\, c\, t_2'} \rightsquigarrow_{\mathsf{prim}}$$

5. $t_1 = \mathsf{receive}$, where $A = \mathbf{Chan}\,(\mathbf{Recv}\; A'\; P) \multimap A' \otimes \mathbf{Chan}\, P$. Therefore, we induct on the second argument. There are two cases.

   * t_2 is a value. Then, by the value lemma, $v_2 = c$ and the runtime typing derivation refines to:

$$\frac{\dfrac{\dots}{c :^r \mathbf{Chan}\,(\mathbf{Recv}\; A'\; P) \vdash c : \mathbf{Chan}\,(\mathbf{Recv}\; A'\; P)}\,\text{VAR} \qquad \dfrac{}{\emptyset \vdash \mathsf{receive} : \mathbf{Chan}\,(\mathbf{Recv}\; A\; P) \multimap (A \otimes \mathbf{Chan}\, P)}\,\text{RECV}}{c :^r \mathbf{Chan}\,(\mathbf{Recv}\; A'\; P) \vdash \mathsf{receive}\, c : A' \otimes \mathbf{Chan}\, P}\,\multimap_e$$

   Therefore, by Lemma 23 our overall succedent is of the form $\gamma_2', c :^r \mathbf{Chan}\,(\mathbf{Recv}\; A'\; P) \vdash Ctxt\{\mathsf{receive}\, c\} : A$.

   Now, since the channel configuration $\mathcal{C}$ must be buffered with respect to Δ, the channel c cannot be empty, and so it must contain at least one value v. Hence, there is only one possibility for the channel configuration typing:

$$\frac{\begin{array}{c}\gamma_1 \vdash v : A \\ \gamma_1, \gamma_2, c :^r \mathbf{Chan}\, P \vdash \mathcal{C}, c = \vec{v}\end{array}}{\gamma_1, \gamma_2, c :^r \mathbf{Chan}\,(\mathbf{Recv}\; A\; P) \vdash \mathcal{C}, c = v : \vec{v}}\,\text{RECV}$$

   with $\gamma = \gamma_1, \gamma_2$.

   Then there is a reduction as follows:

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{receive}\, c\};\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{(v, c)\};\ \mathcal{C}, c = \vec{v})}\,\text{RECV}$$

   * t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by REDUCE using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{receive}\, t_2 \rightsquigarrow \mathsf{receive}\, t_2'} \rightsquigarrow_{\mathsf{prim}}$$

6. $t_1 = \mathsf{forkLinear}$, where $A = (\mathbf{Chan}\, P \multimap \mathsf{unit}) \multimap \mathbf{Chan}\, \overline{P}$. Therefore, we induct on the second argument. There are two cases.

   * t_2 is a value. Then, by the value lemma, $t_2 = \lambda x.t_2'$ for some t_2' and we can construct the following runtime typing derivation:

$$\dfrac{\dfrac{x : \mathbf{Chan}\, P \vdash t_2' : \mathsf{unit}}{\emptyset \vdash \lambda x.t_2' : \mathbf{Chan}\, P \multimap \mathsf{unit}}\multimap_i \quad \dfrac{}{\emptyset \vdash \mathsf{forkLinear} : (\mathbf{Chan}\, P \multimap \mathsf{unit}) \multimap \mathbf{Chan}\, \overline{P}}\textsc{fork}}{\emptyset \vdash \mathsf{forkLinear}\, \lambda x.t_2' : \mathbf{Chan}\, \overline{P}}\multimap_e$$

Therefore, by Lemma 23 our overall succedent is of the form $\gamma_2' \vdash Ctxt\{\mathsf{forkLinear}\, \lambda x.t_2'\} : A$.

Then there is a reduction as follows:

$$\dfrac{\#c}{(\mathcal{P} \mid Ctxt\{\mathsf{forkLinear}\, (\lambda x.t)\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^\circ \mid Ctxt\{\overline{c}\};\ \mathcal{C}, \overline{c} = \langle\rangle, c = \langle\rangle)}\textsc{forkLinear}$$

where the channel configuration is typed $\gamma_2' = \mathcal{C}$ and we select a fresh name c for the channel being forked.

* t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by Reduce using the reduction:

$$\dfrac{t_2 \rightsquigarrow t_2'}{\mathsf{forkLinear}\, t_2 \rightsquigarrow \mathsf{forkLinear}\, t_2'}\rightsquigarrow_{\mathsf{prim}}$$

7. $t_1 = \mathsf{forkNonLinear}$, where $A = (\Box_r(\mathbf{Chan}\, P) \multimap \mathsf{unit}) \multimap \Box_r(\mathbf{Chan}\, \overline{P})$. Therefore, we induct on the second argument. There are two cases.

   * t_2 is a value. Then, by the value lemma, $t_2 = \lambda x.t_2'$ for some t_2' and we can construct the following runtime typing derivation:

$$\dfrac{\dfrac{x : \Box_r(\mathbf{Chan}\, P) \vdash t_2' : \mathsf{unit}}{\emptyset \vdash \lambda x.t_2' : \Box_r(\mathbf{Chan}\, P) \multimap \mathsf{unit}}\multimap_i \quad \textsc{forkNonLin}}{\emptyset \vdash \mathsf{forkNonLinear}\, \lambda x.t_2' : \Box_r(\mathbf{Chan}\, \overline{P})}\multimap_e \tag{d.1}$$

   Note that this derivation cannot include any instances of the Approx rule, because of the `ExactSemiring` constraint present in the type signature of `forkNonLinear`, and so the grade r must remain the same throughout the typing.

   Therefore, by Lemma 23 our overall succedent is of the form $\gamma_2' \vdash Ctxt\{\mathsf{forkNonLinear}\, \lambda x.t_2'\} : A$.

   Then there is a reduction as follows:

$$\dfrac{\#c}{\begin{array}{ll} & (\mathcal{P} \mid Ctxt\{\mathsf{forkNonLinear}\, (\lambda x.t)\};\ \mathcal{C}) \\ \rightsquigarrow & (\mathcal{P} \mid Ctxt\{[\overline{c}]\} \mid t[[c]/x]^\circ;\ \mathcal{C}, \overline{c}^* = \langle\rangle, c^* = \langle\rangle)\end{array}}\textsc{forkNonLinear}$$

   where the channel configuration is typed $\gamma_2' = \mathcal{C}$ and we select a fresh name c for the channel being forked.

* t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by REDUCE using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{forkNonLinear}\, t_2 \rightsquigarrow \mathsf{forkNonLinear}\, t_2'} \rightsquigarrow_{\mathsf{prim}}$$

8. $t_1 = \mathsf{forkReplicate}$, where $A = \square_{0..1}(\mathbf{Chan}\, P \multimap \mathsf{unit}) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})$. Therefore, we induct on the second argument. There are two cases.

   * t_2 is a value. Then, by the value lemma, $t_2 = [\lambda c.t_2']$ for some t_2' and $t_1\, t_2$ is $\mathsf{forkReplicate}\, [\lambda c.t_2']$ which is also a value.

   * t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by REDUCE using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{forkReplicate}\, t_2 \rightsquigarrow \mathsf{forkReplicate}\, t_2'} \rightsquigarrow_{\mathsf{prim}}$$

9. $t_1 = \mathsf{forkReplicate}\, c$, where $A = \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})$. Therefore, we induct on the second argument. There are two cases.

   * t_2 is a value. Then, by the value lemma, $t_2 = \lambda x.t_2'$ for some t_2' and the runtime typing derivation refines to:

$$\dfrac{\dfrac{\dfrac{\dfrac{x : \mathbf{Chan}\, P \vdash t_2' : \mathsf{unit}}{\emptyset \vdash \lambda x.t_2' : \mathbf{Chan}\, P \multimap \mathsf{unit}} \multimap_i}{\emptyset \vdash [\lambda x.t_2'] : \square_{0..n}(\mathbf{Chan}\, P \multimap \mathsf{unit})}\ \mathrm{PR}\ \textsc{forkRep}}{\emptyset \vdash \mathsf{forkReplicate}\, [\lambda x.t_2'] : \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\square_{0..1}(\mathbf{Chan}\, \overline{P}))} \multimap_e}{\emptyset \vdash \mathsf{forkReplicate}\, [\lambda x.t_2']\, n : \mathbf{Vec}\, n\, (\square_{0..1}(\mathbf{Chan}\, \overline{P}))} \multimap_e$$

   Therefore, by Lemma 23 our overall succedent is of the form $\gamma_2' \vdash Ctxt\{\mathsf{forkReplicate}\, [\lambda x.t_2']\, n\} : A$.

   Then there is a reduction as follows:

$$\begin{array}{rl} & \dfrac{\#c_1 \ldots \#c_n}{(\mathcal{P} \mid Ctxt\{\mathsf{forkReplicate}\, [\lambda x.t]\, n\};\ \mathcal{C})}\ \textsc{forkRep} \\ \rightsquigarrow & (\mathcal{P} \mid Ctxt\{\langle [\overline{c_1}], .., [\overline{c_n}] \rangle\} \mid \mathsf{Wch}\, (c_1 \mapsto \lambda x.t) \mid .. \mid \mathsf{Wch}\, (c_n \mapsto \lambda x.t) \\ & \quad ;\ \mathcal{C}, {c_1}^* = \langle\,\rangle, .., {c_n}^* = \langle\,\rangle, \overline{c_1}^* = \langle\,\rangle, .., \overline{c_n}^* = \langle\,\rangle) \end{array}$$

   where the channel configuration is typed $\gamma_2' = \mathcal{C}$ and we select fresh names c and $c_1 \ldots c_n$ for the channels being forked.

* t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by REDUCE using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{forkReplicate}\, c\, t_2 \rightsquigarrow \mathsf{forkReplicate}\, c\, t_2'} \rightsquigarrow_{\mathsf{prim}}$$

10. $t_1 = \mathsf{forkMulticast}$, where $A = (\mathbf{Chan}\,(\mathsf{Graded}_n P) \multimap \mathsf{unit}) \multimap \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})$. Therefore, we induct on the second argument. There are two cases.
    * t_2 is a value. Then, by the value lemma, $t_2 = \lambda c.t_2'$ for some t_2' and $t_1\, t_2$ is $\mathsf{forkMulticast}\, \lambda c.t_2'$ which is also a value.
    * t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by REDUCE using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{forkMulticast}\, t_2 \rightsquigarrow \mathsf{forkMulticast}\, t_2'} \rightsquigarrow_{\mathsf{prim}}$$

11. $t_1 = \mathsf{forkMulticast}\, c$, where $A = \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})$. Therefore, we induct on the second argument. There are two cases.
    * t_2 is a value. Then, by the value lemma, $t_2 = \lambda x.t_2'$ for some t_2' and the runtime typing derivation refines to:

$$\frac{\dfrac{\dfrac{x : \mathbf{Chan}\,(\mathsf{Graded}_r P) \vdash t_2' : \mathsf{unit}}{\emptyset \vdash \lambda x.t_2' : \mathbf{Chan}\,(\mathsf{Graded}_r P) \multimap \mathsf{unit}}\ \multimap_i}{\emptyset \vdash \mathsf{forkMulticast}\,(\lambda x.t_2') : \mathbf{N}\ n \multimap \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})}\ \multimap_e \qquad \textsc{forkMulti}}{\emptyset \vdash \mathsf{forkMulticast}\,(\lambda x.t_2')\, n : \mathbf{Vec}\, n\, (\mathbf{Chan}\, \overline{P})}\ \multimap_e$$

    Therefore, by Lemma 23 our overall succedent is of the form $\gamma_2' \vdash Ctxt\{\mathsf{forkMulticast}\, \lambda x.t_2'\, n\} : A$.

    Then there is a reduction as follows:

$$\frac{\#c \qquad \#c_1 \ldots \#c_n}{\begin{array}{ll} & (\mathcal{P} \mid Ctxt\{\mathsf{forkMulticast}\,(\lambda x.t)\, n\};\ \mathcal{C}) \\ \rightsquigarrow & (\mathcal{P} \mid Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\} \mid t[c/x]^\circ \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n) \\ & \quad ;\ \mathcal{C}, c = \langle\,\rangle, \overline{c} = \langle\,\rangle, c_1 = \langle\,\rangle, .., c_n = \langle\,\rangle, \overline{c_1} = \langle\,\rangle, .., \overline{c_n} = \langle\,\rangle) \end{array}}\ \textsc{forkMulticast}$$

    where the channel configuration is typed $\gamma_2' = \mathcal{C}$ and we select fresh names c and $c_1 \ldots c_n$ for the channels being forked.
    * t_2 is not a value, and has a reduction $t_2 \rightsquigarrow t_2'$ for some t_2'. Then we can reduce by REDUCE using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{forkMulticast}\, c\, t_2 \rightsquigarrow \mathsf{forkMulticast}\, c\, t_2'} \rightsquigarrow_{\mathsf{prim}}$$

In the second case, by induction on the premise we can reduce by REDUCE using the reduction:

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_2}\ \rightsquigarrow_{\text{APP}}$$

– (pr)

$$\frac{\Gamma \vdash t : A}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textbf{PR}$$

There are two cases: either $t = v$ for some value v, or $t \rightsquigarrow t'$ for some t'.

In the first case, $[v]$ is a value.

In the second case, we can reduce by REDUCE using the reduction:

$$\frac{t \rightsquigarrow t'}{[t] \rightsquigarrow [t']}\ \rightsquigarrow \Box\text{reduce}$$

– (elim)

$$\frac{\begin{array}{c}\Gamma_1 \vdash t_1 : \Box_r A \\ \Gamma_2, x : [A]_r \vdash t_2 : B\end{array}}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ [x]\ = t_1\ \textbf{in}\ t_2 : B}\ \textbf{ELIM}$$

By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'.

In the first case, by the value lemma $t_1 = [v']$ for some value v'. This refines the typing as follows:

$$\frac{\dfrac{\emptyset \vdash v' : A}{\emptyset \vdash [v'] : \Box_r A}\ \textbf{PR} \qquad x : [A]_r \vdash t_2 : B}{\emptyset \vdash \textbf{let}\ [x]\ = [v']\ \textbf{in}\ t_2 : B}\ \textbf{ELIM}$$

Therefore, we can reduce by REDUCE using the reduction:

$$\frac{}{\textbf{let}\ [x]\ = [v]\ \textbf{in}\ t \rightsquigarrow t[v/x]}\ \rightsquigarrow \Box\beta$$

In the second case, by induction on the premise we can reduce by REDUCE using the reduction:

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let}\ [x]\ = t_1\ \textbf{in}\ t_2 \rightsquigarrow \textbf{let}\ [x]\ = t_1'\ \textbf{in}\ t_2}\ \rightsquigarrow_{\text{LET}\Box}$$

– (der)

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textbf{DER}$$

Goal achieved immediately by induction on the premise.

– (weak)

$$\frac{\Gamma \vdash t : B}{\Gamma, 0 \cdot \Gamma' \vdash t : B}\ \textsc{weak}$$

Goal achieved immediately by induction on the premise.

– (approx)

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx}$$

Goal achieved immediately by induction on the premise.

– (tensor)

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : A \otimes B}\ \otimes_i$$

There are three cases: either $t_1 = v_1$ and $t_2 = v_2$ for some pair of values (v_1, v_2), $t_1 = v$ for some value v but $t_2 \rightsquigarrow t_2'$ for some t_2', or $t_1 \rightsquigarrow t_1'$ for some t_1'.

In the first case, (v_1, v_2) is a value.

In the second case, we can reduce by Reduce using the reduction:

$$\frac{t_2 \rightsquigarrow t_2'}{(v, t_2) \rightsquigarrow (v, t_2')}\ \rightsquigarrow_{\otimes \mathsf{reduceR}}$$

In the third case, we can reduce by Reduce using the reduction:

$$\frac{t_1 \rightsquigarrow t_1'}{(t_1, t_2) \rightsquigarrow (t_1', t_2)}\ \rightsquigarrow_{\otimes \mathsf{reduceL}}$$

– (pairElim)

$$\frac{\Gamma_1 \vdash t_1 : A \otimes A' \qquad \Gamma_2, x : A, y : A' \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ (x, y) = t_1\ \mathbf{in}\ t_2 : B}\ \otimes_e$$

By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'.

In the first case, by the value lemma $t_1 = (v_1, v_2)$ for some pair of values (v_1, v_2). This refines the typing as follows:

$$\frac{\dfrac{\emptyset \vdash v_1 : A \quad \emptyset \vdash v_2 : A'}{\emptyset \vdash (v_1, v_2) : A \otimes A'}\ \otimes_i \qquad x : A, y : A' \vdash t_2 : B}{\emptyset \vdash \mathbf{let}\ (x, y) = (v_1, v_2)\ \mathbf{in}\ t_2 : B}\ \otimes_e$$

Therefore, we can reduce by Reduce using the reduction:

$$\frac{}{\mathbf{let}\ (x, y) = (v_1, v_2)\ \mathbf{in}\ t_3 \rightsquigarrow t_3[v_1/x][v_2/y]}\ \rightsquigarrow_{\otimes\beta}$$

In the second case, by induction on the premise we can reduce by REDUCE using the reduction:

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let } (x,y) = t_1 \textbf{ in } t_2 \rightsquigarrow \textbf{let } (x,y) = t_1' \textbf{ in } t_2} \rightsquigarrow_{\text{LET}\otimes}$$

– (unit)

$$\frac{}{\emptyset \vdash \textbf{unit} : \mathsf{unit}} \text{UNIT}_i$$

A value.

– (unitElim)

$$\frac{\Gamma_1 \vdash t_1 : \mathsf{unit} \quad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let unit} = t_1 \textbf{ in } t_2 : B} \text{UNIT}_e$$

By induction on the first premise, there are two possibilities: either $t_1 = v$ for some value v or $t_1 \rightsquigarrow t_1'$ for some t_1'.

In the first case, by the value lemma $t_1 = \textbf{unit}$ and so we can reduce by REDUCE using the reduction:

$$\frac{}{\textbf{let unit} = \textbf{unit in } t \rightsquigarrow t} \rightsquigarrow_{\beta\mathsf{unit}}$$

In the second case, by induction on the premise we can reduce by REDUCE using the reduction:

$$\frac{t_1 \rightsquigarrow t_1'}{\textbf{let unit} = t_1 \textbf{ in } t_2 \rightsquigarrow \textbf{let unit} = t_1' \textbf{ in } t_2} \rightsquigarrow_{\text{LET}\mathsf{unit}}$$

□

A process configuration $\mathcal{P}$ contains all values, defined by the predicate $\mathsf{values}(\mathcal{P})$, if every process is either a value v or is a forwarder or watcher process.

Theorem 12 (Progress global config)**.**

$$\begin{aligned} &\gamma \vdash \mathcal{P} : A \ \wedge\ \gamma \vdash \mathcal{C} \ \wedge\ \mathsf{buffered}(\mathcal{C}, \Delta) \\ \Longrightarrow\ &\mathsf{values}(\mathcal{P}) \vee \exists \mathcal{C}', \mathcal{P}'.(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}') \end{aligned}$$

Proof. By induction on the two judgments forming the assumptions.

First, we case split on the typing of process configurations: $\gamma \vdash \mathcal{P} : A$.

- (empty)

$$\frac{}{\emptyset \vdash \cdot : \bot} \text{EMPTY}$$

This case is trivial, as $\mathsf{values}(\mathcal{P})$ holds by default for an empty process configuration.

- (cons)

$$\frac{\Gamma_1, \gamma_1 \vdash \mathcal{P} : A \qquad \Gamma_2, \gamma_2 \vdash t : B}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2 \vdash \mathcal{P} \mid t^{\mathcal{R}} : A} \ \textsc{cons}\mathcal{R}$$

Here, we can decompose $\mathcal{P}$ into $\mathcal{P}' \mid t$. First, focus on the particular term t and apply Theorem 1.

Then, either t is a value or there exists some possible reduction $(\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t';\ \mathcal{C}')$.

If we can make such a reduction, then the theorem is satisfied. If not, then apply induction to the smaller process configuration $\mathcal{P}'$. We find that either $\mathsf{values}(\mathcal{P}')$ holds or there exists a reduction $(\mathcal{P}';\ \mathcal{C}) \rightsquigarrow (\mathcal{P}'';\ \mathcal{C}')$.

If $\mathsf{values}(\mathcal{P}')$, then since t is also a value we have $\mathsf{values}(\mathcal{P})$ and we are done.

Otherwise, we apply the following rule to find a reduction satisfying the theorem:

$$\frac{(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')}{(\mathcal{P} \mid t;\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid t;\ \mathcal{C}')} \ \textsc{par}$$

- (watch)

$$\frac{\Gamma_1, \gamma_1 \vdash \mathcal{P} : A \qquad \Gamma_2, x : \mathbf{Chan}\ (\mathbf{Recv}\ A'\ P), \gamma_2 \vdash t : \mathsf{unit}}{\Gamma_1 + (0..1 \cdot \Gamma_2), \gamma_1 + \gamma_2, c : \mathbf{Chan}\ (\mathbf{Recv}\ A'\ P) \vdash \mathcal{P} \mid \mathsf{Wch}\ (c \mapsto \lambda x.t) : A} \ \textsc{watch}$$

By induction on the first assumption, we have either that $\mathsf{values}(\mathcal{P})$ or that there exists some reduction $(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')$.

If there is such a reduction, we can make a global reduction as follows:

$$\frac{(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')}{(\mathcal{P} \mid \mathsf{Wch}\ (c \mapsto \lambda x.t);\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid \mathsf{Wch}\ (c \mapsto \lambda x.t);\ \mathcal{C}')} \ \textsc{par}$$

Otherwise, we must reduce on the watcher process itself. Note that due to the session typing of the channel in the runtime context, and the condition requiring that $\mathcal{C}$ is buffered, the channel configuration must be typed as follows:

$$\frac{\gamma_1 + \gamma_2 \vdash \mathcal{C} \qquad \gamma_1 + \gamma_2 \vdash v : A'}{\gamma_1 + \gamma_2, c : (\mathbf{Recv}\ A'\ P) \vdash \mathcal{C}, c = v : \vec{v}} \ \textsc{recv} \tag{d.2}$$

Hence, we can construct the following global reduction:

$$\frac{}{(\mathcal{P} \mid \mathsf{Wch}\ (c \mapsto \lambda x.t);\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^{\circ};\ \mathcal{C}, c = v : \vec{v})} \ \textsc{watcher}$$

- (forward)

$$\frac{\mathsf{Sends}(P) \qquad \Gamma, \gamma \vdash \mathcal{P} : A}{\Gamma, \gamma, c : \mathbf{Chan}\,(\mathsf{Graded}_n \overline{P}), c_i : \mathbf{Chan}\,P \vdash \mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n) : A}\ \textsc{fwd}$$

By induction on the first assumption, we have either that $\mathsf{values}(\mathcal{P})$ or that there exists some reduction $(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')$.

If there is such a reduction, we can make a global reduction as follows:

$$\frac{(\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}')}{(\mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n);\ \mathcal{C}) \rightsquigarrow (\mathcal{P}' \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n);\ \mathcal{C}')}\ \textsc{par}$$

Otherwise, we must reduce on the forwarder process itself. Note that due to the session typing of the channel in the runtime context, and the the condition requiring that $\mathcal{C}$ is buffered, then the channel configuration must be typed as follows:

$$\frac{\dfrac{\emptyset \vdash v : A'}{\emptyset \vdash [v] : \Box_n A'}\ \textsc{pr} \qquad \dfrac{\gamma, c_1 : \mathbf{Chan}\,P, .., c_n : \mathbf{Chan}\,P, c : \mathbf{Chan}\,\overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v, c = \vec{v}}{\gamma, c_1 : \mathbf{Chan}\,(\mathbf{Send}\ A\ P), .., c_n : \mathbf{Chan}\,(\mathbf{Send}\ A\ P), c : \mathbf{Chan}\,\overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n, c = \vec{v}}\ \textsc{send}^n}{\gamma, c_1 : \mathbf{Chan}\,(\mathbf{Send}\ A\ P), .., c_n : \mathbf{Chan}\,(\mathbf{Send}\ A\ P), c : \mathbf{Chan}\,(\mathbf{Recv}\ (\Box_n A')\ \overline{P}) \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n, c = [v] : \vec{v}}\ \textsc{recv} \tag{d.3}$$

Hence, we can construct the following global reduction:

$$\frac{}{(\mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n);\ \mathcal{C}, c = [v] : \vec{v}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n) \rightsquigarrow (\mathcal{P} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n);\ \mathcal{C}, c = \vec{v}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v)}\ \textsc{forwarder}$$

□

D.3.2 Preservation proof

Theorem 13 (Preservation)**.**

$$\begin{aligned} &\Gamma, \gamma \vdash \mathcal{P} : A \ \wedge\ \gamma \vdash \mathcal{C} \ \wedge\ (\mathcal{P};\ \mathcal{C}) \rightsquigarrow (\mathcal{P}';\ \mathcal{C}') \\ \implies &\exists \gamma'.\ \Gamma, \gamma' \vdash \mathcal{P}' : A \ \wedge\ \gamma' \vdash \mathcal{C}' \end{aligned}$$

Proof. By induction on the three judgments forming the assumptions here.

We first case split on the typing of process configurations: $\Gamma, \gamma \vdash \mathcal{P} : A$.

- (empty)

$$\frac{}{\emptyset \vdash \cdot : \bot}\ \textsc{empty}$$

There is no reduction on the empty process configuration so this case is trivially true since the antecedent is false.

- (cons) We consider an abstraction over the two possibilities for extending a process configuration with a term which is either main or subordinate process:

$$\frac{\Gamma_1, \gamma_1 \vdash t : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2 \vdash \mathcal{P} \mid t^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R}$$

where $\mathcal{R} = \bullet$ in which case $B = A$, otherwise $B = \mathsf{unit}$.

We then case on the reduction of process configurations consider the role $\mathcal{R}$ where pertinent.

– (send)

$$\frac{}{(\mathcal{P} \mid \mathcal{C}txt\{\mathsf{send}\, c\, v\};\ \mathcal{C}, \bar{c} = \vec{v}) \rightsquigarrow (\mathcal{P} \mid \mathcal{C}txt\{c\};\ \mathcal{C}, \bar{c} = \vec{v} : v)}\ \textsc{send}$$

with process configuration typing:

$$\frac{\begin{array}{c}\Gamma_1, \gamma_1, c :^r \mathbf{Chan}\,(\mathbf{Send}\ A\ P) \vdash \mathcal{C}txt\{\mathsf{send}\, c\, v\} : B \\ \Gamma_2, \gamma_2 \vdash \mathcal{P} : A\end{array}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c :^r \mathbf{Chan}\,(\mathbf{Send}\ A\ P) \vdash \mathcal{P} \mid \mathcal{C}txt\{\mathsf{send}\, c\, v\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R} \tag{d.4}$$

and channel configuration typing:

$$\frac{\gamma_3 \vdash v : A \qquad \gamma_3, \gamma_4, c :^r \mathbf{Chan}\, P \vdash \mathcal{C}, \bar{c} = \vec{v} : v}{\gamma_3, \gamma_4, c :^r \mathbf{Chan}\,(\mathbf{Send}\ A\ P) \vdash \mathcal{C}, \bar{c} = \vec{v}}\ \textsc{send} \tag{d.5}$$

where $(\gamma_3, \gamma_4) = \gamma_1 + \gamma_2$.

We can then construct the consequent channel configuration typing given by the premise of (d.5):

$$\gamma_1 + \gamma_2, c :^r \mathbf{Chan}\, P \vdash \mathcal{C}, \bar{c} = v : \vec{v}$$

From typing $\Gamma_1, \gamma_1 \vdash \mathcal{C}txt\{\mathsf{send}\, c\, v\} : B$, and by lemma (22) we have two cases:

1. Typing $\Gamma_1', \gamma_1' \vdash \mathsf{send}\, c\, v : A''$ and $\Gamma_2', y : A'', \gamma_2' \vdash \mathcal{C}txt\{y\} : B$ with $\Gamma_1 = \Gamma_1' + \Gamma_2'$ and $\gamma_1 = \gamma_1' + \gamma_2'$, which by inversion must therefore be of the form:

$$\frac{\dfrac{\ldots \qquad \dfrac{}{c : \mathbf{Chan}\,(\mathbf{Send}\ A'\ P) \vdash c : \mathbf{Chan}\,(\mathbf{Send}\ A'\ P)}\ \textsc{var}}{\emptyset, c : \mathbf{Chan}\,(\mathbf{Send}\ A'\ P) \vdash \mathsf{send}\ c : A' \multimap \mathbf{Chan}\, P}\ \multimap_e \qquad \emptyset \vdash v : A'}{\emptyset, c : \mathbf{Chan}\,(\mathbf{Send}\ A'\ P) \vdash \mathsf{send}\, c\, v : \mathbf{Chan}\, P}\ \multimap_e$$

i.e., where $A'' = \mathbf{Chan}\, P$ and $\Gamma_1' = \cdot$ and $\gamma_1' = c : \mathbf{Chan}\,(\mathbf{Send}\ A'\ P)$.

Therefore, for the goal of this theorem we construct the typing:

$$\frac{}{\emptyset, c : \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\ \textsc{var}$$

By lemma 23 then $\Gamma_2', \gamma_2', c : \mathbf{Chan}\, P \vdash Ctxt\{c\} : B$ which is equal to $\Gamma_1, \gamma_2', c : \mathbf{Chan}\, P \vdash Ctxt\{c\} : B$.

We then build the process configuration type:

$$\frac{\dfrac{\Gamma_2', \gamma_2', c : \mathbf{Chan}\, P \vdash Ctxt\{c\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_2' + \Gamma_2, \gamma_2' + \gamma_2, c : \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{c\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c : \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{c\}^{\mathcal{R}} : A}\ \equiv \tag{d.6}$$

where the equality step follows by $\gamma_1 = \gamma_1' + \gamma_2' = \gamma_2'$ and $\Gamma_1 = \Gamma_1' + \Gamma_2' = \Gamma_2'$.

2. Typing $[\Gamma_1', \gamma_1'] \vdash \mathsf{send}\, c\, v : A''$ and $\Gamma_2', y : [A'']_r, \gamma_2' \vdash Ctxt\{y\} : B$ with $\Gamma_1 = r \cdot \Gamma_1' + \Gamma_2'$ and $\gamma_1 = \gamma_1' + \gamma_2'$, which by inversion must therefore be of the form:

$$\frac{\dfrac{\dfrac{}{c :^* \mathbf{Chan}\,(\mathbf{Send}\, A'\, P) \vdash c : \mathbf{Chan}\,(\mathbf{Send}\, A'\, P)}\ \textsc{varChanS}}{c :^* \mathbf{Chan}\,(\mathbf{Send}\, A'\, P) \vdash \mathsf{send}\ c : A' \multimap \mathbf{Chan}\, P}\ \multimap_e \qquad \emptyset \vdash v : A'}{c :^* \mathbf{Chan}\,(\mathbf{Send}\, A'\, P) \vdash \mathsf{send}\, c\, v : \mathbf{Chan}\, P}\ \multimap_e$$

i.e., where $A'' = \mathbf{Chan}\, P$ and $\Gamma_1' = \cdot$ and $\gamma_1' = c :^* \mathbf{Chan}\,(\mathbf{Send}\, A'\, P)$.

Therefore, for the goal of this theorem we construct the typing:

$$\frac{}{c :^* \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\ \textsc{varChanS}$$

By lemma 24 then $\Gamma_2', \gamma_2', c :^* \mathbf{Chan}\, P \vdash Ctxt\{c\} : B$ which is equal to $\Gamma_1, \gamma_2', c :^* \mathbf{Chan}\, P \vdash Ctxt\{c\} : B$ (since $\Gamma_1 = r \cdot \Gamma_1' + \Gamma_2' = r \cdot \emptyset + \Gamma_2' = \Gamma_2'$).

We then build the process configuration type:

$$\frac{\dfrac{\Gamma_2', \gamma_2', c :^* \mathbf{Chan}\, P \vdash Ctxt\{c\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_2' + \Gamma_2, \gamma_2' + \gamma_2, c :^* \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{c\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c :^* \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{c\}^{\mathcal{R}} : A}\ \equiv \tag{d.7}$$

where the equality step follows by $\gamma_1 = \gamma_1' + \gamma_2' = \gamma_2'$ and $\Gamma_1 = r \cdot \Gamma_1' + \Gamma_2' = \Gamma_2'$.

– (recv)

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{receive}\, c\};\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{(v, c)\};\ \mathcal{C}, c = \vec{v})}\ \textsc{recv}$$

with process configuration typing:

$$\frac{\Gamma_1, \gamma_1, c :^r \mathbf{Chan}\,(\mathbf{Recv}\, A\, P) \vdash Ctxt\{\mathsf{receive}\, c\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c :^r \mathbf{Chan}\,(\mathbf{Recv}\, A\, P) \vdash \mathcal{P} \mid Ctxt\{\mathsf{receive}\, c\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R} \tag{d.8}$$

with channel configuration typing:

$$\frac{\gamma_3 \vdash v : A \qquad \gamma_3, \gamma_4, c :^r \mathbf{Chan}\, P \vdash \mathcal{C}, c = \vec{v}}{\gamma_3, \gamma_4, c :^r \mathbf{Chan}\,(\mathbf{Recv}\ A\ P) \vdash \mathcal{C}, c = v : \vec{v}}\ \textsc{recv} \tag{d.9}$$

where $(\gamma_3, \gamma_4) = \gamma_1 + \gamma_2$. (note that EMPTY cannot be used in the root of this derivation as the reduction constraints the channel to start non-empty).

We then construct the consequent channel configuration typing given by the premise of (d.9):

$$\gamma_1 + \gamma_2, c :^r \mathbf{Chan}\, P \vdash \mathcal{C}, c = \vec{v}$$

From typing $\Gamma_1, \gamma_1 \vdash Ctxt\{\mathsf{receive}\, c\} : B$, and by lemma 22 then there are two possible cases:

1. Typing $\Gamma_1', \gamma_1' \vdash \mathsf{receive}\, c : A''$ and $\Gamma_2', y : A'', \gamma_2' \vdash Ctxt\{y\} : B$ with $\Gamma_1 = \Gamma_1' + \Gamma_2'$ and $\gamma_1 = \gamma_1' + \gamma_2'$, which by inversion must therefore be of the form:

$$\frac{\dfrac{}{\emptyset, c : \mathbf{Chan}\,(\mathbf{Recv}\ A\ P) \vdash c : \mathbf{Chan}\,(\mathbf{Recv}\ A\ P)}\ \textsc{var}}{\emptyset, c : \mathbf{Chan}\,(\mathbf{Recv}\ A\ P) \vdash \mathsf{receive}\, c : (A \otimes \mathbf{Chan}\, P)}\ \multimap_e$$

   where $A'' = (A \otimes \mathbf{Chan}\, P)$ and $\Gamma_1' = \cdot$ and $\gamma_1' = c : \mathbf{Chan}\,(\mathbf{Recv}\ A\ P)$.

   Therefore, for the goal of this theorem we construct the typing:

$$\frac{\dfrac{}{c : \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\ \textsc{var} \qquad \emptyset \vdash v : A' \text{ (by (d.9))}}{c : \mathbf{Chan}\, P \vdash (v, c) : (A \otimes \mathbf{Chan}\, P)}\ \otimes_i$$

   By Lemma 23 then $\Gamma_2', \gamma_2', c : \mathbf{Chan}\, P \vdash Ctxt\{(v, c)\} : B$ which is equal to $\Gamma_1, \gamma_2', c : \mathbf{Chan}\, P \vdash Ctxt\{(v, c)\} : B$

   We then construct the preserved process configuration typing:

$$\frac{\dfrac{\Gamma_2', \gamma_2', c : \mathbf{Chan}\, P \vdash Ctxt\{(v, c)\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_2' + \Gamma_2, \gamma_2' + \gamma_2, c : \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{(v, c)\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c : \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{(v, c)\}^{\mathcal{R}} : A}\ \equiv \tag{d.10}$$

   where the equality step follows by $\gamma_1 = \gamma_1' + \gamma_2' = \gamma_2'$ and $\Gamma_1 = \Gamma_1' + \Gamma_2' = \Gamma_2'$.

2. Typing $[\Gamma_1', \gamma_1'] \vdash \mathsf{receive}\, c : A''$ and $\Gamma_2', y : [A'']_r, \gamma_2' \vdash Ctxt\{y\} : B$ with $\Gamma_1 = r \cdot \Gamma_1' + \Gamma_2'$ and $\gamma_1 = \gamma_1' + \gamma_2'$, which by inversion must therefore be of the form:

$$\frac{\dfrac{}{c :^* \mathbf{Chan}\,(\mathbf{Recv}\ A\ P) \vdash c : \mathbf{Chan}\,(\mathbf{Recv}\ A\ P)}\ \textsc{varChanS}}{c :^* \mathbf{Chan}\,(\mathbf{Recv}\ A\ P) \vdash \mathsf{receive}\, c : (A \otimes \mathbf{Chan}\, P)}\ \multimap_e$$

where $A'' = (A \otimes \mathbf{Chan}\, P)$ and $\Gamma_1' = \cdot$ and $\gamma_1' = c :^* \mathbf{Chan}\,(\mathbf{Recv}\; A\; P)$.

Therefore, for the goal of this theorem we construct the typing:

$$\dfrac{\dfrac{}{c :^* \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\ \textsc{varChanS} \qquad \emptyset \vdash v : A' \text{ (by (d.9))}}{c :^* \mathbf{Chan}\, P \vdash (v, c) : (A \otimes \mathbf{Chan}\, P)}\ \otimes_i$$

By Lemma 24 then $\Gamma_2', \gamma_2', c :^* \mathbf{Chan}\, P \vdash Ctxt\{(v,c)\} : B$ which is equal to $\Gamma_1, \gamma_2', c :^* \mathbf{Chan}\, P \vdash Ctxt\{(v,c)\} : B$ (since $\Gamma_1 = r \cdot \Gamma_1' + \Gamma_2' = r \cdot \emptyset + \Gamma_2' = \Gamma_2'$).

We then construct the preserved process configuration typing:

$$\dfrac{\dfrac{\Gamma_2', \gamma_2', c :^* \mathbf{Chan}\, P \vdash Ctxt\{(v,c)\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_2' + \Gamma_2, \gamma_2' + \gamma_2, c :^* \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{(v,c)\}^{\mathcal{R}} : A}\ \textsc{consR}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c :^* \mathbf{Chan}\, P \vdash \mathcal{P} \mid Ctxt\{(v,c)\}^{\mathcal{R}} : A} \equiv \tag{d.11}$$

where the equality step follows by $\gamma_1 = \gamma_1' + \gamma_2' = \gamma_2'$ and $\Gamma_1 = r \cdot \Gamma_1' + \Gamma_2' = \Gamma_2'$.

– (close)

$$\dfrac{}{(\mathcal{P} \mid Ctxt\{\mathsf{close}\, c\};\ \mathcal{C}, c^1 = \langle\,\rangle) \rightsquigarrow (\mathcal{P} \mid Ctxt\{\mathbf{unit}\};\ \mathcal{C})}\ \textsc{close}$$

with process configuration typing:

$$\dfrac{\Gamma_1, \gamma_1, c : \mathbf{Chan}\, \mathbf{End} \vdash Ctxt\{\mathsf{close}\, c\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c : \mathbf{Chan}\, \mathbf{End} \vdash \mathcal{P} \mid Ctxt\{\mathsf{close}\, c\}^{\mathcal{R}} : A}\ \textsc{consR} \tag{d.12}$$

(i.e., $c \notin \mathsf{dom}(\gamma_2)$) with only one potential channel configuration typing:

$$\dfrac{\gamma_1 + \gamma_2 \vdash \mathcal{C}}{\gamma_1 + \gamma_2, c : \mathbf{Chan}\, \mathbf{End} \vdash \mathcal{C}, c^1 = \langle\,\rangle}\ \textsc{empty} \tag{d.13}$$

The goal channel configuration $\gamma_1 + \gamma_2 \vdash \mathcal{C}$ is then given by the premise of (d.13).

From typing $\Gamma_1, \gamma_1 \vdash Ctxt\{\mathsf{close}\, c\} : B$, and by lemma 22, and since we know here that c is a linear channel, then the typing factorises to the typing $\Gamma_1', \gamma_1' \vdash \mathsf{close}\, c : A''$ and $\Gamma_2', y : A'', \gamma_2' \vdash Ctxt\{y\} : B$ with $\Gamma_1 = \Gamma_1' + \Gamma_2'$ and $\gamma_1 = \gamma_1' + \gamma_2'$, which by inversion must therefore be of the form:

$$\dfrac{c : \mathbf{Chan}\, \mathbf{End} \vdash c : \mathbf{Chan}\, \mathbf{End}}{c : \mathbf{Chan}\, \mathbf{End} \vdash \mathsf{close}\, c : \mathsf{unit}}\ \multimap_e$$

where $A'' = \mathsf{unit}$ and $\Gamma_1' = \cdot$ and $\gamma_1' = c : \mathbf{Chan}\, \mathbf{End}$.

Therefore, for the goal of this theorem we construct the typing:

$$\frac{}{\emptyset \vdash \mathbf{unit} : \mathsf{unit}}\ \textsc{unit}_i$$

By Lemma 23 then $\Gamma_2', \gamma_2' \vdash Ctxt\{\mathbf{unit}\} : B$.

We then construct the preserved process configuration typing:

$$\frac{\dfrac{\Gamma_2', \gamma_2' \vdash Ctxt\{\mathbf{unit}\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_2' + \Gamma_2, \gamma_2' + \gamma_2 \vdash \mathcal{P} \mid Ctxt\{\mathbf{unit}\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2 \vdash \mathcal{P} \mid Ctxt\{\mathbf{unit}\}^{\mathcal{R}} : A}\ \equiv \tag{d.14}$$

where the equality step follows by $\gamma_1 = \gamma_1' + \gamma_2' = \gamma_2'$ and $\Gamma_1 = \Gamma_1' + \Gamma_2' = \Gamma_2'$.

– (sharedClose)

$$\frac{}{(\mathcal{P} \mid Ctxt\{\mathsf{close}\, c\};\ \mathcal{C}, c^* = \vec{v}) \rightsquigarrow (\mathcal{P} \mid Ctxt\{\mathbf{unit}\};\ \mathcal{C}, c^* = \vec{v})}\ \textsc{sharedClose}$$

with process configuration typing:

$$\frac{\Gamma_1, \gamma_1, c :^* \mathbf{Chan}\ \mathbf{End} \vdash Ctxt\{\mathsf{close}\, c\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c :^* \mathbf{Chan}\ \mathbf{End} \vdash \mathcal{P} \mid Ctxt\{\mathsf{close}\, c\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R} \tag{d.15}$$

and channel configuration typing $\gamma_1 + \gamma_2, c :^* \mathbf{Chan}\ \mathbf{End} \vdash \mathcal{C}, c^* = \vec{v}$.

Since the reduction semantics preserves the channel configuration, this channel configuration typing then also provides the goal configuration typing as-is.

From typing $\Gamma_1, \gamma_1 \vdash Ctxt\{\mathsf{close}\, c\} : B$, and by lemma 22, and since we know here that c is a non-linear channel, then the typing factorises to the typing $[\Gamma_1', \gamma_1'] \vdash \mathsf{close}\, c : A''$ and $\Gamma_2', y : [A'']_r, \gamma_2' \vdash Ctxt\{y\} : B$ with $\Gamma_1 = r \cdot \Gamma_1' + \Gamma_2'$ and $\gamma_1 = \gamma_1' + \gamma_2'$, which by inversion must therefore be of the form:

$$\frac{c :^* \mathbf{Chan}\ \mathbf{End} \vdash c : \mathbf{Chan}\ \mathbf{End}}{c :^* \mathbf{Chan}\ \mathbf{End} \vdash \mathsf{close}\, c : \mathsf{unit}}\ \multimap_e$$

where $A'' = \mathsf{unit}$ and $\Gamma_1' = \cdot$ and $\gamma_1' = c :^* \mathbf{Chan}\ \mathbf{End}$.

Therefore, for the goal of this theorem we construct the typing:

$$\frac{\dfrac{}{\emptyset \vdash \mathbf{unit} : \mathsf{unit}}\ \textsc{unit}_i}{c :^* \mathbf{Chan}\ \mathbf{End} \vdash \mathbf{unit} : \mathsf{unit}}\ \textsc{weakChanS}$$

By Lemma 24 then $\Gamma_2', \gamma_2', c :^* \mathbf{Chan}\ \mathbf{End} \vdash Ctxt\{\mathbf{unit}\} : B$.

We then construct the preserved process configuration typing:

$$\frac{\dfrac{\Gamma_2', \gamma_2', c :^* \mathbf{Chan}\,\mathbf{End} \vdash Ctxt\{\mathbf{unit}\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_2' + \Gamma_2, \gamma_2' + \gamma_2, c :^* \mathbf{Chan}\,\mathbf{End} \vdash \mathcal{P} \mid Ctxt\{\mathbf{unit}\}^{\mathcal{R}} : A}\,\text{CONS}\mathcal{R}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c :^* \mathbf{Chan}\,\mathbf{End} \vdash \mathcal{P} \mid Ctxt\{\mathbf{unit}\}^{\mathcal{R}} : A} \equiv \tag{d.16}$$

where the equality step follows by $\gamma_1 = \gamma_1' + \gamma_2' = \gamma_2'$ and $\Gamma_1 = r \cdot \Gamma_1' + \Gamma_2' = \Gamma_2'$.

– (fork)

$$\frac{\#c}{(\mathcal{P} \mid Ctxt\{\mathsf{forkLinear}\,(\lambda x.t)\};\ \mathcal{C}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^\circ \mid Ctxt\{\overline{c}\};\ \mathcal{C}, \overline{c} = \langle\rangle, c = \langle\rangle)}\,\text{FORKLINEAR}$$

with process configuration typing:

$$\frac{\Gamma_1, \gamma_1 \vdash Ctxt\{\mathsf{forkLinear}\,(\lambda x.t)\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2 \vdash \mathcal{P} \mid Ctxt\{\mathsf{forkLinear}\,(\lambda x.t)\}^{\mathcal{R}} : A}\,\text{CONS}\mathcal{R} \tag{d.17}$$

with channel configuration typing:

$$\gamma_1 + \gamma_2 \vdash \mathcal{C} \tag{d.18}$$

From typing $\Gamma_1, \gamma_1 \vdash Ctxt\{\mathsf{forkLinear}\,(\lambda x.t)\} : B$, and by lemma 22 there are two possibilities:

1. (second disjunct) (with $Ctxt\{\mathsf{forkLinear}\,(\lambda x.t)\}$ factorising into a graded binding) Gives $\neg\mathsf{resourceAllocator}(\mathsf{forkLinear}\,(\lambda x.t))$ which is a contradiction, thus we can conclude trivial by ex falso.
2. (first disjunct) Typing $\Gamma_1', \gamma_1' \vdash \mathsf{forkLinear}\,(\lambda x.t) : A$, which by inversion must therefore be of the form:

$$\frac{\dfrac{\Gamma_1', x : \mathbf{Chan}\,P, \gamma_1' \vdash t : \mathsf{unit}}{\Gamma_1', \gamma_1' \vdash (\lambda x.t) : \mathbf{Chan}\,P \multimap \mathsf{unit}}\,\multimap_i}{\Gamma_1', \gamma_1' \vdash \mathsf{forkLinear}\,(\lambda x.t) : \mathbf{Chan}\,(\overline{P})}\,\multimap_e \tag{d.19}$$

where $A = \mathbf{Chan}\,(\overline{P})$ and $\Gamma_1'', x : \mathbf{Chan}\,(\overline{P}), \gamma_1'' \vdash Ctxt\{x\} : B$ where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and $\gamma_1 = \gamma_1' + \gamma_1''$.

Therefore, for the goal of this theorem we construct the typings:

$$\frac{}{\varnothing, \overline{c} : \mathbf{Chan}\,(\overline{P}) \vdash \overline{c} : \mathbf{Chan}\,(\overline{P})}\,\text{VAR}$$

and

$$\frac{}{\varnothing, c : \mathbf{Chan}\,P \vdash c : \mathbf{Chan}\,P}\,\text{VAR} \tag{d.20}$$

By Lemma 23 then $\Gamma_1'', \overline{c} : \mathbf{Chan}\,(\overline{P}) \vdash Ctxt\{\overline{c}\} : B$.

We construct the goal channel configuration typing as:

$$\dfrac{\dfrac{\gamma_1 \vdash \mathcal{C}}{\gamma_1 + \gamma_2, c : \mathbf{Chan}\, P \vdash \mathcal{C}, c = \langle\,\rangle}\ \textsc{empty}}{\gamma_1 + \gamma_2, c : \mathbf{Chan}\, P, \overline{c} : \mathbf{Chan}\, (\overline{P}) \vdash \mathcal{C}, c = \langle\,\rangle, \overline{c} = \langle\,\rangle}\ \textsc{empty}$$

with process configuration typing then given by:

$$\dfrac{\dfrac{\Gamma_1'', \gamma_1'', \overline{c} : \mathbf{Chan}\, (\overline{P}) \vdash Ctxt\{\overline{c}\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1'' + \Gamma_2, \gamma_1'' + \gamma_2, \overline{c} : \mathbf{Chan}\, (\overline{P}) \vdash \mathcal{P} \mid Ctxt\{\overline{c}\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R} \qquad \dfrac{\text{(d.19)}\ \ \text{(d.20)}}{\Gamma_1', \gamma_1', c : \mathbf{Chan}\, P \vdash t[c/x] : \mathsf{unit}}\ \textsc{subst}}{\Gamma_1'' + \Gamma_2 + \Gamma_1', \gamma_1' + \gamma_2 + \gamma_1'', c : \mathbf{Chan}\, P, \overline{c} : \mathbf{Chan}\, (\overline{P}) \vdash \mathcal{P} \mid t[c/x]^\circ \mid Ctxt\{\overline{c}\}^{\mathcal{R}} : A}\ \textsc{sub}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c : \mathbf{Chan}\, P, \overline{c} : \mathbf{Chan}\, (\overline{P}) \vdash \mathcal{P} \mid t[c/x]^\circ \mid Ctxt\{\overline{c}\}^{\mathcal{R}} : A} \equiv$$

where disjointness conditions in the above for c and $\overline{c}$ are satisfied by the fact that c is fresh as per the semantics, and disjointness of the consituent contexts comes from their original disjointness being defined (e.g., $\gamma_1 = \gamma_1' + \gamma_1''$).

– (forkNonLinear)

$$\dfrac{\#c}{\begin{array}{ll} & (\mathcal{P} \mid Ctxt\{\mathsf{forkNonLinear}\, (\lambda x.t)\};\ \mathcal{C}) \\ \rightsquigarrow & (\mathcal{P} \mid Ctxt\{[\overline{c}]\} \mid t[[c]/x]^\circ;\ \mathcal{C}, \overline{c}^* = \langle\,\rangle, c^* = \langle\,\rangle) \end{array}}\ \textsc{forkNonLinear}$$

with process configuration typing:

$$\dfrac{\Gamma_1, \gamma_1 \vdash Ctxt\{\mathsf{forkNonLinear}\, (\lambda x.t)\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2 \vdash \mathcal{P} \mid Ctxt\{\mathsf{forkNonLinear}\, (\lambda x.t)\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R} \tag{d.21}$$

with channel configuration typing:

$$\gamma_1 + \gamma_2 \vdash \mathcal{C} \tag{d.22}$$

From typing $\Gamma_1, \gamma_1 \vdash Ctxt\{\mathsf{forkNonLinear}\, (\lambda x.t)\} : B$, and by lemma 22 there are two possibilities:

1. (second disjunct) (with $Ctxt\{\mathsf{forkNonLinear}\, (\lambda x.t)\}$ factorising into a graded binding) Gives $\neg\mathsf{resourceAllocator}(\mathsf{forkNonLinear}\, (\lambda x.t))$ which is a contradiction, thus we can conclude trivial by ex falso.
2. (first disjunct) There is a typing $\Gamma_1', \gamma_1' \vdash \mathsf{forkNonLinear}\, (\lambda x.t) : A$, which by inversion must therefore be of the form:

$$\dfrac{\dfrac{\Gamma_1', x : \Box_r(\mathbf{Chan}\, P), \gamma_1' \vdash t : \mathsf{unit}}{\Gamma_1', \gamma_1' \vdash (\lambda x.t) : \Box_r(\mathbf{Chan}\, P) \multimap \mathsf{unit}}\ \multimap_i}{\Gamma_1', \gamma_1' \vdash \mathsf{forkNonLinear}\, (\lambda x.t) : \Box_r(\mathbf{Chan}\, (\overline{P}))}\ \multimap_e \tag{d.23}$$

where $A = \Box_r(\mathbf{Chan}\, (\overline{P}))$ and $\Gamma_1'', x : \Box_r(\mathbf{Chan}\, (\overline{P})), \gamma_1'' \vdash Ctxt\{x\} : B$ and where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and $\gamma_1 = \gamma_1' + \gamma_1''$.

For the goal of this theorem we construct the typings:

$$\dfrac{\dfrac{}{\emptyset, \overline{c} :^* \mathbf{Chan}\, (\overline{P}) \vdash \overline{c} : \mathbf{Chan}\, (\overline{P})}\ \textsc{var}}{\emptyset, \overline{c} :^* \mathbf{Chan}\, (\overline{P}) \vdash [\overline{c}] : \Box_r(\mathbf{Chan}\, (\overline{P}))}\ \textsc{pr} \tag{d.24}$$

and

$$\dfrac{\dfrac{}{\varnothing, c :^* \mathbf{Chan}\, P \vdash c : \mathbf{Chan}\, P}\,\textsc{var}}{\varnothing, c :^* \mathbf{Chan}\, P \vdash [c] : \Box_r(\mathbf{Chan}\, P)}\,\textsc{pr} \tag{d.25}$$

By Lemma 23 then $\Gamma_1'', \gamma_1'', \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash Ctxt\{[\overline{c}]\} : B$.

We then construct the consequent channel configuration typing:

$$\dfrac{\dfrac{\dfrac{}{\gamma_1 + \gamma_2 \vdash \mathcal{C}}(\text{d.22})}{\gamma_1 + \gamma_2, c :^* \mathbf{Chan}\, P \vdash \mathcal{C}, c^* = \langle\rangle}\,\textsc{empty}}{\gamma_1 + \gamma_2, c :^* \mathbf{Chan}\, P, \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash \mathcal{C}, c^* = \langle\rangle, \overline{c^*} = \langle\rangle}\,\textsc{empty}$$

and process configuration:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1'', \gamma_1'', \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash Ctxt\{[\overline{c}]\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1'' + \Gamma_2, \gamma_1'' + \gamma_2, \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash \mathcal{P} \mid Ctxt\{[\overline{c}]\}^{\mathcal{R}} : A}\,\textsc{cons}\mathcal{R} \qquad \dfrac{(\text{d.23}) \quad (\text{d.25})}{\Gamma_1', \gamma_1', c :^* \mathbf{Chan}\, P \vdash t[[c]/x] : \mathsf{unit}}\,\textsc{graded-subst}}{\Gamma_1' + \Gamma_1'' + \Gamma_2, \gamma_1' + \gamma_1'' + \gamma_2, c :^* \mathbf{Chan}\, P, \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash \mathcal{P} \mid Ctxt\{[\overline{c}]\}^{\mathcal{R}} \mid t[[c]/x]^\circ : A}\,\textsc{sub}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c :^* \mathbf{Chan}\, P, \overline{c} :^* \mathbf{Chan}\, \overline{P} \vdash \mathcal{P} \mid Ctxt\{[\overline{c}]\}^{\mathcal{R}} \mid t[[c]/x]^\circ : A} \equiv$$

where disjointness conditions in the above for c and $\overline{c}$ are satisfied by the fact that c is fresh as per the semantics, and disjointness of the consituent contexts comes from their original disjointness being defined (e.g., $\gamma_1 = \gamma_1' + \gamma_1''$.

– (forkReplicate)

$$\dfrac{\#c_1 \ldots \#c_n}{(\mathcal{P} \mid Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\};\ \mathcal{C})}\,\textsc{forkRep}$$
$$\rightsquigarrow \quad (\mathcal{P} \mid Ctxt\{\langle[\overline{c_1}], .., [\overline{c_n}]\rangle\} \mid \mathsf{Wch}\,(c_1 \mapsto \lambda x.t) \mid .. \mid \mathsf{Wch}\,(c_n \mapsto \lambda x.t) \;;\; \mathcal{C}, {c_1}^* = \langle\rangle, .., {c_n}^* = \langle\rangle, \overline{c_1}^* = \langle\rangle, .., \overline{c_n}^* = \langle\rangle)$$

with process configuration typing:

$$\dfrac{\Gamma_1, \gamma_1 \vdash Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2 \vdash \mathcal{P} \mid Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\}^{\mathcal{R}} : A}\,\textsc{cons}\mathcal{R} \tag{d.26}$$

with channel configuration typing:

$$\gamma_1 + \gamma_2 \vdash \mathcal{C} \tag{d.27}$$

From typing $\Gamma_1, \gamma_1 \vdash Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\} : B$, and by lemma 22 there are two possibilities:

1. (second disjunct) (with $Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\}$ factorising into a graded binding) Gives $\neg\mathsf{resourceAllocator}(\mathsf{forkReplicate}\,[\lambda x.t]\,n)$ which is a contradiction, thus we can conclude trivial by ex falso.
2. (first disjunct) then there is a typing $\Gamma_1'', \gamma_1'' \vdash \mathsf{forkReplicate}\,[\lambda x.t]\,n : A$, which by inversion must therefore be of the form:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1'', x : \mathbf{Chan}\,P, \gamma_1'' \vdash t : \mathsf{unit}}{\Gamma_1'', \gamma_1'' \vdash \lambda x.t : \mathbf{Chan}\,P \multimap \mathsf{unit}}\multimap_i}{(0..n)\cdot\Gamma_1'', \gamma_1'' \vdash [\lambda x.t] : \Box_{0..n}(\mathbf{Chan}\,P \multimap \mathsf{unit})}\,\text{PR}}{(0..n)\cdot\Gamma_1'', \gamma_1'' \vdash \mathsf{forkReplicate}\,[\lambda x.t] : \mathbf{N}\,n \multimap \mathbf{Vec}\,n\,(\Box_{0..1}(\mathbf{Chan}\,(\overline{P})))}\multimap_e}{(0..n)\cdot\Gamma_1'', \gamma_1'' \vdash \mathsf{forkReplicate}\,[\lambda x.t]\,n : \mathbf{Vec}\,n\,(\Box_{0..1}(\mathbf{Chan}\,(\overline{P})))}\multimap_e \quad \text{(d.28)}$$

where $A = \mathbf{Vec}\,n\,(\Box_{0..1}(\mathbf{Chan}\,(\overline{P})))$ and $\Gamma_1', x : \mathbf{Vec}\,n\,(\Box_{0..1}(\mathbf{Chan}\,(\overline{P}))), \gamma_1' \vdash Ctxt\{x\} : B$ and where $\gamma_1 = \gamma_1' + \gamma_1''$ and where $\Gamma_1 = \Gamma_1' + (0..n)\cdot\Gamma_1''$

Furthermore by the $\mathsf{ReceivePrefix}(P)$ constraint of forkReplicate we know that $P = \mathbf{Recv}\ A'\ P'$.

For the goal of this theorem we construct the typings:

$$\dfrac{\cdots}{\overline{c_1} :^* \mathbf{Chan}\,(\mathbf{Send}\ A'\ P'), .., \overline{c_n} :^* \mathbf{Chan}\,(\mathbf{Send}\ A'\ P') \vdash \langle [\overline{c_1}], .., [\overline{c_n}]\rangle : \mathbf{Vec}\,n\,(\Box_{0..1}(\mathbf{Chan}\,(\mathbf{Send}\ A'\ P')))}\,\text{CONS}^n + \text{NIL}$$

By Lemma 23 then:

$\Gamma_1', \gamma_1', \overline{c_1} :^* \mathbf{Chan}\,(\mathbf{Send}\ A'\ P'), .., \overline{c_n} :^* \mathbf{Chan}\,(\mathbf{Send}\ A'\ P') \vdash Ctxt\{\langle [\overline{c_1}], .., [\overline{c_n}]\rangle\} : B$.

The consequent channel configuration has type derivation (and as a short hand we write $c_i : \mathbf{Chan}\,P_1$ for $c_1 :^* \mathbf{Chan}\,P_1, .., c_n :^* \mathbf{Chan}\,P_n$):

$$\dfrac{\gamma_1 + \gamma_2 \vdash \mathcal{C}}{\gamma_1 + \gamma_2, c_i : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P'), \overline{c_i} :^* \mathbf{Chan}\,(\mathbf{Send}\ A'\ P') \vdash \mathcal{C}, {c_1}^* = \langle\rangle, .., {c_n}^* = \langle\rangle, \overline{c_1}^* = \langle\rangle, .., \overline{c_n}^* = \langle\rangle}\,\text{EMPTY}^*$$

We then construct the process configuration typing where $\gamma' = \gamma, c_i :^* \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P'), \overline{c_i} :^* \mathbf{Chan}\,(\mathbf{Send}\ A'\ \overline{P'})$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1', \gamma_1', \overline{c_i} :^* \mathbf{Chan}\,(\mathbf{Send}\ A'\ \overline{P'}) \vdash Ctxt\{\langle [\overline{c_1}], .., [\overline{c_n}]\rangle\} : B \quad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1' + \Gamma_2, \gamma_1' + \gamma_2, \overline{c_i} :^* \mathbf{Chan}\,(\mathbf{Send}\ A'\ \overline{P'}) \vdash \mathcal{P} \mid Ctxt\{\langle [\overline{c_1}], .., [\overline{c_n}]\rangle\}^{\mathcal{R}} : A}\,\text{CONS}\mathcal{R} \quad \overline{\Gamma_1'', x : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P'), \gamma_1'' \vdash t : \mathsf{unit}}\,(\text{d.28})}{\Gamma_1' + (0..1\cdot\Gamma_1'') + \Gamma_2, \gamma_1' + \gamma_1'' + \gamma_2, \gamma' \vdash \mathcal{P} \mid Ctxt\{\langle [\overline{c_1}], .., [\overline{c_n}]\rangle\}^{\mathcal{R}} \mid \mathsf{Wch}\,(c_1 \mapsto \lambda x.t) : A}\,\text{WATCH}}{\Gamma_1' + (0..n\cdot\Gamma_1'') + \Gamma_2, \gamma_1' + \gamma_1'' + \gamma_2, \gamma' \vdash \mathcal{P} \mid Ctxt\{\langle [\overline{c_1}], .., [\overline{c_n}]\rangle\}^{\mathcal{R}} \mid \mathsf{Wch}\,(c_1 \mapsto \lambda x.t) \mid .. \mid \mathsf{Wch}\,(c_n \mapsto \lambda x.t) : A}\,\text{WATCH}^*}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, \gamma' \vdash \mathcal{P} \mid Ctxt\{\langle [\overline{c_1}], .., [\overline{c_n}]\rangle\}^{\mathcal{R}} \mid \mathsf{Wch}\,(c_1 \mapsto \lambda x.t) \mid .. \mid \mathsf{Wch}\,(c_n \mapsto \lambda x.t) : A} \equiv$$

since $0..1 + \ldots + 0..1$ n-times is equal to $0..n$, satisfing the typing goal, and where disjointness conditions in the above for c_i and $\overline{c_i}$ are satisfied by the fact that c_i are all fresh as per the semantics, and

disjointness of the consituent contexts comes from their original disjointness being defined (e.g., $\gamma_1 = \gamma_1' + \gamma_1''$.

– (watch)

$$\frac{}{(\mathcal{P} \mid \mathsf{Wch}\,(c \mapsto \lambda x.t);\ \mathcal{C}, c = v : \vec{v}) \rightsquigarrow (\mathcal{P} \mid t[c/x]^{\circ};\ \mathcal{C}, c = v : \vec{v})}\ \textsc{watcher}$$

with process configuration typing:

$$\frac{\Gamma_1, x : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P), \gamma_1 \vdash t : \mathsf{unit} \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{0..1 \cdot \Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P) \vdash \mathcal{P} \mid \mathsf{Wch}\,(c \mapsto \lambda x.t) : A}\ \textsc{watch} \tag{d.29}$$

and channel configuration typing:

$$\frac{\gamma_1 + \gamma_2 \vdash \mathcal{C} \qquad \gamma_1 + \gamma_2 \vdash v : A'}{\gamma_1 + \gamma_2, c : (\mathbf{Recv}\ A'\ P) \vdash \mathcal{C}, c = v : \vec{v}} \tag{d.30}$$

For the goal of this theorem we construct the typing:

$$\Gamma_1, x : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P), \gamma_1 \vdash t : \mathsf{unit}\ \ (\text{d.35})$$

$$\frac{\dfrac{\dfrac{}{c : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P) \vdash c : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P)}\ \textsc{var}}{1..1 \cdot \Gamma_1, \gamma_1, c : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P) \vdash t[c/x] : \mathsf{unit}}\ \textsc{subst}}{0..1 \cdot \Gamma_1, \gamma_1, c : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P) \vdash t[c/x] : \mathsf{unit}}\ \textsc{approx} \tag{d.31}$$

From this we can construct the channel configuration typing as just (d.30) and the process config typing:

$$\frac{\Gamma_2, \gamma_2 \vdash \mathcal{P} : A \qquad 0..1 \cdot \Gamma_1, \gamma_1, c : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P) \vdash t[c/x] : \mathsf{unit}\ (\text{d.31})}{0..1 \cdot \Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, c : \mathbf{Chan}\,(\mathbf{Recv}\ A'\ P) \vdash \mathcal{P} \mid t[c/x]^{\circ} : A}\ \textsc{watch}$$

and the premise channel configuration provides the consequent channel configuration.

– (forkMulticast)

$$\frac{\#c \qquad \#c_1 \ldots \#c_n}{\begin{array}{l}(\mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkMulticast}\,(\lambda x.t)\,n\};\ \mathcal{C}) \\ \rightsquigarrow\ (\mathcal{P} \mid \mathit{Ctxt}\{\langle \overline{c_1}, .., \overline{c_n} \rangle\} \mid t[c/x]^{\circ} \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n) \\ \qquad ;\ \mathcal{C}, c = \langle\,\rangle, \bar{c} = \langle\,\rangle, c_1 = \langle\,\rangle, .., c_n = \langle\,\rangle, \overline{c_1} = \langle\,\rangle, .., \overline{c_n} = \langle\,\rangle)\end{array}}\ \textsc{forkMulticast}$$

with process configuration typing:

$$\frac{\Gamma_1, \gamma_1 \vdash \mathit{Ctxt}\{\mathsf{forkMulticast}\,(\lambda x.t)\,n\} : B \qquad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2 \vdash \mathcal{P} \mid \mathit{Ctxt}\{\mathsf{forkMulticast}\,(\lambda x.t)\,n\}^{\mathcal{R}} : A}\ \textsc{cons}\mathcal{R} \tag{d.32}$$

with channel configuration typing:

$$\gamma_1 + \gamma_2 \vdash \mathcal{C} \tag{d.33}$$

From typing $\Gamma_1, \gamma_1 \vdash Ctxt\{\mathsf{forkMulticast}\,(\lambda x.t)\,n\} : B$, and by lemma 22 there are two possibilities:

1. (second disjunct) (with $Ctxt\{\mathsf{forkReplicate}\,[\lambda x.t]\,n\}$ factorising into a graded binding) Gives $\neg\mathsf{resourceAllocator}(\mathsf{forkReplicate}\,[\lambda x.t]\,n)$ which is a contradiction, thus we can conclude trivial by ex falso.
2. (first disjunct) Then there is a typing $\Gamma_1', \gamma_1' \vdash \mathsf{forkMulticast}\,(\lambda x.t)\,n : A$, which by inversion must therefore be of the form:

$$\dfrac{\dfrac{\dfrac{\Gamma_1', x : \mathbf{Chan}\,(\mathsf{Graded}_n P'), \gamma_1' \vdash t : \mathsf{unit}}{\Gamma_1', \gamma_1' \vdash \lambda x.t : \mathbf{Chan}\,(\mathsf{Graded}_n P') \multimap \mathsf{unit}}\,\multimap_i}{\Gamma_1', \gamma_1' \vdash \mathsf{forkMulticast}\,(\lambda x.t) : \mathbf{N}\,n \multimap \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P'})}\,\multimap_e}{\Gamma_1', \gamma_1' \vdash \mathsf{forkMulticast}\,(\lambda x.t)\,n : \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P'})}\,\multimap_e \tag{d.34}$$

where $A = \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})$ and $\Gamma_1'', x : \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P}), \gamma_1'' \vdash Ctxt\{x\} : B$ and $P = \mathsf{Graded}_n P'$ and where $\Gamma_1 = \Gamma_1' + \Gamma_1''$ and where $\gamma_1 = \gamma_1' + \gamma_1''$.

For the goal of this theorem we construct the typing:

$$\dfrac{\cdots}{\overline{c_1} : \mathbf{Chan}\,\overline{P}, .., \overline{c_n} : \mathbf{Chan}\,\overline{P} \vdash \langle \overline{c_1}, .., \overline{c_n} \rangle : \mathbf{Vec}\,n\,(\mathbf{Chan}\,\overline{P})}\,\textsc{cons}^n + \textsc{nil}$$

By Lemma 23 then $\Gamma_1'', \gamma_1'', \gamma' \vdash Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\} : B$

The consequent channel configuration has type derivation where we write $\gamma' = c_1 : \mathbf{Chan}\,P, .., c_n : \mathbf{Chan}\,P$ and $\overline{\gamma'} = \overline{c_1} : \mathbf{Chan}\,\overline{P}, .., \overline{c_n} : \mathbf{Chan}\,\overline{P}$.

$$\dfrac{\gamma_1 + \gamma_2 \vdash \mathcal{C}}{\gamma_1 + \gamma_2, c : \mathbf{Chan}\,(\mathsf{Graded}_n P), \bar{c} : \mathbf{Chan}\,\overline{(\mathsf{Graded}_n P)}, \gamma', \overline{\gamma'} \vdash \mathcal{C}, c = \langle\,\rangle, \bar{c} = \langle\,\rangle, c_1 = \langle\,\rangle, .., c_n = \langle\,\rangle, \overline{c_1} = \langle\,\rangle, .., \overline{c_n} = \langle\,\rangle}\,\textsc{empty}^*$$

We then construct the process configuration typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1'', \gamma_1'', \gamma' \vdash Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\} : B \quad \Gamma_2, \gamma_2 \vdash \mathcal{P} : A}{\Gamma_1'' + \Gamma_2, \gamma_1'' + \gamma_2, \gamma' \vdash \mathcal{P} \mid Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\}^{\mathcal{R}} : A}\,\textsc{cons}^{\mathcal{R}} \quad \dfrac{\Gamma_1', x : \mathbf{Chan}\,(\mathsf{Graded}_r P'), \gamma_1' \vdash t : \mathsf{unit}}{\Gamma_1', \gamma_1', c : \mathbf{Chan}\,(\mathsf{Graded}_n P') \vdash t[c/x] : \mathsf{unit}}\,\textsc{subst}}{\Gamma_1' + \Gamma_1'' + \Gamma_2, \gamma_1' + \gamma_1'' + \gamma_2, c : \mathbf{Chan}\,(\mathsf{Graded}_n P'), \overline{\gamma'} \vdash \mathcal{P} \mid Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\}^{\mathcal{R}} \mid t[c/x]^\circ : A}\,\textsc{cons}^{\mathcal{R}}}{\Gamma_1' + \Gamma_1'' + \Gamma_2, \gamma_1' + \gamma_1'' + \gamma_2, \gamma', \overline{\gamma'}, c : \mathbf{Chan}\,(\mathsf{Graded}_n P'), \bar{c} : \mathbf{Chan}\,(\overline{\mathsf{Graded}_n P'}) \vdash \mathcal{P} \mid Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\}^{\mathcal{R}} \mid t[c/x]^\circ \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n) : A}\,\textsc{fwd}}{\Gamma_1 + \Gamma_2, \gamma_1 + \gamma_2, \gamma', \overline{\gamma'}, c : \mathbf{Chan}\,(\mathsf{Graded}_n P'), \bar{c} : \mathbf{Chan}\,(\overline{\mathsf{Graded}_n P'}) \vdash \mathcal{P} \mid Ctxt\{\langle \overline{c_1}, .., \overline{c_n} \rangle\}^{\mathcal{R}} \mid t[c/x]^\circ \mid \mathsf{Fwd}\,(c \mapsto c_1, .., c_n) : A} \equiv$$

satisfying the goal.

– (forward)

$$\frac{}{(\mathcal{P} \mid \mathsf{Fwd}\ (c \mapsto c_1, .., c_n);\ \mathcal{C}, c = [v] : \vec{v}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n) \rightsquigarrow (\mathcal{P} \mid \mathsf{Fwd}\ (c \mapsto c_1, .., c_n);\ \mathcal{C}, c = \vec{v}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v)}\ \textsc{forwarder}$$

with process configuration typing:

$$\frac{\mathsf{Sends}(P) \qquad \Gamma, \gamma \vdash \mathcal{P} : A}{\Gamma, \gamma, c : \mathbf{Chan}\ (\mathsf{Graded}_n \overline{P}), c_i : \mathbf{Chan}\ P \vdash \mathcal{P} \mid \mathsf{Fwd}\ (c \mapsto c_1, .., c_n) : A}\ \textsc{fwd} \qquad \text{(d.35)}$$

(specialisation of the FWD rule since $\mathsf{Sends}(\overline{\mathbf{Recv}\ A'\ P}) = \mathsf{Sends}(\overline{P})$) and with channel configuration typing:

$$\frac{\dfrac{\gamma_0 \vdash v : A'}{\gamma_0 \vdash [v] : \Box_n A'}\ \textsc{pr} \qquad \dfrac{\gamma_0 \vdash v : A' \quad \gamma_0, \gamma_1, c_1 : \mathbf{Chan}\ P, .., c_n : \mathbf{Chan}\ P, c : \mathbf{Chan}\ \overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v, c = \vec{v}}{\gamma_0, \gamma_1, c_1 : \mathbf{Chan}\ (\mathbf{Send}\ A\ P), .., c_n : \mathbf{Chan}\ (\mathbf{Send}\ A\ P), c : \mathbf{Chan}\ \overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n, c = \vec{v}}\ \textsc{send}^n}{\gamma_0, \gamma_1, c_1 : \mathbf{Chan}\ (\mathbf{Send}\ A\ P), .., c_n : \mathbf{Chan}\ (\mathbf{Send}\ A\ P), c : \mathbf{Chan}\ (\mathbf{Recv}\ (\Box_n A')\ \overline{P}) \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n, c = [v] : \vec{v}}\ \textsc{recv}$$

(d.36)

where $\gamma = \gamma_0, \gamma_1$.

The goal channel configuration typing is then given by the premise of (d.36):

$$\frac{\gamma_0 \vdash v : A' \quad \gamma_1, c_1 : \mathbf{Chan}\ P, .., c_n : \mathbf{Chan}\ P, c : \mathbf{Chan}\ \overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1 : v, .., \overline{c_n} = \vec{v}_n : v, c = \vec{v}}{\gamma_0, \gamma_1, c_1 : \mathbf{Chan}\ (\mathbf{Send}\ A\ P), .., c_n : \mathbf{Chan}\ (\mathbf{Send}\ A\ P), c : \mathbf{Chan}\ \overline{P} \vdash \mathcal{C}, \overline{c_1} = \vec{v}_1, .., \overline{c_n} = \vec{v}_n, c = \vec{v}}\ \textsc{send}^n$$

From the definition of $\mathsf{Sends}(\overline{\mathbf{Recv}\ A'\ P}) = \mathsf{Sends}(\overline{P})$ and $\mathsf{Sends}(\mathbf{Send}\ A'\ P) = \mathsf{Sends}(P)$ therefore we can define the process configuration typing as:

$$\frac{\mathsf{Sends}((\overline{P})) \qquad \mathsf{Sends}(P) \qquad \Gamma, \gamma \vdash \mathcal{P} : A}{\Gamma, \gamma, \overline{c} : \mathbf{Chan}\ (\overline{P}), c_1 : \mathbf{Chan}\ P, .., c_n : \mathbf{Chan}\ P \vdash \mathcal{P} \mid \mathsf{Fwd}\ (c \mapsto c_1, .., c_n) : A}\ \textsc{fwd}$$

□

D.4 FURTHER COMPARISON WITH GV

The construction in GV of the replicate primitive can also be captured in Granule. Their construction [80, Appendix B] can be written in Granule with $\Box A = A$ `[Many]` (where $\mathcal{R} = \{\mathtt{Zero}, \mathtt{One}, \mathtt{Many}\}$ is a semiring for capturing linear/non-linear behaviour [63]) but only if we both turn *off* the resource allocator restriction on promotion that we introduced in Chapter 5, which was critical for soundness in our call-by-value semantics, and use call-by-name semantics, both of which are provided by a language extension:

```
-- Call-by-Name semantics, also turns off the restriction
-- to non-resource allocating terms in promotion
language CBN

-- GV 'replicate'
replicate : ∀ {s : Protocol}
          . (LChan (Send ((LChan (Dual s)) [Many]) End)
          , (LChan s → ()) [Many]) → LChan End
replicate (x, f) =
let [g] = f in send x [forkLinear g]

-- Dual 'request' primitive
request : ∀ {s : Protocol}
        . LChan (Recv ((LChan s) [Many]) End) → LChan s
request s = let (w , s) = recv s
            in let () = close s
            in  let [y] = w in y
```

Granule

D.5 INVOLUTION IS AN ISOMORPHISM

We show that for all τ, then τ^{-1} is a sequentially realizable involution up to isomorphism, i.e., $(\tau^{-1})^{-1} \cong \tau$, with $\tau \multimap ((\tau \multimap 1) \multimap 1)$ implemented as $\lambda x.\lambda f.f\ x$ and the converse direction as follows, using the syntax of GV calculus (as formulated by Lindley and Morris [80]) rather than Granule, of type $((\tau \multimap 1) \multimap 1) \multimap \tau$:

$$(\lambda k.\mathbf{let}\ (x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.k(\lambda x.\mathsf{send}\ c\ x)))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x)$$

In order to prove that τ^{-1} is an involution up to isomorphism, we need to show that the functions $i : \tau \multimap ((\tau \multimap 1) \multimap 1$ and $j : ((\tau \multimap 1) \multimap 1) \multimap \tau$ are mutually inverse, or in other words that $j(i(t)) = t : \tau$ and $i(j(h)) = h : (\tau \multimap 1) \multimap 1$.

We leverage the $\beta\eta$-equality theory of GV based on its operational semantics given by Lindley and Morris [80].

We show both directions separately, as follows:

$$\begin{aligned}
j(i(t)) &= (\lambda k.\mathbf{let}\ (x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.k(\lambda x.\mathsf{send}\ c\ x)))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x)(\lambda f.f\ t) \\
&= (\mathbf{let}\ (x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.(\lambda f.f\ t)(\lambda x.\mathsf{send}\ c\ x)))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x) \\
&= (\mathbf{let}\ (x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.(\lambda x.\mathsf{send}\ c\ x)t))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x) \\
&= (\mathbf{let}\ (x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t))\ \mathbf{in}\ \mathbf{let}\ () = \mathsf{wait}\ c\ \mathbf{in}\ x)
\end{aligned}$$

Checking the typing of the inner expression, we have:

$$\begin{aligned}
\lambda c.\mathsf{send}\ c\ t &: \mathsf{Chan}(!\tau.\mathsf{end}_!) \multimap \mathsf{Chan}(\mathsf{end}_!) \\
\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t) &: \mathsf{Chan}(?\tau.\mathsf{end}_?) \\
\mathsf{recv}(\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t)) &: \tau \otimes \mathsf{Chan}(\mathsf{end}_?)
\end{aligned}$$

So we have the binding $(x, c) : \tau \otimes \mathsf{Chan}(\mathsf{end}_?)$. Applying the global configuration semantics of GV [80, Figure 4], we then get the following:

$$\begin{aligned}
& \textbf{let } (x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.\mathsf{send}\ c\ t))\ \textbf{in let } () = \mathsf{wait}\ c\ \textbf{in}\ x \\
(\textsc{Lift+Fork}) \rightsquigarrow\ & (\nu \mathsf{c})(\textbf{let } (x,c) = \mathsf{recv}\ \mathsf{c}\ \textbf{in let } () = \mathsf{wait}\ c\ \textbf{in}\ x) \mid (\mathsf{send}\ \mathsf{c}\ t) \\
(\textsc{Lift+Send}) \rightsquigarrow\ & (\nu \mathsf{c})(\textbf{let } (x,c) = (t,\mathsf{c})\ \textbf{in let } () = \mathsf{wait}\ c\ \textbf{in}\ x) \mid \mathsf{c} \\
(\textsc{LiftV}) \rightsquigarrow\ & (\nu \mathsf{c})(\textbf{let } () = \mathsf{wait}\ \mathsf{c}\ \textbf{in}\ t) \mid \mathsf{c} \\
(\textsc{Lift+Wait}) \rightsquigarrow\ & \textbf{let } () = ()\ \textbf{in}\ t \\
(\textsc{LiftV}) \rightsquigarrow\ & t
\end{aligned}$$

Thus, $j(i(t)) = t : \tau$ as required.

In the opposite direction of the isomorphism we then have, $h : (\tau^{-1})^{-1}$ with

$$\begin{aligned}
i(j(h)) &= (\lambda x.\lambda f.f\ x)(\textbf{let } (x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\ \textbf{in let } () = \mathsf{wait}\ c\ \textbf{in}\ x) \\
&= (\lambda f.f\ (\textbf{let } (x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\ \textbf{in let } () = \mathsf{wait}\ c\ \textbf{in}\ x)) \\
&= \textbf{let } (x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\ \textbf{in let } () = \mathsf{wait}\ c\ \textbf{in}\ (\lambda f.f\ x)
\end{aligned}$$

Similarly to the first case, we then have the inner typing:

$$\begin{aligned}
\lambda x.\mathsf{send}\ c\ x &: \tau \multimap \mathsf{Chan}(\mathsf{end}_!) \\
h(\lambda x.\mathsf{send}\ c\ x) &: \mathsf{Chan}(\mathsf{end}_!) \\
\mathsf{fork}(h(\lambda x.\mathsf{send}\ c\ x)) &: \mathsf{Chan}(?\tau.\mathsf{end}_?) \\
\mathsf{recv}(\mathsf{fork}(h(\lambda x.\mathsf{send}\ c\ x))) &: \tau \otimes \mathsf{Chan}(\mathsf{end}_?)
\end{aligned}$$

So, again, $(x, c) : \tau \otimes \mathsf{Chan}(\mathsf{end}_?)$.

Applying the global configuration semantics of GV [80, Figure 4], we get:

$$\begin{aligned}
& \textbf{let } (x,c) = \mathsf{recv}(\mathsf{fork}(\lambda c.h(\lambda x.\mathsf{send}\ c\ x)))\ \textbf{in let } () = \mathsf{wait}\ c\ \textbf{in}\ (\lambda f.f\ x) \\
(\textsc{Lift+Fork}) \rightsquigarrow\ & (\nu \mathsf{c})\ (\textbf{let } (x,c) = \mathsf{recv}\ \mathsf{c}\ \textbf{in let } () = \mathsf{wait}\ c\ \textbf{in}\ (\lambda f.f\ x) \mid h\ (\lambda x.\mathsf{send}\ \mathsf{c}\ x))
\end{aligned}$$

Recall that $h : (\tau \multimap 1) \multimap 1$ therefore we know that h must necessarily use the input parameter, applying it to some $t : \tau$, therefore after some reduction in $h\ (\lambda x.\mathsf{send}\ \mathsf{c}\ x)$ we get some $\textbf{let } c' = \mathsf{send}\ \mathsf{c}\ t\ \textbf{in}\ h' : ()$ and some configuration C in the case that evaluating to this point had some other communication effects. Note that c' is not in the free variables of h' since the session typing tells us it is unused, i.e. $c' : \mathsf{Chan}(\mathsf{end}_!)$.

Then we get the continuing reduction sequence:

$$\begin{aligned}
(\textsc{LiftV}*) \rightsquigarrow^*\ & (\nu \mathsf{c})\ (\textbf{let } (x,c) = \mathsf{recv}\ \mathsf{c}\ \textbf{in let}() = \mathsf{wait}\ c\ \textbf{in}(\lambda f.f\ x) \mid \textbf{let } c' = \mathsf{send}\ \mathsf{c}\ t\ \textbf{in}\ h' \mid C) \\
(\textsc{Lift+Send}) \rightsquigarrow\ & (\nu \mathsf{c})\ (\textbf{let } (x,c) = (t,\mathsf{c})\ \textbf{in let}() = \mathsf{wait}\ c\ \textbf{in}(\lambda f.f\ x) \mid \textbf{let } c' = \mathsf{c}\ \textbf{in}\ h' \mid C) \\
(\textsc{LiftV}) \rightsquigarrow\ & (\nu \mathsf{c})\ (\textbf{let } () = \mathsf{wait}\ \mathsf{c}\ \textbf{in}\ (\lambda f.f\ t) \mid \textbf{let } c' = \mathsf{c}\ \textbf{in}\ h' \mid C) \\
(\textsc{Lift+Wait}) \rightsquigarrow\ & (\lambda f.f\ t) \mid h' \mid C
\end{aligned}$$

The result is a term that behaves like the original h; applying the term t from inside h to the continuation to f results in a configuration C and has some remaining reduction to do as h'. Thus $i(j(h)) = h : (\tau \multimap 1) \multimap 1$ as required.

DECLARATION

I declare that this thesis has been composed by myself and that the work has not been submitted for any other degree or professional qualification. I confirm that the work submitted is my own, except where work which has formed part of jointly-authored publications has been included. My contribution and those of the other authors to this work have been explicitly indicated below. I confirm that appropriate credit has been given within this thesis where reference has been made to the work of others.

The background and foundations described in Chapters 2 and 3 is largely based on existing work on linear and graded types, but some of the details on the ! modality were previously presented at TLLA 2021 in *"Linear Exponentials as Graded Modal Types"* by Jack Hughes, myself, James Wood and Dominic Orchard. The first three authors (including myself) held joint first authorship.

The work described in Chapter 4 was previously published at ESOP 2022 in *"Linearity and Uniqueness: An Entente Cordiale"* by myself, Michael Vollmer, and Dominic Orchard. I was first author.

The work described in Chapter 5 was previously published at OOPSLA 2024 in *"Functional Ownership as Fractional Uniqueness"* by myself and Dominic Orchard. I was first author.

Some of the ideas around inverses mentioned in Chapter 6 were previously published at ECOOP 2022 in *"How to Take the Inverse of a Type"* by myself and Dominic Orchard. I was first author.

The work described in Chapters 6 and 7 was previously published in Information and Computation as *"Non-Linear Communication via Graded Modal Session Types"* by myself and Dominic Orchard. I was first author. This paper was based on a previous presentation at PLACES 2022 titled *"Replicate, Reuse, Repeat: Capturing Non-Linear Communication via Session Types and Graded Modal Types"* by myself and Dominic Orchard. I was first author on that extended abstract also.

The ideas presented in Section 8.1 were previously presented at PLAS 2022 in *"Graded Modal Types for Integrity and Confidentiality"* by myself and Dominic Orchard. I was first author.

Glasgow, March 2025

Danielle Georgia Marshall

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author; a collection of postcards received so far is featured here:

http://postcards.miede.de/

Thank you very much for your feedback and contribution.

Final Version as of April 7, 2026 (classicthesis v4.6).