JACK OLIVER HUGHES

PROGRAM SYNTHESIS FROM LINEAR AND GRADED TYPES

PROGRAM SYNTHESIS FROM LINEAR AND GRADED TYPES

JACK OLIVER HUGHES

A Thesis Submitted to the University of Kent in the Subject of Computer Science for the Degree of Doctor of Philosohpy

School of Computing
Division of Computing, Engineering and Mathematical Sciences
University of Kent

January, 2024

Jack Oliver Hughes: *Program Synthesis from Linear and Graded Types,* A Thesis Submitted to the University of Kent in the Subject of Computer Science for the Degree of Doctor of Philosohpy, © January, 2024

ABSTRACT

Graded types are a class of *resourceful* types which allow for fine-grained quantitative reasoning about data-flow in programs. Tracing their roots from linear types, the use of resource annotations (or *grades*) on data, allows a programmer to express structural or semantic properties of their program at the type level. Such systems have become increasingly popular in recent years, mainly for the expressive power that they offer to programmers; judicious use of grades in type specifications significantly reduces the number of typeable programs. These additional constraints on types lend themselves naturally to type-directed program synthesis, which leverage the information provided by types to prune ill-resourced programs from the search space of candidate programs. In synthesis, this grade information can be exploited to constrain the search space of programs even further than in standard type systems. We present an approach to program synthesis for linear and graded type systems, where grades form an arbitrary pre-ordered semiring. Harnessing this grade information in synthesis is non-trivial, and we explore the issues involved in designing and implementing a resource-aware program synthesis tool, culminating in an efficient and expressive program synthesis tool for the research programming language Granule, which uses a graded type system. We show that by harnessing grades in synthesis, the majority of our benchmarking synthesis problems (many of which involve recursive functions over recursive ADTs) require less exploration of the synthesis search space than a purely type-driven approach and with fewer needed input-output examples. Our type-and-graded-directed approach is demonstrated in the Granule but we also adapt it for synthesising Haskell programs that use GHC's Linear Types extension, demonstrating the versatility of our approach to resourceful program synthesis.

ACKNOWLEDGMENTS

Firstly, I am extremely grateful to my supervisor Dominic Orchard for his endless insight, encouragement, patience, and enthusiasm throughout the entirety of my PhD. It would not have been possible to ask for a better supervisor.

I am also indebted to both the PLAS group at the University of Kent, and the and the entirety of the Granule team, both of whom have helped me immensely along the way. In particular, I would like to thank Mark Batty and Mike Vollmer. Their advice and support in the final year of my PhD has been invaluable.

Thank you to my PhD examiners, Edwin Brady, Cristina David, and Vineet Rajani for their thoughtful feedback and questions, which have vastly improved the quality of this thesis.

I would also like to extend my gratitude to Martin Berger, who introduced me to the worlds of programming language theory and program synthesis, setting me on a path that I didn't imagine would end up here.

Of course, I could not have done it without my friends in Brighton, Eastbourne, and further afield. Thank you all - especially Paul, Sona, Maddie, James, Kaitlin, and Tian.

Most of all, thank you to my family. Mum, Dad, Nan, Grandad, Debbie, and of course my brother Jacob. Without you none of this would have been possible. Your support and encouragement over these past four years has been incredible. I love you all so much.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

PROGRAM SYNTHESIS FROM LINEAR AND GRADED TYPES

1

INTRODUCTION

Writing programs is difficult. Ensuring they are correct even more so. A long-standing constant among programmers is the desire to write correct programs with greater ease. To fulfil this end came *program synthesis*, where the computer shares the burden of writing the program with the human by writing part or all of the program for them. The essence of program synthesis is the automatic generation of program code from some specification of desired program behaviour. Program synthesis techniques take many forms and have been applied to almost all paradigms of programming, from early example-driven approaches such as Manna and Waldinger [1979]'s DEADALUS and Summers [1977]'s THESYS systems, to machine learning approaches such as *inductive logic programming* [Muggleton and de Raedt, 1994], to synthesising transformations between spreadsheet columns in Excel [Gulwani, 2011]. In recent years, the emergence of Large Language Models (LLMs) has led to their application in program synthesis tasks [Austin et al., 2021, Jain et al., 2021], where they are trained on datasets to generate code based on various specifications, including natural language descriptions of desired behaviour. However, a general observation one can make is that the task of generating a program automatically becomes easier the richer the specification provided by the programmer.

One of the most useful and well-studied verification and specification tools available to modern programmers is the type system. Not only do type systems allow many kinds of errors to be caught statically, they also help inform the design of a program. Many programmers begin writing their programs by first defining the types, from which the program code follows naturally. This phenomenon will be familiar to any who have written programs in typed functional programming languages, and results from the fact that types form a high-level abstract specification of program behaviour.

Type-directed program synthesis is a well-studied technique for automatically generating program code from a type specification - the *goal* type [Osera and Zdancewic, 2015, Kuncak et al., 2010, Frankle et al., 2016a, Albarghouthi et al., 2013, Feser et al., 2015, Osera, 2019, Smith and Albarghouthi, 2019, Knoth et al., 2019]. This approach has

a long history, which is deeply intertwined with automated theorem proving and proof synthesis, thanks to the Curry-Howard correspondence [Manna and Waldinger, 1980, Green, 1969].

One lens through which we can view this task is as an inversion of type checking: we start with a goal type and inductively synthesise well-typed sub-terms by breaking the goal into sub-goals, pruning the search space of programs via typing as we go. This approach follows the treatment of program synthesis as a form of proof search in logic: given a type A we want to find a program term t which inhabits A. We can express this in terms of a synthesis *judgement* which acts as the inversion of typing or proof rules:

$$\Gamma \vdash A \Rightarrow t$$

meaning that the term t can be synthesised for the goal type A under a context of assumptions Γ. We may construct a calculus of synthesis *rules* for a programming language, inductively defining the above synthesis judgement for each type former. For example, we may define a rule for standard product types in the following way:

$$\frac{\Gamma \vdash A \Rightarrow t_1 \qquad \Gamma \vdash B \Rightarrow t_2}{\Gamma \vdash A \times B \Rightarrow (t_1, t_2)} \times_{\textsc{Intro}}$$

Reading 'clockwise' from the bottom-left: to synthesise a value of type $A \times B$, we synthesise a value of type A and then a value of type B and combine them into a pair in the conclusion. The 'ingredients' for synthesising the sub-terms t_1 and t_2 come from the free-variable assumptions contained in Γ and any constructors of A and B.

Depending on the context, there could be many possible combinations of assumption choices to synthesise such a pair. Consider the following partial program containing a program *hole*, marked with ?, specifying a position at which we wish to perform synthesis:

$$f : A \to A \to A \to A \times A$$
$$f\ x\ y\ z = ?$$

The function has three parameters all of type A which can be used to synthesise an expression of the goal type $A \times A$. Expressing this synthesis problem as an instantiation of the above $\times_{\textsc{Intro}}$ rule:

$$\frac{x : A, y : A, z : A \vdash A \Rightarrow t_1 \qquad x : A, y : A, z : A \vdash A \Rightarrow t_2}{x : A, y : A, z : A \vdash A \times A \Rightarrow (t_1, t_2)} \times_{\textsc{Intro}}$$

Even in this simple setting, the number of possibilities starts to become unwieldy. For example, we could synthesise (x, x), (x, y), (y, x) etc. In total, there are nine (3^2) possible candidate programs based on combinations of x, y and z. Ideally, we would like some way of

constraining the number of choices that are required by the synthesis algorithm. Many systems achieve this by allowing the user to specify additional information about their desired program behaviour. For example, recent work has extended type-directed synthesis to refinement types [Polikarpova et al., 2016], cost specifications [Knoth et al., 2019], differential privacy [Smith and Albarghouthi, 2019], example-guided synthesis [Feser et al., 2015, Albarghouthi et al., 2013] or examples integrated with types [Frankle et al., 2016b, Osera and Zdancewic, 2015], and ownership information [Fiala et al., 2023]. The general idea is that, with more information, whether that be richer types, additional examples, or behavioural specifications, the proof search / program synthesis process can be pruned and refined, requiring less exploration of the search space of candidate programs, ideally reducing the overall synthesis time.

This work presents a program synthesis approach that leverages the information contained in *linear* and *graded type systems* that track and enforce program properties related to data flow, statically. We refer to these systems as *resourceful* type systems, since they treat data as though it is a physical resource, constraining how data can be used by a program and thus reducing the number of possible synthesis choices that need to be made. Our hypothesis is that resource-and-type-directed synthesis speeds up type-directed synthesis, reducing the number of paths that need to be explored and the amount of additional specification (e.g. input-output examples) required.

1.1 GRADED TYPE SYSTEMS

Graded type systems trace their roots to linear logic. In linear logic, data is treated as though it were a finite resource which must be consumed exactly once with arbitrary copying and discarding disallowed [Girard, 1987]. Since Girard's original formulation of linear logic, several works have rendered linear logic as a type system [Wadler, 1990, Abramsky, 1993], yielding *linear types*. The identify function $\lambda x.x$ is the ideal linearly typed program: it binds a variable and then uses it exactly once. The K combinator $\lambda x.\lambda y.x$ from SKI combinatory logic, however, would not be linearly typed as the second variable y is never used inside the body. Non-linear use of data is expressed through the ! modal operator (the *exponential modality*). This gives a binary view—a value may either be used exactly once (i.e. as a resource) or in a completely unconstrained way. For example, using ! the K combinator can be written as $\lambda x.\lambda !y.x$, where ! provides the capability to discard y. Bounded Linear Logic (BLL) refines this view, replacing ! with a family of indexed modal operators where the index provides an upper bound on usage [Girard et al., 1992], e.g., $!_{\leq 4}A$ represents a value A which may be used up to 4 times. In recent years, various works

have generalised BLL, resulting in *graded* type systems in which these indices are drawn from an arbitrary pre-ordered semiring [Brunel et al., 2014, Ghica and Smith, 2014, Petricek et al., 2014, Abel and Bernardy, 2020, Choudhury et al., 2021, Atkey, 2018, McBride, 2016]. This allows numerous program properties to be tracked and enforced statically, including various kinds of reuse, privacy and confidentiality, and capabilities. Graded types are becoming applied and adopted in practical systems and form the basis of Haskell's `LinearTypes` extension [Bernardy et al., 2018], Idris 2 [Brady, 2021], as well as the experimental language Granule [Orchard et al., 2019].

Semantically, these semiring indexed !-modalities are modelled by *graded exponential comonads* [Gaboardi et al., 2016]. The terminology of *graded modal types* was proposed by Orchard et al. [2019] encompassing both semiring indexed !-modalities and the notion of *graded monads* [Orchard et al., 2014, Katsumata, 2014, Smirnov, 2008], which generalise monads. In this work we do not consider the latter, dealing only with graded comonadic modalities, which are closely related to linearity. In general, we will refer to systems which make use of graded modal types as *graded* type systems.

Returning to our example in a graded setting, the arguments of the function now have *grades* (annotations) that, in this context, are natural numbers describing the exact number of times the parameters must be used (the choice here was ours):

$$f : A^2 \to A^0 \to A^0 \to A \times A$$
$$f\ x\ y\ z = ?$$

The first A is annotated with a grade 2, which in this context indicates that it *must* be used twice. Likewise, the types of y and z are graded with 0, enforcing zero usage, i.e., we are not allowed to use them in the body of f and must discard them.

The result is that there is only one (normal form) inhabitant for this type: (x, x); the other assumptions will not even be considered in synthesis, allowing us to effectively prune out branches which use resources in a way which violates the grades. In this example, these annotations take the form of natural numbers explaining how many times a value can be used, but we may instead wish to represent different kinds of program properties, such as sensitivity, strictness, or security levels for tracking non-interference, all of which are well-known instances of graded type systems [Orchard et al., 2019, Gaboardi et al., 2016, Abel and Bernardy, 2020]. Note that all of these examples are technically graded presentations of *coeffects*, tracking how a program uses its context, in contrast with graded types for side *effects* [Orchard et al., 2014, Katsumata, 2014], which we do not consider here.

We can divide these graded coeffect systems into two main classes, which we refer to as "linear graded" (or *linear base*) and "fully graded"

(or *graded base*) throughout. The first are those which trace their origins to linear logic, refining an existing non-graded type system with graded modal types. The functional programming language Granule is one such system. Granule is based on an underlying linear type system, with graded modal types introduced and eliminated explicitly, through a family of indexed modal operators $\Box_r$. For instance, $\Box_2 A \multimap B$ represents the type of a linear function where the argument type is a graded modal type, graded by 2. By "unboxing" the modality, we obtain a value which can be used twice inside the body of the function. This system is the default basis of Granule [Orchard et al., 2019]. The second approach does away with this linear basis, embedding graded modalities into the function types à la Idris 2 [Brady, 2021], McBride's QTT [McBride, 2016, Atkey, 2018], the coeffect calculi of Petricek et al. [2013] and Petricek et al. [2014], the core of Linear Haskell [Bernardy et al., 2018], and the unified graded modal calculus of [Abel and Bernardy, 2020]. In these systems, grades are expressed as annotations function arrows, i.e. $A^2 \to B$ expresses a similar idea as before, where A is an argument type graded by 2. However, in these systems, we do not need to do any unboxing to be able to use the value of type A according to its grade in the body of the function.

As graded type systems develop into fully fledged programming languages such as Idris 2 [Brady, 2021], and as traditionally non-graded languages, such as Haskell, incorporate features from graded types into their existing type systems [Bernardy et al., 2018], the proposition of harnessing the information conveyed by these richer types becomes increasingly attractive. As of this moment, program synthesis in the context of graded types has remained relatively unexplored despite the growing prevalence of such systems. Past works have tackled the problem of proof search in Linear Logic [Hodas and Miller, 1994, Cervesato et al., 2000], which are analogous, through the lens of Curry-Howard, to program synthesis solutions for linear types. These works lay the foundations for our approach here, however, graded types pose an entirely new set of challenges and integrating them into these existing approaches is complex.

For this work, we consider Granule [Orchard et al., 2019] to be an ideal candidate for the target language of a program synthesis tool for graded type systems. As mentioned, Granule is a functional programming language which combines a linear type system at its core with indexed data types. On top of this, graded modalities are integrated both as graded comonads and graded monads. We forgo the treatment of Granule's indexed types in synthesis, leaving this as future work. Instead we focus on building a synthesis tool for Granule, which unlocks the expressivity held by graded comonads for use inside the synthesis algorithm itself.

Granule also provides a language extension, known as *GradedBase*, which does away with this linear type system as its foundation. Instead, grades permeate the type system à la Petricek et al. [2014] and other graded base systems (as mentioned above). This extension provides us with a language capable of giving a broad representation of graded type systems. This will factor heavily into the design of our calculi, and we structure the rest of the thesis with this in mind.

1.2 CONTRIBUTIONS

Linear and graded types provide a rich specification of program behaviour, which is enforced by the type checker. But how do we use this information to make writing programs automatically easier?

THESIS STATEMENT

> Linear and graded types can be integrated into type-directed program synthesis to prune incorrect branches from the search space of candidate programs, yielding an efficient program synthesis algorithm which ensures that generated programs are type correct and *well-resourced* (by construction), i.e. they use linear values linearly, and graded values according to their grades.

In other words, grades can be used to synthesise programs which describe how they use their context of values. Having been given a graded type, the synthesis tool can know exactly how values are allowed to be used inside a program, and can use this information during the synthesis to aid its search for the correct solution.

The primary aim of this work is to demonstrate the feasibility and power of using resourceful types as the basis of a type-directed program synthesis tool, culminating in the development and implementation of an expressive, efficient, and feature-rich program synthesis tool for the Granule programming language.

We show that not only is program synthesis feasible in the context of a graded type system, but that the information conveyed by the grades can be used to prune the search space of programs in synthesis. This results in our synthesis algorithm needing to consider far less potential programs when constructing a program, as grade-violating candidate programs may be pruned out.

Specifically, this work makes the following contributions:

- We identify an approach which makes type-directed program synthesis in a resourceful setting feasible. Drawing inspiration from the work of Hodas and Miller [1994] on theorem proving, we adapt their work to graded types, and propose a dual to their

own method, yielding two schemes for managing resources in the synthesis of a program term.

- We use these schemes to construct two simple synthesis calculi for a simplified core of Granule, which demonstrate their effectiveness as tools for resourceful program synthesis. We implement both approaches as part of the Granule toolchain and compare their performance.
- We showcase an alternative and complementary approach to generating a subset of Granule programs, based around useful combinators for working in this setting. To do so, we make use of a system inspired by Haskell's generic type deriving mechanism [Magalhães et al., 2010] adapted to graded types.
- We then define a synthesis calculus for a fully graded type system. This type system is a feature-rich language based on Granule's *GradedBase* language extension, which includes recursion, recursive types, and user-defined ADTs. Furthermore, we again implement this calculus as part of the Granule toolchain.
- We evaluate our tool on a benchmark suite of recursive functional programs leveraging standard data types like lists, streams, and trees. We compare against non-graded synthesis provided by Myth [Osera and Zdancewic, 2015]. We also compare against our own tool *modulo grades*, i.e. we compare the number of synthesis paths explored by our tool when taking grades into account versus traditional un-graded synthesis.
- We prove that each of these systems is sound, i.e., synthesised programs are typed by the goal type. A key property here is that synthesised programs are not only well-typed, but also *well-resourced*, meaning that all values inside the program are used according to their resource constraints. We show that this property holds for each of our synthesis calculi as part of their soundness proofs.
- We demonstrate how our approach to resourceful program synthesis can be readily applied to other graded systems. Leveraging our calculus and implementation, we provide a prototype tool for synthesising Haskell programs written using GHC 9's Linear Types language extension.

1.3 STRUCTURE

This dissertation is structured into seven chapters. In the next chapter, Chapter 2, the theoretical background of linear and graded types is laid out. In doing so, we introduce two core calculi with simple types

and grades, which demonstrate the two major lineages of resourceful type systems; linear base and graded base. This chapter also introduces the programming language Granule, which implements each of the discussed calculi.

The rest of the dissertation is structured such that synthesis calculi for both of these systems are defined and presented, minimising any redundancy in the presentation. Despite the variances between the core calculi of Chapter 2, there is a substantial degree of overlap between the two. Thus, we adopt the following structure:

1. Chapter 3 introduces the core concepts of type-directed program synthesis from resourceful type systems using an extension of the typing calculus of Section 2.3.1. Specifically this chapter introduces the *resource management problem* as it relates to program synthesis: how do we ensure that a synthesised program is not only well-typed but also well-resourced? To address this question, we define two calculi of synthesis rules based on a linear λ-calculus extended with graded modal types which tackle the problem in different ways. To better illustrate and test the practicality of the synthesis calculi, we extend the language with multiplicative conjunction (product types $\otimes$ and unit Unit) and additive disjunction (sum types $\otimes$). These calculi are then implemented targeting default Granule. We produce a comparative evaluation of the implemented tool, contrasting the efficiency of the two resource management approaches against each other, before selecting the most performant to use going forward.

2. Following this, in Chapter 4 we present a synthesis calculus for a target language based instead on the core graded λ-calculus of 2.3.2. This calculus introduces several new language features such as data constructors, pattern matching, and recursive data types, for a rich synthesis tool implementation targeting Granule's *graded base* language extension. Furthermore, we outline several other useful extensions to the synthesis tool, such as the inclusion of example-based synthesis, and a post-synthesis refactoring process which re-writes synthesised programs in a more idiomatic style.

   We then evaluate the implementation on a set of benchmarks, including several non-trivial programs which make use of these new features. In this evaluation, we compare synthesis in a setting which uses grades to prune the search space, to ungraded purely type-directed synthesis. From our evaluation we find that using grades in synthesis outperforms purely type-driven program synthesis in terms of number of paths explored in the search space, number of input-output examples required or number of retries to get the desired program.

To demonstrate the practicality and versatility of our approach, we apply our synthesis algorithm of Chapter 4 to synthesising programs in Haskell from type signatures using GHC's *linear types* extension (which is implemented underneath by a graded type system).

3. Finally, we then consider an alternative approach to type-directed synthesis, exploring a mechanism for automatically deriving programs from their type à la Haskell's generic deriving mechanism [Magalhães et al., 2010]. We base the approach on the graded linear λ-calculus, extended with data constructors, pattern matching, and recursive data types.

This approach strikes a balance between maximising coverage of different approaches to resourceful type systems, and avoiding unnecessary repetition, whilst gradually increasing the complexity of the target language. By the end, we will have two synthesis tool implementations for Granule, targeting both styles of graded systems.

1.4 PUBLICATIONS

In this thesis, the content of some chapters is formed from previously published papers:

- Hughes and Orchard [2020] Resourceful Program Synthesis from Graded Linear Types. In Logic-Based Program Synthesis and Transformation - 30th International Symposium, pp. 151-170.

  This paper builds on the resource management techniques of previous work for Linear Logic proof search to graded types. It presents two synthesis calculi based on these approaches, and evaluates them against each other using a suite of benchmarks of Granule programs.

  This paper constitutes a large part of Chapter 3, where a program synthesis tool for a linear graded type system is introduced.

  I was the primary author of this paper, with my contributions including the adaptation of the resource management model to graded types, the design of the synthesis calculi, the soundness proofs, as well as the implementation and evaluation in Granule.

- Hughes et al. [2020] Deriving Distributive Laws for Graded Linear Types. In 6th edition of the International Workshop on Linearity and of the 4th edition of the International Workshop on Trends in Linear Logic and its Applications.

  The majority of Chapter 5 is derived from this paper, which discusses an approach for automatically deriving certain classes of graded programs in Granule as an alternative to search-based

synthesis. The focus is on deriving distributive laws of graded programs in Granule, but also considers deriving other useful structural combinators. The core contribution of this work is a tool in the vein of Haskell's Generic Type Deriving mechanism [Magalhães et al., 2010], which is implemented as part of the Granule compiler.

My contributions to this paper included the design and implementation of the automated deriving mechanism, as well as proving its soundness and other properties.

- Hughes et al. [2021] Linear Exponentials as Graded Modal Types. In 5th International Workshop on Trends in Linear Logic and Applications.

  In Chapter 5 we make reference to the disparity between Granule's □ modality and Linear Logic's !, and how ! can be recovered in Granule through the use of a grade-level operator defined in this paper.

  My contribution in this paper was as part of the formulation of our solution to recovering Linear Logic's ! and implementing it into Granule's type checker.

- Hughes and Orchard [2024] Program Synthesis from Graded Types. In 33rd European Symposium on Programming.

  This paper presents a program synthesis calculus and implementation for a fully graded type system à la Petricek et al. [2014], and demonstrates how these techniques can be applied to Linear Haskell. This work forms the basis of much of Chapter 4.

  I was the primary author on this paper, responsible for the design of the synthesis calculus, proving its soundness, implementing it into the Granule toolchain, conducting the evaluation, as well as implementing the prototype application to Linear Haskell.

2

BACKGROUND

Before diving straight into designing program synthesis calculi, we first need to formally define our target languages. We take this chapter as an opportunity to do so, and to examine more closely some of the properties of these linear and graded systems.

Since Girard [1987]'s original work on Linear Logic, the development of type systems which convey additional information about the program's structure has evolved into a distinct paradigm, culminating in recent years with the notion of *graded types*. Approaches to graded type systems run the gamut, incorporating a wide range of *effect* and *coeffect* systems, however, they can typically be distilled into two categories, with distinct lineages:

- Systems where a graded modal type operator introduces and eliminates graded modalities above some existing type system. This is the default approach of Granule, where the underlying type system is linear, and grade modalities are introduced and eliminated via a family of $\Box$ modal type operators. We refer to these as "graded linear" or "linear base" systems.

- Systems where grades permeate the program, and are introduced via annotations on function arrows. These systems do away with the underlying linear basis that typifies the former category of systems. This is the approach taken by Linear Haskell [Bernardy et al., 2018], where grades (or "multiplicities") are specified using the `%` operator. We refer to these as "fully graded" or "graded base" systems.

These two different styles to graded types mirror the dual development of effect systems and graded monadic systems in the literature (see Wadler and Thiemann [2003]). In the latter case, the two were eventually found to be equivalent, while the same treatment for the former remains ongoing work.

ROADMAP The remainder of this chapter formally defines these calculi, starting with the linear λ-calculus in Section 2.2.1. Following this, in Section 2.3 we extend the linear λ-calculus with a graded modal

type (Section 2.3.1) before presenting the fully graded λ-calculus (Section 2.3.2). In Section 2.4 we briefly provide a call-by-name operational semantics for our calculi, and note that substitution is admissible in each. We then provide an introduction to Granule syntax in Section 2.5, the programming language which our synthesis tool targets, and which implements the calculi of Sections 2.3.1 and 2.3.2. We include examples of programs typeable in their respective calculi as Granule code. Finally we outline our approach to how we these calculi will be used in synthesis in Section 2.6.

2.1 TERMINOLOGY

Throughout this thesis we will tend towards using a *types-and-programs* terminology rather than *propositions-and-proofs*. Via the Curry-Howard correspondence, one can switch smoothly to viewing our approach to program synthesis as proof search in logic.

The functional programming languages we discuss are presented as typed calculi given by sets of *types*, *terms* (programs), and *typing rules* that relate a term to its type. The most well-known typed calculus is the simply-typed λ-calculus (STLC), which corresponds to the implication fragment of intuitionistic logic. We assume familiarity with STLC, and use this to facilitate our explanation of linear and graded types.

A *judgment* defines the typing relation between a type and a term based on a *context*. In STLC, judgments have the form: $\Gamma \vdash t : A$, stating that under some context of *assumptions* Γ the program term t can be assigned the type A. An assumption is a name with an associated type, written $x : A$ and corresponds to an in-scope variable in a program.

A term can be related to a type if we can derive a valid judgment through the application of typing rules. The application of these rules forms a tree structure known as a *typing derivation*.

2.2 LINEAR AND SUBSTRUCTURAL LOGICS

Linear logic allows one to be more descriptive about the properties of a derivation in intuitionistic logic. In type systems such as STLC, the properties of *weakening*, *contraction*, and *exchange* are assumed implicitly. These are typing rules which are *structural* as they determine how the context may be used rather than being directed by the syntax. Weakening is a rule which allows terms that are not needed in a typing derivation to be discarded. Contraction works as a dual to weakening, allowing an assumption in the context to be used more than once. Finally, exchange allows assumptions in a context to arbitrarily reordered. The rules themselves are provided by Figure 2.1.

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

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

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

Linear logic is thus known as a *substructural* logic because it lacks the weakening and contraction rules, while permitting exchange. The exclusion and inadmissibility of these rules means that in order to construct a typing derivation, each assumption must be used exactly once: arbitrarily copying or discarding hypotheses is disallowed, excluding many programs from being typeable.

As one can imagine, the different combinations of permitted structural rules yields various other substructural logics. Affine logic permits exchange and weakening, but disallows contraction, resulting in system where values may be used *at most* once. Relevant logic only disallows weakening (hypotheses can be used *at least* once), while Ordered logic disallows all three (hypotheses must be used exactly once and in order).

2.2.1 *The Linear λ-Calculus*

Our focus is on the implication fragment of linear logic. By permitting only the exchange structural rule, we can integrate linear logic into STLC, yielding a *linear* $\lambda-$calculus. This provides us with a foundation for programming with substructural logics, which we will later refine with graded types.

The types of the linear λ-calculus are given by the grammar:

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

Like STLC, we have one type former: the function type. Here, however, our function type is a *linear* function arrow (denoted by $\multimap$).

Typing judgments are of the form $\Gamma \vdash t : A$, where Γ ranges over contexts of assumptions:

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

Thus, a context may be empty $\emptyset$, or extended with a linear assumption $x : A$. Throughout, comma denotes disjoint context concatenation as well as context extension. We treat contexts as unordered lists, thus making the exchange rule admissible.

The syntax of terms is the same as STLC, given by:

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

$$\frac{}{x : A \vdash x : A}\ \textsc{Var}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{Abs} \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{App}$$

Figure 2.2: Typing rules for the linear λ-calculus

That is, a term is either a linear variable usage x, an abstraction which binds a linear variable x in the term t, or an application of the argument t_1 to t_2.

Figure 2.2 relates the terms to their types via typing rules. Linear variables are typed in a singleton context (Var). Abstraction (Abs) binds a linear variable which is used to type the premise. Application (App) combines the usages across both premises through the use of *context addition* (2.3.3). Context addition provides an analogue to contraction, combining contexts that have come from typing multiple sub-terms in a rule. Context addition, written $\Gamma_1 + \Gamma_2$, is undefined if Γ_1 and Γ_2 overlap in their assumptions, i.e. a linear assumption may not appear in both sides of the addition. This is equivalent to concatenation on disjoint contexts.

Definition 2.2.1 (Linear context addition)**.**

$$\begin{aligned}
\Gamma + \varnothing &= \Gamma \\
\varnothing + \Gamma &= \Gamma \\
(\Gamma, x : A) + \Gamma' &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma'| \\
\Gamma + (\Gamma', x : A) &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin |\Gamma|
\end{aligned}$$

This leaves us with a simple resourceful typing calculus, which serves as the building block for further refinement.

Example 2.2.1. An example of a program typeable in this calculus is the combinator for function composition:

$$\begin{aligned}
&comp : (A \multimap B) \multimap (B \multimap C) \multimap A \multimap C \\
&comp = \lambda x.\lambda y.\lambda z.y\ (x\ z)
\end{aligned}$$

Here, each argument is used exactly once in body of the innermost lambda, satisfying linearity. In contrast, the K combinator from SKI combinatory logic is not typeable in the linear λ-calculus:

$$\begin{aligned}
&k : A \multimap B \multimap A \\
&k = \lambda x.\lambda y.x
\end{aligned} \qquad \text{(ill-typed)}$$

as y is discarded in the body of the abstraction, violating linearity.

2.3 FROM LINEARITY TO GRADES

Now that we have seen the linear λ-calculus itself, we are ready to generalise the notion of data as a resource inside a program, and formally define a graded type system. We now present two calculi, each based on the differing view of graded types mentioned at the top of this chapter. The first directly follows from the calculus in 2.2.1, merely extending it with a *graded modal type*. The second reflects the other approach to graded type systems where instead of the underlying linear structure, grades are introduced via the function type because all assumptions carry a grade.

2.3.1 *The Graded Linear λ-calculus*

We now define a core type system, based on the linear λ-calculus of section 2.2.1, extended with a graded modal type. This calculus is equivalent to the core calculus of Granule, GrMini [Orchard et al., 2019], as well as the calculi of Brunel et al. [2014], Ghica and Smith [2014], Petricek [2017], Petricek et al. [2014], and Gaboardi et al. [2016] (minus graded monads). Similar approaches have been used in more specialised systems such as the linear dependent type system of Lago and Gaboardi [2012], and the work of Gaboardi et al. [2013], which uses grades to capture and enforce properties of differential privacy.

We refer to the system in this section as the *graded linear λ-calculus*, reflecting the underlying linear structure of the system.

Given a typing judgment $\Gamma \vdash t : A$ we say that t is both *well-typed* and *well-resourced* to highlight the role of grading in accounting for resource use via the semiring information.

This system forms the basis of the target language for our synthesis tool in Chapter 3, although we extend it with some basic types for increased expressivity.

The types of the graded linear λ-calculus are given by:

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

where the type $\Box_r A$ is an indexed family of type operators where r is a *grade* ranging over the elements of a pre-ordered semiring $(\mathcal{R}, \cdot, 1, +, 0, \sqsubseteq)$ parametrising the calculus (where $\cdot$ and $+$ are monotonic operations (which may be partial) with respect to the pre-order $\sqsubseteq$).

From this semiring structure, we can derive two partial operations on grades which will be useful later on when considering sum types: *greatest lower bounds* ($\sqcap$), and *least upper bounds* ($\sqcup$). These are given by Definitions 2.3.1 and 2.3.2, respectively.

Definition 2.3.1 (Partial greatest-lower bound of grades)**.** For two grades r and s of a given semiring, $r \sqcap s$ is defined in terms of the

semiring's preorder: $r \sqcap s = t$ if $t \sqsubseteq r$ and $t \sqsubseteq s$ and there exists no other t' where $t' \sqsubseteq r$, $t' \sqsubseteq s$ and $t \sqsubseteq t'$

Definition 2.3.2 (Partial least-upper bound of grades)**.** For two grades r and s of a given semiring, $r \sqcup s$ is defined in terms of the semiring's preorder: $r \sqcup s = t$ if $r \sqsubseteq t$ and $s \sqsubseteq t$ and there exists no other t' where $r \sqsubseteq t'$, $s \sqsubseteq t'$ and $t' \sqsubseteq t$

As before, typing judgments are of the form $\Gamma \vdash t : A$, where Γ ranges over contexts:

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

Thus, a context may also be extended with a graded assumption $x :_r A$. For linear assumptions, structural rules of weakening and contraction are disallowed. Graded assumptions may be used non-linearly according to the constraints provided by their grade, the semiring element r.

Various operations on contexts are used to capture non-linear data flow via grading. As with the linear λ-calculus, context addition (2.3.3) combines the contexts used to provide multiple sub-terms. However, our new definition includes an extra case for dealing with graded assumptions appearing in both contexts, which are combined via the semiring $+$ of their grades. Note that this is again a partial definition where addition may be undefined.

Definition 2.3.3 (Graded linear context addition)**.**

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

Note that this is a declarative specification of context addition. Graded assumptions may appear in any position in Γ and Γ' as witnessed by the algorithmic specification where for all Γ_1, Γ_2 context addition is defined as follows by ordered cases matching inductively on the structure of Γ_2, where $\Gamma_1 + \Gamma_2 =$

$$\begin{cases}
\Gamma_1 & \Gamma_2 = \emptyset \\
(\Gamma_1 + \Gamma_2'), x :_r A & \Gamma_2 = \Gamma_2', x :_r A \ \wedge \ x :_r A \notin \Gamma_1 \\
((\Gamma_1', \Gamma_1'') + \Gamma_2'), x :_{(r+s)} A & \Gamma_2 = \Gamma_2', x :_s A \wedge \Gamma_1 = \Gamma_1', x :_r A, \Gamma_1'' \\
(\Gamma_1 + \Gamma_2'), x : A & \Gamma_2 = \Gamma_2', x : A \ \wedge \ x : A \notin \Gamma_1
\end{cases}$$

$$\frac{}{x : A \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{Abs}$$

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

$$\frac{\Gamma \vdash t : A}{\Gamma, [\Delta]_0 \vdash t : A}\ \textsc{Weak} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x :_1 A \vdash t : B}\ \textsc{Der}$$

$$\frac{\Gamma, x :_r A, \Gamma' \vdash t : A \quad r \sqsubseteq s}{\Gamma, x :_s A, \Gamma' \vdash t : A}\ \textsc{Approx}$$

$$\frac{[\Gamma] \vdash t : A}{r \cdot [\Gamma] \vdash [t] : \Box_r A}\ \textsc{Pr} \qquad \frac{\Gamma_1 \vdash t_1 : \Box_r A \quad \Gamma_2, x :_r A \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 : B}\ \textsc{Let}\Box$$

Figure 2.3: Typing rules of the graded linear λ-calculus

For example, the context addition of $x :_4 A, y :_3 B$ and $x :_2 A, y :_5 B$ would yield a context $x :_6 A, y :_8 B$ (using the definition of $+$ from the natural numbers semiring). However, the context addition of $x : A, y :_2 B$ and $x : A, y :_3 B$ is undefined, as the linear assumption x is present in both sides of the addition.

The terms of the language are given by:

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

In addition to the the terms of the linear λ-calculus, we also have the construct $[t]$ which introduces a graded modal type $\Box_r A$ by 'promoting' a term t to the graded modality, and it's dual $\mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2$ eliminates a graded modal value t_1, binding a graded variable x in scope of t_2. The typing rules relate these terms to types.

Figure 2.3 gives the typing rules. The Weak rule captures weakening of assumptions graded by 0, where $[\Delta]_0$ denotes a context containing only graded assumptions graded by 0, a predicate on contexts given by Definition 2.3.4:

Definition 2.3.4 (Zeroed graded contexts)**.** For all contexts Γ, $[\Gamma]_0$ is defined:

$$[\emptyset]_0 = \emptyset \qquad [\Gamma, x :_0 A]_0 = [\Gamma]_0, x :_0 A$$
$$[\Gamma, x :_r A]_0 = \bot \qquad [\Gamma, x : A]_0 = \bot$$

Context addition and Weak together therefore provide the rules of substructural rules of contraction and weakening for graded variables. Dereliction (Der), allows a linear assumption to be converted to a graded assumption with grade 1.

We allow grade *approximation* via their resource algebras through the application of the APPROX rule. A grade s may be converted to another grade r, providing that r iw *approximated* by s, where the relation $\sqsubseteq$ is the pre-order provided with the semiring. This relation is occasionally lifted pointwise to contexts, ignoring linear assumptions: we write $\Gamma \sqsubseteq \Gamma'$ to mean that Γ' overapproximates Γ meaning that for all $(x :_r A) \in \Gamma$ then $(x :_{r'} A) \in \Gamma'$ and $r \sqsubseteq r'$.

Introduction and elimination of the graded modality is provided by the PR and LET rules respectively. The PR rule propagates the grade r to the assumptions through *scalar multiplication* of Γ by r where every assumption in Γ must already be graded. Definition 2.3.5 defines a predicate on Γ in the form of a partial operation which ensures that Γ contain only graded variables (written $[\Gamma]$):

Definition 2.3.5 (Solely graded contexts)**.** For all contexts Γ, $[\Gamma]$ is defined:

$$[\emptyset] = \emptyset \qquad [\Gamma, x :_r A] = [\Gamma], x :_r A \qquad [\Gamma, x : A] = \bot$$

When an assumption in $[\Gamma]$ is graded, the identity is returned, otherwise (i.e. if an assumption is linear) the operation is undefined.

Scalar multiplication is then given by: Definition 2.3.6.

Definition 2.3.6 (Scalar context multiplication)**.** A context which consists solely of graded assumptions, i.e. $[\Gamma]$, can be multiplied by a semiring grade $r \in \mathcal{R}$

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

The LET rule eliminates a graded modal value $\Box_r A$ into a graded assumption $x :_r A$ with a matching grade in the scope of the **let** body. This is also referred to as "unboxing".

We give an example of graded modalities using a graded modality indexed by the semiring of natural numbers.

Example 2.3.1. The natural number semiring with discrete ordering $(\mathbb{N}, \cdot, 1, +, 0, \equiv)$ provides a graded modality that counts exactly how many times non-linear values are used. As an example, the S combinator from the SKI system of combinatory logic is typed and defined:

$$s : (A \multimap (B \multimap C)) \multimap (A \multimap B) \multimap \Box_2 A \multimap C$$
$$s = \lambda x.\lambda y.\lambda z'.\mathbf{let}\,[z] = z'\,\mathbf{in}\,(x\,z)\,(y\,z)$$

The graded modal value z' captures the capability for a value of type A to be used twice. This capability is made available by eliminating $\Box$ (via **let**) to the variable z, which has grade 2 in the scope of the body. Likewise, we can now correctly type and write the K combinator from Section 2.2.1 as:

$$k : A \multimap \Box_0 B \multimap A$$
$$k = \lambda x.\lambda y'.\mathbf{let}\,[y] = y'\,\mathbf{in}\,x$$

Example 2.3.2. The ! modality can be (almost) recovered via the {Zero, One, ω} semiring. For this semiring, we define $+$ as $r + s = r$ if $s = \mathsf{Zero}$, $r + s = s$ if $r = \mathsf{Zero}$, otherwise ω. Multiplication is $r \cdot \mathsf{Zero} = \mathsf{Zero} \cdot r = \mathsf{Zero}$, $r \cdot \omega = \omega \cdot r = \omega$ (where $r \neq \mathsf{Zero}$), and $r \cdot 1 = 1 \cdot r = r$. Ordering is defined as $\mathsf{Zero} \sqsubseteq \omega$ and $\mathsf{One} \sqsubseteq \omega$. This semiring allows us to express both linear and non-linear usage of values, with a One grade indicating linear use, Zero requires the value be discarded, and ω acting as linear logic's ! and permitting unrestrained use. This is similar to Linear Haskell's multiplicity annotations (although Linear Haskell has no equivalent of a Zero grade, only having One and Many annotations) [Bernardy et al., 2018]. Using this semiring, we can write the *K* combinator from Example 2.3.1 above in Granule as:

```
k : ∀ { a b : Type } . a [One] → b [Zero] → a
k x' y' = let [x] = x' in let [y] = y' in x
```

Note however that some additional restrictions are required on typing to get exactly the behaviour of ! with respect to products [Hughes et al., 2021]. This is an orthogonal discussion and not relevant to the rest of this work.

2.3.2 *The Fully Graded λ-calculus*

As mentioned before, the second style of graded cacluli that we consider are *fully graded* systems, such as those used in practical systems today. For example, this is the approach taken by Idris 2 [Brady, 2021] and the linear types extension to Haskell [Bernardy et al., 2018].

We now define a core calculus for such a type system, where grades permeate the entire program, drawing from the coeffect calculus of Petricek et al. [2014], Quantitative Type Theory (QTT) by McBride [2016] and refined further by Atkey [2018], the calculus of Abel and Bernardy [2020], and other graded dependent type theories, such as Moon et al. [2021] (although we omit dependent types from our language). We refer to this system as the *fully graded λ-calculus*.

The syntax of types is given by:

$$A, B ::= A^r \to B \mid \Box_r A \qquad \textit{(types)}$$

where the function arrow $A^r \to B$ annotates the input type with a *grade* r which is again drawn from a pre-ordered semiring $(\mathcal{R}, \cdot, 1, +, 0, \sqsubseteq)$ parametrising the calculus. The graded necessity modality $\Box_r A$ is similarly annotated by the grade r being an element of the semiring.

Typing judgements have the same form as Section 2.3.1, however, variable contexts are instead given by:

$$\Delta, \Gamma ::= \emptyset \mid \Gamma, x :_r A \qquad \textit{(contexts)}$$

$$\frac{}{0 \cdot \Gamma, x :_1 A \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma, x :_r A \vdash t : B}{\Gamma \vdash \lambda x.t : A^r \to B}\ \textsc{Abs}$$

$$\frac{\Gamma_1 \vdash t_1 : A^r \to B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + r \cdot \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{App}$$

$$\frac{\Gamma, x :_r A, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x :_s A, \Gamma' \vdash t : B}\ \textsc{Approx} \qquad \frac{\Gamma \vdash t : A}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{Pr}$$

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

Figure 2.4: Typing rules for the fully graded λ-calculus

That is, a context may be empty $\emptyset$ or extended with a *graded* assumption $x :_r A$, which must be used in a way which adheres to the constraints of the grade r. Note that we no longer include linear assumptions as part of our contexts. As before, structural exchange is permitted, allowing a context to be arbitrarily reordered.

The syntax of terms is given as:

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

As before, terms comprise the λ-calculus, extended with the *promotion* construct [t] as seen in Section 2.3.1.

Figure 2.4 gives the full typing rules, which explains the meaning of the syntax with reference to their static semantics.

Variables (rule Var) are typed in a context where the variable x has grade 1 denoting its single usage here. All other variable assumptions are given the grade of the 0 semiring element (providing weakening), using *scalar multiplication* of contexts by a grade, re-using Definition 2.3.6 from Section 2.3.1.

Abstraction (Abs) captures the assumption's grade r onto the function arrow in the conclusion, that is, abstraction binds a variable x which may be used in the body t according to grade r. Application again (App) makes use of context addition to combine the contexts used to type the two sub-terms in the premises of the application rule:

Definition 2.3.7 (Graded context addition)**.**

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

As in the linear graded case, we also provide the algorithmic definition of graded context addition, where graded assumptions may appear in any position in Γ and Γ' as witnessed by the algorithmic specification where for all Γ_1, Γ_2 context addition is defined as follows by ordered cases matching inductively on the structure of Γ_2, where $\Gamma_1 + \Gamma_2 =$

$$\begin{cases} \Gamma_1 & \Gamma_2 = \emptyset \\ (\Gamma_1 + \Gamma_2'), x :_r A & \Gamma_2 = \Gamma_2', x :_r A \ \wedge \ x :_r A \notin \Gamma_1 \\ ((\Gamma_1', \Gamma_1'') + \Gamma_2'), x :_{(r+s)} A & \Gamma_2 = \Gamma_2', x :_s A \wedge \Gamma_1 = \Gamma_1', x :_r A, \Gamma_1'' \end{cases}$$

Note that Definition 2.3.7 differs only from 2.3.3, in that the former need not consider linear assumptions.

As an example of the App rule, consider the following typing derivation for a program of type $(A^3 \to B)^1 \to A^3 \to B$:

$$\dfrac{\dfrac{\dfrac{}{x :_1 A^3 \to B \vdash x : A^3 \to B}\textsc{Var} \quad \dfrac{}{y :_1 A \vdash y : A}\textsc{Var}}{x :_1 A^3 \to B + 3 \cdot y :_1 A \to B \vdash x\ y : B}\textsc{App}}{x :_1 A^3 \to B \vdash \lambda y.x\ y : A^3 \to B}\textsc{Abs}}{\emptyset \vdash \lambda x.\lambda y.x\ y : (A^3 \to B)^1 \to A^3 \to B}\textsc{Abs}$$

This program binds a function x of type $A^3 \to B$ with grade 1, i.e. it consumes its input according to a grade of 3 and the function itself can be used once. The application of y to x satisfied this grade usage, as y has a grade of 3. In the App rule, we type a single use of y in the second premise via the Var rule, and then scale this usage by the grade 3 on the function arrow of x, satisfying the grade y was bound with in the Abs rule.

Explicit introduction of graded modalities is achieved via the rule for promotion (Pr). This rule is almost identical to that of 2.3.1 with the only difference being here Γ is known to always contain only graded assumptions, so the predicate $[\Gamma]$ is not needed. Explicit unboxing (Let$\Box$), and approximation (Approx) are likewise identical to the calculus of 2.3.1.

Example 2.3.3. We now consider the programs shown in Examples 2.2.1, 2.3.1 (page 15) but written instead in this fully graded style. The combinator for linear function composition is written:

$$comp : (A^1 \to B)^1 \to (B^1 \to C)^1 \to A^1 \to C$$
$$comp = \lambda x.\lambda y.\lambda z = y\ (x\ z)$$

Note that at the term level, we do not need to unbox values graded by a function arrow to use them in according to their grades. However,

we may make explicit use of graded modalities which does necessitate unboxing. Likewise, S and K can be typed and written as:

$$s : (A^1 \to (B^1 \to C))^1 \to (A^1 \to B)^1 \to A^2 \to C$$
$$s = \lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z)$$

$$k : A^1 \to B^0 \to A$$
$$k = \lambda x.\lambda y.x$$

2.4 OPERATIONAL SEMANTICS

We briefly present the reduction rules for a call-by-name operational semantics for our calculi [Liepelt et al., 2024]. The Granule interpreter also provides a call-by-value semantics which can be enabled via a language extension. Note that the rules are identical for both the graded linear and fully graded systems! Figure 2.5 collects the rules.

$$\frac{}{(\lambda x.t_2)\, t_1 \rightsquigarrow [t_1/x]t_2}\ \beta \qquad \frac{t_1 \rightsquigarrow t_1'}{t_1\, t_2 \rightsquigarrow t_1'\, t_2}\ \textsc{App}$$

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

$$\frac{t_1 \rightsquigarrow t_1'}{\mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \rightsquigarrow \mathbf{let}\,[x] = t_1'\,\mathbf{in}\,t_2}\ \textsc{Let}\Box$$

Figure 2.5: Reduction rules for the linear graded and fully graded λ-calculi

The rules are fairly straightforward and we do not rely on them heavily in subsequent chapters, instead providing them for completeness and to give an intuition of the dynamic behaviour of our calculi.

ADMISSIBILITY OF SUBSTITUTION We remark that substitution is admissible in each of the calculi. Rather than provide the details at this point, we present the lemmas for both of the calculi once we have extended them with the additional types mentioned above in their respective chapters.

2.5 THE GRANULE PROGRAMMING LANGUAGE

Having defined the core calculi used throughout this thesis, we now offer a brief introduction to the programming language Granule - the target language of our implementation. Throughout this section, we include the Granule equivalent code of many of the examples seen in Sections 2.3.1 and 2.3.2.

Granule comes in two variants, "linear base" (the default) and "graded base" (enabled via a language extension of the same name) which respectively correspond to the calculi of the two previous sections. We will first introduce Granule syntax in the context of the linear base variant of Granule before showing the alternative style in Section 2.5.3.

Granule syntax is inspired by Haskell. However, a single colon is used for typing rather than double. Note that Granule code uses $\rightarrow$ syntax rather than $\multimap$ for linear functions for the sake of familiarity with standard functional languages.

By default, Granule uses a linear type system at its core. For variables, this means that a function must either return a variable unchanged or pass it to another linear function. If the variable has a function type, then appling a value to that variable also counts as a use. Consider the following (well-typed) example of a program in Granule, the identity function:

```
id : ∀ { a : Type } . a → a
id x = x
```

Here `id` binds a variable `x` on the left, and is simply returned on the right, adhering to linearity. Granule features ML style polymorphism, thus top level definitions are annotated with type schemes which quantify over kind-annotated type variables.

But what happens if we try to write a program which violates this linearity? Consider the following equivalent Granule code of the k combinator from Example 2.3.1:

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

If we try to compile this program, we get a type error

```
Linearity error: Linear variable y is never used.
```

as the discarding of `y` violates linearity. Instead, we can accommodate non-linear behaviour through the use of grades (as seen in the calculus of Figure 2.3):

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

In contrast to the calculus, in Granule the grade is written post-fix on the type, i.e. $\Box_0 B$ becomes `b [0]`. A term-level unboxing via the `let` construct allows us to use the variable `y'` according to the grade `0` in the body of `k`. If we try use the values in a way which violates their grades, we also get a type error at compilation. For example if we changed the grade of `b` in the above example to `1`, the fact that `y` is discarded in the implementation would yield the following error:

```
Falsifiable theorem:
When checking ‘k‘, expected 1 uses, but instead there are (0:
     Nat) actual uses.
```

Likewise, the s combinator from Example 2.3.1 can be written in Granule code as:

```
s : ∀ { a b c : Type }
  . (a → (b → c)) → (a → b) → a [2] → c
s x y z' = let [z] = z' in (x z) (y z)
```

Granule also includes pattern matching (which we extend our calculi with in Chapter 4), allowing us to refactor the above two programs to pattern match on the graded modalities in the function equations:

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

s : ∀ { a b c : Type }
  . (a → (b → c)) → (a → b) → a [2] → c
s x y [z] = (x z) (y z)
```

2.5.1 *Semiring Polymorphism*

While our calculus in Figure 2.3 is parametrised by a semiring making grades monomorphic, in Granule code we may be explicitly polymorphic in the grades used by quantifying over the semiring. For example, we can then write a graded version of `comp` from Section 2.2.1 (i.e. coKleisli composition) as:

$$\begin{aligned} &\mathit{comp\text{-}coK}_{\mathcal{R}} : \Box_r(\Box_s A \multimap B) \multimap (\Box_r B \multimap C) \multimap \Box_{r \cdot s} A \multimap C \\ &\mathit{comp\text{-}coK}_{\mathcal{R}} = \lambda x.\lambda y.\lambda z.\mathbf{let}[u] = x \ \mathbf{in}\ \mathbf{let}[v] = z \ \mathbf{in}\ y\ [u\ [v]] \end{aligned}$$

which we can write as a Granule program which is polymorphic with respect to grades as well as types:

```
compCoK : ∀ {k : Semiring, r s : k, a b c : Type}
     . (a [s] → b) [r]
     → (b [r] → c)
     → a [r * s]
     → c
compCoK x y z = let [u] = x in let [v] = z in y  [ u [v] ]
```

The type scheme for `compCoK` quantifies over the semiring `k`, and then grade variables `r` and `s` with the kind `k`. The grades must then be used in the type in a polymorphic way. This gives a brief overview of Granule syntax, and how the calculus of Figure 2.3 corresponds to Granule code. We now consider the Granule equivalent of the fully-graded λ-calculus, presented in Section 2.3.2.

2.5.2 *Data Types and Constructors*

Granule allows (Generalised) Algebraic Data Types as in ML/Haskell. We omit discussion of GADTS for this work and focus instead on ADTs, which can be written like so:

```
data Maybe a = Nothing; Just a
```

To unwrap a `Maybe a` value to an `a`, we need to provide a default value (making use of an interval grade) in case the value of `Maybe a` is a `Nothing`:

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

Here, we also make use of a wildcared pattern (`_`) in the second function equation which indicates are intent to discard the value being unboxed. The type checker then checks that the grade of `a` allows us to do this discarding (which in this case is permissible).

Recursive ADTs can be constructed as usual, with a `List` data type having the definition:

```
data List a = Nil; Cons a (List a)
```

2.5.3 *The GradedBase Language Extension*

To write fully graded Granule programs, we enable the `GradedBase` language extension. This allows us to write programs in Granule which correspond to the fully-graded λ-calculus of Section 2.3.2. Grades are written on function arrows using `%` (in the same way that *multiplicity* annotations are written in Linear Haskell). This can be seen in the *GradedBase* equivalent of the program `comp` from Example 2.2.1 (using natural number grades):

```
language GradedBase

comp : ∀ { a b c : Type }
     . (a %1 → b) %1
     → (b %1 → c) %1
     → a %1
     → c
comp x y z = y (x z)
```

Omitting a grade annotation defaults to a grade of 1. So this function may also be written without any annotations with the same meaning:

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

The semiring polymorphic version of `comp` in `GradedBase` Granule, would be written as:

```
compCoK : ∀ {k : Semiring, r s : k, a b c : Type }
        . (a % s → b) % n
        → (b % r → c)
        → a %(r * s)
        → c
compCoK x y z = y (x z)
```

2.5.4 *Granule Syntax for Program Synthesis*

To synthesise an implementation for a function in Granule, a programmer can use a typed program hole in place of a function body (written `?`), i.e.:

```
data List a = Nil; Cons a (List a)

tail : ∀ { a : Type } . List a [0..1] → List a
tail = ?
```

Invoking the synthesis tool on a Granule file will attempt to synthesise an implementation for each program hole, and replace the hole with the synthesised implementation.

When specifying the synthesis context of top-level definitions in Granule, the user may supply a series of input-output examples showcasing desired behaviour. Our approach to examples is deliberately naïve; we evaluate a fully synthesised candidate program against the inputs and check that the results match the corresponding outputs. Unlike many sophisticated example-driven synthesis tools, the examples here do not themselves influence the search procedure, and are used solely to allow the user to clarify their intent. This lets us consider the effectiveness of basing the search primarily around the use of grade information. An approach to synthesis of resourceful programs with examples closely integrated into the search as well is further work.

We augmented the Granule language with first-class syntax for specifying input-output examples, both as a feature for aiding synthesis but also for aiding documentation that is type checked (and therefore more likely to stay consistent with a code base as it evolves). Synthesis specifications are written in Granule directly following a function's type scheme using the `spec` keyword. The input-output examples are then listed per-line. Annotating a function with a `spec` construct is optional, but incredibly useful in a synthesis setting:

```
data List a = Nil; Cons a (List a)

tail : ∀ { a : Type } . List a [0..1] → List a
spec
  tail (Cons 1 Nil) = Nil;
  tail (Cons 1 (Cons 2 Nil)) = Cons 2 Nil;
tail = ?
```

Any synthesised term must then behave according to the supplied examples. This `spec` structure can also be used to describe additional synthesis components that the user wishes the tool to make use of. These components comprise a list of in-scope definitions separated by commas. The user can choose to annotate each component with a grade, describing the required usage in the synthesised term. This

defaults to a 1 grade if not specified. For example, the specification for a function which returns the length of a list might be (in `GradedBase`):

```
language GradedBase

data List a where Nil; Cons a (List a)

length : ∀ { a : Type } . List a %0..∞ → N
spec
    length Nil = Z;
    length (Cons 1 Nil) = S Z;
    length (Cons 1 (Cons 1 Nil)) = S (S Z);
    length 0..∞
length = ?
```

with the following resulting program produced by our synthesis algorithm (on average in about 400ms on a standard laptop, see Section 4.5 where this is one of the benchmarks for evaluation):

```
length Nil = Z;
length (Cons y z) = S (length z)
```

2.6 TWO TYPING CALCULI

Having outlined the two lineages of graded type systems, we are left with the question: what approach should we use as the basis of a target language for a program synthesis tool? Both systems embed properties for reasoning about program structure into the language, however, they differ in how this information is expressed, as shown by the variance in typing and syntax between Sections 2.3.1 and 2.3.2.

Rather than focus entirely on one approach, we opt to instead build synthesis tools which target both systems. Our approach to synthesis is rooted in the techniques for resource management in automated theorem proving for linear logic [Hodas and Miller, 1994, Cervesato et al., 2000]. Therefore, Chapter 3 uses the graded linear λ-calculus as a natural starting point for the design of a synthesis calculus, where we build upon the existing proof search literature to accommodate graded modal types. In doing so we also extend graded linear λ-calculus with some other useful types to make our language more practical, such as linear multiplicative products $\otimes$, additive sums $\oplus$, and a Unit type.

Having established how to handle synthesis in the graded linear setting, in Chapter 4 we then pivot to a synthesis calculi for a language derived from the fully graded λ-calculus, as this is the approach that is most commonly in use by practical systems today [Brady, 2021, Bernardy et al., 2018]. In doing so, we also extend our fully graded λ-calculus with recursive ADTs, recursive function definitions, and polymorphism. As we have seen, systems based on both approaches are in use today, and both pose their unique challenges in design-

ing a synthesis tool. Furthermore, the target programming language Granule of our implementations includes both approaches.[1]

1 As of Granule v0.9.3.0. Available at: https://github.com/granule-project/granule/releases/tag/v0.9.3.0

3

A CORE SYNTHESIS CALCULUS

We begin our first exploration into program synthesis with a synthesis system for the graded linear λ-calculus of Section 2.3.1. The primary aim of this chapter is to introduce the core concepts of type-directed program synthesis in a resourceful setting, in particular, the problem of *resource management*. We therefore prioritise simplicity over expressivity for our target language, with the core typing calculus of Section 2.3.1 forming an ideal candidate. Much of the work in this chapter is derived from Hughes and Orchard [2020].

In Chapter 1 we posed the question "how do we harness [linear and graded types] to make writing programs automatically easier?". This chapter goes some way to providing an answer, by showing how we can integrate these types into the design of a synthesis tool, using resource constraints to prune the search space of programs.

As mentioned in Chapter 1, type-directed program synthesis can be framed as an inversion of type checking. In type checking, we have a judgement of the form:

$$\Gamma \vdash t : A \qquad \text{(type checking)}$$

which states that under some context of assumptions Γ we can assign the program term t the type A. In type-checking, we typically think of starting with a term and generating its type. Synthesis inverts this interpretation, leaving us with a *synthesis judgement* form:

$$\Gamma \vdash A \Rightarrow t \qquad \text{(synthesis)}$$

which states that we can construct a program term t from the type A, using the assumptions in Γ. Program synthesis then becomes a task of inductively enumerating programs "bottom-up" starting from the goal type A: we break A into sub-goals, from which sub-terms are synthesised until the goal cannot be broken into further sub-goals. At this point, we either synthesise a usage of a variable from Γ if possible, a constructor for a null-ary data type if one is available, or synthesis fails. This is the essence of type-directed program synthesis.

ROADMAP Resourceful types introduce another dimension to synthesis: how do we ensure that the assumptions in Γ are used according

to their resource constraints as described by their linearity or grades in the synthesised term t? I.e. if $x : A$ is a linear assumption in Γ that is used in some way to construct t, then the synthesis tool must synthesise a t which uses x exactly once. Likewise, if $x :_r A$ is a graded assumption, then it must be used in t in a way which satisfies its grade r.

Firstly, in Section 3.1 we extend our calculus from Section 2.3.1 to include types that help us to write more useful programs which properly convey the problem of resource management. The linear side of this problem has been explored before in the context of automated theorem proving for linear logic, and has been termed the *resource management problem*. We describe this problem in detail in Section 3.2 and propose two candidate solutions which incorporate grades as well as linearity, basing our approach on the *input-output context management* model described by Hodas and Miller [1994], and further developed by Cervesato et al. [2000].

The challenges posed by ensuring the well-resourcedness of synthesised programs are best illustrated by the inclusion in our target language of multiplicative conjunction, and additive disjunction. Therefore, prior to fully describing the problem of resource management and our proposed solutions, we first expand our target language with multiplicative product ($\otimes$), and unit types (Unit), as well as disjunctive sum types ($\oplus$). These extensions are detailed in Section 3.1, which will be the target language of the synthesis calculi of this chapter. As well as helping to conceptualise the challenges posed by program synthesis in a resourceful setting, these have the added benefit of allowing the synthesis of more expressive programs, without introducing unnecessary complexity at this stage.

Having outlined both a suitable target language and two approaches to dealing with the issue of resource management, we then present two synthesis calculi in Sections 3.3 and 3.4 as augmented inversions of the typing rules. Each calculus is based on a one of our proposed solutions to the resource management problem.

Both calculi are implemented as a synthesis tool for Granule.[1] The calculi are turned into an algorithm written in Haskell. In order to do so, we apply an important technique from proof search literature to our calculi: *focusing* [Andreoli, 1992]. Focusing removes much of the unnecessary non-determinism present in our synthesis rules by fixing an ordering on the application of rules. We present the two *focused* forms of our original synthesis calculi in Section 3.5, which form the basis of our Granule implementation. Following this, we provide some details of the implementation in Section 3.6

1 The exact implementation of the rules as they appear in this thesis is deprecated, but may be found in Granule release v0.7.8.0: https://github.com/granule-project/granule/releases/tag/v0.7.8.0

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

$$\frac{\Gamma_1 \vdash t_1 : A \otimes B \quad \Gamma_2, x_1 : A, x_2 : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x_1, x_2) = t_1\,\mathbf{in}\,t_2 : C}\ \textsc{LetPair}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathbf{inl}\,t : A \oplus B}\ \textsc{Inl} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathbf{inr}\,t : A \oplus B}\ \textsc{Inr}$$

$$\frac{\Gamma_1 \vdash t_1 : A \oplus B \qquad \Gamma_2, x_1 : A \vdash t_2 : C \qquad \Gamma_3, x_2 : B \vdash t_3 : C}{\Gamma + (\Gamma_2 \sqcup \Gamma_3) \vdash\ \mathbf{case}\ t_1\ \mathbf{of\ inl}\ x_1 \to t_2;\ \mathbf{inr}\ x_2 \to t_3 : C}\ \textsc{Case}$$

$$\frac{}{\varnothing \vdash () : \mathsf{Unit}}\ \mathsf{Unit} \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{Unit} \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : A}\ \textsc{LetUnit}$$

Figure 3.1: Typing rules of for $\otimes$, $\oplus$, and 1

Each synthesis calculus implementation is then evaluated and contrasted against each other in Section 3.7, which measures the synthesis calculi on a suite of simple Granule benchmark programs. Finally we conclude with a discussion of related work in Section 6.1, and highlight some limitations of our system in Section 3.8, showing how we intend to proceed in subsequent chapters to address these.

3.1 A CORE TARGET LANGUAGE

The syntax of types and terms for our extended language are given by the following two grammars:

$$\begin{aligned} A, B &::= A \multimap B \mid A \otimes B \mid A \oplus B \mid \mathsf{Unit} \mid \Box_r A && \text{(types)} \\ t &::= x \mid \lambda x.t \mid t_1\, t_2 \\ &\quad \mid [t] \mid \mathbf{let}\,[x] = t_1\,\mathbf{in}\,t_2 \\ &\quad \mid (t_1, t_2) \mid \mathbf{let}\,(x_1, x_2) = t_1\,\mathbf{in}\,t_2 \\ &\quad \mid () \mid \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 \\ &\quad \mid \mathbf{inl}\,t \mid \mathbf{inr}\,t \mid \mathbf{case}\ t_1\ \mathbf{of\ inl}\ x_1 \to t_2;\ \mathbf{inr}\ x_2 \to t_3 && \text{(terms)} \end{aligned}$$

Type formers comprise the graded linear λ-calculus of Section 2.3.1, extended with multiplicative products ($\otimes$), additive coproducts ($\oplus$), and multiplicative units Unit.

We use the syntax $()$ for the inhabitant of Unit. Pattern matching via a **let** is used to eliminate products and unit types; for sum types, **case** is used to distinguish the constructors.

Figure 3.1 gives the typing rules. Rules for multiplicative products (pairs) and additive coproducts (sums) are routine, where pair intro-

duction (PAIR) adds the contexts used to type the pair's constituent sub-terms. Pair elimination (LETPAIR) binds a pair's components to two linear variables in the scope of the body t_2. The INL and INR rules handle the typing of constructors for the sum type $A \oplus B$. Elimination of sums (CASE) takes the *least upper bound* of the two contexts used to type each branch, defined:

Definition 3.1.1 (Partial least-upper bounds of contexts)**.** For all Γ_1, Γ_2, $\Gamma_1 \sqcup \Gamma_2 =$

$$\begin{cases} \emptyset & \Gamma_1 = \emptyset & \wedge\ \Gamma_2 = \emptyset \\ (\Gamma_1' \sqcup (\Gamma_2', \Gamma_2'')), x : A & \Gamma_1 = \Gamma_1', x : A & \wedge\ \Gamma_2 = \Gamma_2', x : A, , \Gamma_2'' \\ (\Gamma_1' \sqcup (\Gamma_2', \Gamma_2'')), x :_{r \sqcup s} A & \Gamma_1 = \Gamma_1', x :_r A & \wedge\ \Gamma_2 = \Gamma_2', x :_s A, \Gamma_2'' \end{cases}$$

where $r \sqcup s$ is the least-upper bound of grades r and s if it exists, derived from $\sqsubseteq$ (given by Definition 2.3.2).

As an example of the partiality of $\sqcup$, if one branch of a **case** uses a linear variable, then the other branch must also use it to maintain linearity overall, otherwise the upper-bound of the two contexts for these branches is not defined.

With these extensions in place, we now have the capacity to write more idiomatic functional programs in our target language. As a demonstration of this, and to showcase how graded modalities interact with these new type extensions, we provide two further examples of different graded modalities which complement these new types.

Example 3.1.1. Exact usage analysis is less useful when control-flow is involved, e.g., eliminating sum types where each control-flow branch uses variables differently. The $\mathbb{N}$-semiring shown on page 19 can be imbued with a notion of *approximation* via less-than-equal ordering, providing upper bounds. A more expressive semiring is that of natural number intervals [Orchard et al., 2019], given by pairs $\mathbb{N} \times \mathbb{N}$ written $r...s$ here for the lower-bound $r \in \mathbb{N}$ and upper-bound usage $s \in \mathbb{N}$ with $0 = 0...0$ and $1 = 1...1$, addition and multiplication defined point-wise, and ordering $r...s \sqsubseteq r'...s' = r' \leq r \wedge s \leq s'$. Thus a coproduct elimination function can be written and typed:

$$\begin{aligned} &\oplus_e : \Box_{0...1}(A \multimap C) \multimap \Box_{0...1}(B \multimap C) \multimap (A \oplus B) \multimap C \\ &\oplus_e = \lambda x'.\lambda y'.\lambda z.\mathbf{let}\ [x] = x'\ \mathbf{in} \\ &\qquad\qquad \mathbf{let}\ [y] = y'\ \mathbf{in} \\ &\qquad\qquad\quad \mathbf{case}\ z\ \mathbf{of}\ \mathbf{inl}\ u \to x\ u \mid \mathbf{inr}\ v\ \to y\ v \end{aligned}$$

Here, $\oplus_{0...1}$ takes two functions as arguments which each have a return type C and are graded by the interval $0...1$, as well as a linear sum of type $A \oplus B$. The sum is eliminated via a **case** statement, where each branch uses the appropriate argument function to form

an application with type C. Despite each branch only using one of the argument functions, the program is well-resourced thanks to the interval grade $0...1$ giving us the ability to discard the irrelevant function.

Example 3.1.2. Graded modalities can capture a form of information-flow security, tracking the flow of labelled data through a program [Orchard et al., 2019], with a 2-point semiring of security levels where $\mathcal{L} = \{\text{Private}, \text{Public}\}$ forms a set of abstract labels, denoting *high* and *low* security permissions respectively, with a lattice formed by the total order with Private $\sqsubseteq$ Public. Multiplication is given by $\sqcup$, and addition by $\sqcap$, with 0 = Private and 1 = Public.

This allows the following well-typed program, eliminating a pair of Lo and Hi security values, picking the left one to pass to a continuation expecting a Lo input:

$$
\begin{aligned}
&\mathit{noLeak} : (\Box_{\mathsf{Lo}} A \otimes \Box_{\mathsf{Hi}} A) \multimap (\Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B) \multimap B \\
&\mathit{noLeak} = \lambda z.\lambda u.\mathbf{let}\ (x',\ y')\ =\ z\ \mathbf{in} \\
&\qquad\qquad \mathbf{let}\ [x]\ =\ x'\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{let}\ [y]\ =\ y'\ \mathbf{in}\ u\ [(x,\ ())]
\end{aligned}
$$

3.1.1 *Metatheory*

Finally, as hinted at on page 23 of Chapter 2, the admissibility of substitution is a key result that holds for this language [Orchard et al., 2019], which is leveraged in soundness of the synthesis calculi.

Lemma 3.1.1 (Admissibility of substitution)**.** Let $\Delta \vdash t' : A$, then:

- (Linear) If $\Gamma, x : A, \Gamma' \vdash t : B$ then $\Gamma + \Delta + \Gamma' \vdash [t'/x]t : B$
- (Graded) If $\Gamma, x :_r A, \Gamma' \vdash t : B$ then $\Gamma + (r \cdot \Delta) + \Gamma' \vdash [t'/x]t : B$

3.2 THE RESOURCE MANAGEMENT PROBLEM

In Chapter 1 we considered a synthesis rule for pairs and highlighted how graded types could be used to control the number of times assumptions are used in the synthesised term.

Chapter 1 considered (Cartesian) product types $\times$, but in our target language we use the multiplicative product of linear types:

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

Each sub-term is typed by a different context Γ_1 and Γ_2 which are then combined via context addition, which equates to *disjoint union* when considering a solely linear setting: the pair cannot be formed if linear

variables are shared between Γ_1 and Γ_2. This prevents the structural behaviour of *contraction* (where a variable appears in multiple sub-terms). Naïvely inverting this typing rule into a synthesis rule yields:

$$\frac{\Gamma_1 \vdash A \Rightarrow t_1 \qquad \Gamma_2 \vdash B \Rightarrow t_2}{\Gamma_1, \Gamma_2 \vdash A \otimes B \Rightarrow (t_1,\ t_2)} \otimes_{\textsc{Intro}}$$

As a declarative specification, the $\otimes_{\textsc{Intro}}$ synthesis rule is sufficient. However, this rule embeds a considerable amount of non-determinism when considered from an algorithmic perspective. Reading 'clockwise' starting from the bottom-left, given some context Γ and a goal $A \otimes B$, we have to split the context into disjoint subparts Γ_1 and Γ_2 such that $\Gamma = \Gamma_1, \Gamma_2$ in order to pass Γ_1 and Γ_2 to the sub-goals for A and B. For a context of size n there are 2^n possible such partitions! This quickly becomes intractable. Instead, Hodas and Miller [1994] developed a technique for linear logic programming, refined by Cervesato et al. [2000], where proof search for linear logic has both an *input context* of available resources and an *output context* of the remaining resources, which we write as judgements of the form $\Gamma \vdash A \Rightarrow^- t \mid \Gamma'$ for input context Γ and output context Γ'. Synthesis for multiplicative products then becomes:

$$\frac{\Gamma_1 \vdash A \Rightarrow^- t_1 \mid \Gamma_2 \qquad \Gamma_2 \vdash B \Rightarrow^- t_2 \mid \Gamma_3}{\Gamma_1 \vdash A \otimes B \Rightarrow^- (t_1, t_2) \mid \Gamma_3} \otimes^-_{\textsc{Intro}}$$

where the remaining resources after synthesising for A the first term t_1 are Γ_2 which are then passed as the resources for synthesising the second term B. There is an ordering implicit here in 'threading through' the contexts between the premises. For example, starting with a context $x : A, y : B$, and given the following rule for synthesising a variable usage:

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^- x \mid \Gamma} \textsc{Var}$$

then the $\otimes^-_{\textsc{Intro}}$ rule may be instantiated as:

$$\frac{x : A, y : B \vdash A \Rightarrow^- x \mid y : B \qquad y : B \vdash B \Rightarrow^- y \mid \emptyset}{x : A, y : B \vdash A \otimes B \Rightarrow^- (x, y) \mid \emptyset} \otimes^-_{\textsc{Intro}}$$

(example)

Thus this approach neatly avoids the problem of having to split the input context, and facilitates efficient proof search for linear types. We extend this input-output context management model to the graded linear λ-calculus to facilitate the synthesis of programs in Granule. We term the above approach *subtractive* resource management (in a style similar to *left-over* type checking for linear type systems [Allais, 2018, Zalakain and Dardha, 2020]).

Graded type systems, as we consider them here, have typing contexts in which free-variables are assigned a type, and a grade. In a graded setting, the subtractive approach is problematic as there is not necessarily a notion of actual subtraction for grades. Consider a version of the above example for subtractively synthesising a pair, but now for a context with some grades r and s on the input variables. Using a variable to synthesise a sub-term now does not result in that variable being left out of the output context. Instead a new grade must be assigned in the output context that relates to the first by means of an additional constraint describing that some usage took place:

$$\frac{\begin{array}{c}\exists r'.r'+1=r \qquad \exists s'.s'+1=s \\ x :_r A, y :_s B \vdash A \Rightarrow^- x \mid x :_{r'} A, y :_s B \\ x :_{r'} A, y :_s B \vdash B \Rightarrow^- y \mid x :_{r'} A, y :_{s'} B\end{array}}{x :_r A, y :_s B \vdash A \otimes B \Rightarrow^- (x,y) \mid x :_{r'} A, y :_{s'} B}\ \otimes^-_{\text{INTRO}} \qquad \text{(example)}$$

In the first synthesis premise, x has grade r in the input context, x is synthesised for the goal, and thus the output context has some grade r' where $r'+1=r$, denoting that some usage of x occurred (which is represented by the 1 element of the semiring in graded systems).

For the natural numbers semiring, with $r=1$ and $s=1$ then the constraints above are satisfied with $r'=0$ and $s'=0$. In a general setting, this subtractive approach to synthesis for graded types requires solving many such existential equations over semirings, which also introduces a new source of non-determinism if there is more than one solution. These constraints can be discharged via an off-the-shelf SMT solver, such as Z3 [de Moura and Bjørner, 2008]. Such calls to an external solver are costly, however, and thus efficiency of resource management is a key concern.

We propose a dual approach to the subtractive: the *additive* resource management scheme. In the additive approach, output contexts describe what was *used* not what was is *left*. In the case of synthesising a term with multiple sub-terms (like pairs), the output context from each premise is then added together using the semiring addition operation applied pointwise on contexts to produce the final output in the conclusion. For pairs this looks like:

$$\frac{\Gamma \vdash A \Rightarrow^+ t_1 \mid \Delta_1 \qquad \Gamma \vdash B \Rightarrow^+ t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^+ (t_1, t_2) \mid \Delta_1 + \Delta_2}\ \otimes^+_{\text{INTRO}}$$

The entirety of Γ is used to synthesise both premises. For example, for a goal of $A \otimes A$:

$$\frac{\begin{array}{c}x :_r A, y :_s B \vdash A \Rightarrow^+ x \mid x :_1 A, y :_0 B \\ x :_r A, y :_s B \vdash A \Rightarrow^+ x \mid x :_1 A, y :_0 B\end{array}}{x :_r A, y :_s B \vdash A \otimes A \Rightarrow^+ (x,x) \mid x :_{1+1} A, y :_0 B}\ \otimes^+_{\text{INTRO}}$$

(example)

Later checks in synthesis then determine whether the output context describes usage that is within the grades given by Γ, i.e., that the synthesised terms are *well-resourced*.

Both the subtractive and additive approaches avoid having to split the incoming context Γ into two prior to synthesising sub-terms.

We adapt the input-output context management model of linear logic synthesis to graded types, pruning the search space via the quantitative constraints of grades. It is not immediately apparent which approach has better performance, thus we implement synthesis calculi based on both the additive and subtractive approaches, evaluating their performance on a set of benchmarking synthesis problems.

We now present two synthesis calculi based on the subtractive and additive resource management schemes, respectively. The structure of the synthesis calculi mirrors a cut-free sequent calculus, with *left* and *right* rules for each type constructor. Right rules synthesise an introduction form for the goal type. Left rules eliminate (deconstruct) assumptions so that they may be used inductively to synthesise sub-terms. Each type in the core language has right and left rules corresponding to its constructors and destructors respectively. The rules as we present them in this section are highly non-deterministic, and are not enough by themselves to use as the basis of an implementation - an issue which will be addressed in the subsequent section on *focusing*.

3.3 A SUBTRACTIVE SYNTHESIS CALCULUS

Our subtractive approach follows the philosophy of earlier work on linear logic proof search [Hodas and Miller, 1994, Cervesato et al., 2000], structuring synthesis rules around an input context of the available resources and an output context of the remaining resources that can be used to synthesise subsequent sub-terms. Synthesis rules are read bottom-up, with judgments $\Gamma \vdash A \Rightarrow^{-} t \mid \Delta$ meaning from the *goal type* A we can synthesise a term t using assumptions in Γ, with output context Δ. We describe the rules in turn to aid understanding. Figure 3.2 collects the rules for reference.

3.3.1 *Variables*

Variable terms can be synthesised from assumptions in the context of linear and graded assumptions Γ by rules:

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^{-} x \mid \Gamma}\ \textsc{LinVar}^{-} \qquad \frac{\exists s.\, r \sqsupseteq s + 1}{\Gamma, x :_r A \vdash A \Rightarrow^{-} x \mid \Gamma, x :_s A}\ \textsc{GrVar}^{-}$$

On the left, a variable x may be synthesised for the goal A if a linear assumption $x : A$ is present in the input context. The input context

without x is then returned as the output context, since x has been used. On the right, we can synthesise a variable x for A if we have a graded assumption of x matching the type. However, the grading r must permit x to be used once here. Therefore, the premise states that there exists some grade s such that grade r approximates $s + 1$. The grade s represents the use of x in further synthesis judgements, and thus $x :_s A$ is in the output context. For the natural numbers semiring, this constraint is satisfied by $s = r - 1$ whenever $r \neq 0$, e.g., if $r = 3$ then $s = 2$. For intervals, the role of approximation is more apparent: if $r = 0...3$ then this rule is satisfied by $s = 0...2$ where $s + 1 = 0...2 + 1...1 = 1...3 \sqsubseteq 0...3$. In the natural numbers semiring, this existential grade variable could be instantiated by simply subtracting 1 from the assumption's existing grade r. However, as not all semirings have an additive inverse, this is instead handled via a constraint on the new grade s, requiring that $r \sqsupseteq s + 1$. In the implementation, the constraint is discharged via an SMT solver, where an unsatisfiable result terminates this branch of synthesis.

3.3.2 *Functions*

In typing, λ-abstraction binds linear variables to introduce linear functions. Synthesis from a linear function type therefore mirrors typing:

$$\frac{\Gamma, x : A \vdash B \Rightarrow^- t \mid \Delta \quad x \notin |\Delta|}{\Gamma \vdash A \multimap B \Rightarrow^- \lambda x.t \mid \Delta} \multimap^-_R$$

Thus, $\lambda x.t$ can be synthesised given that t can be synthesised from B in the context of Γ extended with a fresh linear assumption $x : A$. To ensure that x is used linearly by t we must therefore check that it is not present in Δ.

The elimination rule for linear function types then synthesises applications (as in Hodas and Miller [1994]):

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^- t_1 \mid \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1 \vdash A \Rightarrow^- t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^- [(x_1\, t_2)/x_2]t_1 \mid \Delta_2} \multimap^-_L$$

The rule synthesises a term for type C in a context that contains an assumption $x_1 : A \multimap B$. The first premise synthesises a term t_1 for C under the context extended with a fresh linear assumption $x_2 : B$, i.e., assuming the result of x_1. This produces an output context Δ_1 that must not contain x_2, i.e., x_2 is used by t_1. The remaining assumptions Δ_1 provide the input context to synthesise t_2 of type A: the argument to the function x_1. This structure corresponds to a left-elimination rule of function types in sequent calculus. In the conclusion, the application $x_1\, t_2$ is substituted for x_2 inside t_1, and Δ_2 is the output context.

Remark 1. One might remark upon the formulation of this rule: a more straightforward alternative would seem to be:

$$\frac{\Gamma, x_3 : B \vdash C \Rightarrow^- t \mid \Delta}{\Gamma, x_1 : A \multimap B, x_2 : A \vdash C \Rightarrow^- x_1\ t \mid \Delta}\ \multimap^-_{\mathrm{L}}$$

The reason for this is that such a rule would not adhere to the format of a *cut-free* sequent calculus [Pfenning, 2002] which allows us to view proof search as a purely "bottom-up" construction of a derivation, eliminating unnecessary non-determinism.

3.3.3 *Dereliction*

Note that the above rule synthesises the application of a function given by a linear assumption. What if we have a graded assumption of function type? Rather than duplicating every left rule for both linear and graded assumptions, we mirror the dereliction typing rule (converting a linear assumption to graded) as:

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^- t \mid \Delta, x :_{s'} A \qquad y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s + 1}{\Gamma, x :_r A \vdash B \Rightarrow^- [x/y]t \mid \Delta, x :_{s'} A}\ \textsc{der}^-$$

Dereliction captures the ability to reuse a graded assumption being considered in a left rule. A fresh linear assumption y is generated that represents a use of the graded assumption x when used in a left rule, and must be used linearly in the subsequent synthesis of t. As with the $\multimap^-_{\mathrm{L}}$ rule, the use of y is substituted for x in the body of the synthesised term t.

The output context of the premise then contains x graded by s', which reflects how x was used in the synthesis of t, i.e. if x was not used then $s' = s$. The premise $\exists s.\, r \sqsupseteq s + 1$ constrains the number of derelictions that can be used so that the grade r is not exceeded.

One may observe that the $\textsc{der}^-$ rule makes the presence of the $\textsc{GrVar}^-$ rule admissible. Synthesising the usage of a graded variable can instead be achieved through the use of dereliction on the graded assumption, followed by the $\textsc{LinVar}^-$. For example, the following derivation for synthesising a value of type A from a context with the graded assumption $x :_1 A$ using $\textsc{der}^-$:

$$\frac{\dfrac{}{x :_s A, y : A \vdash A \Rightarrow^- y \mid x :_s A}\ \textsc{LinVar}^- \qquad y \notin x :_s A \qquad \exists s.1 \sqsupseteq s + 1}{x :_1 A \vdash A \Rightarrow^- [x/y]y \mid x :_s A}\ \textsc{der}^-$$

is equivalent to using the $\textsc{GrVar}^-$ rule (after applying the substitution in the $\textsc{der}^-$ rule's conclusion):

$$\frac{\exists s.1 \sqsupseteq s + 1}{x :_1 A \vdash A \Rightarrow^- x \mid x :_s A}\ \textsc{GrVar}^-$$

Nevertheless, we find the inclusion of GrVar$^-$ useful as an explanatory tool and optimisation in the implementation of the calculus.

3.3.4 *Graded modalities*

For a graded modal goal type $\Box_r A$, we synthesise a promotion $[t]$ if we can synthesise the 'unpromoted' t from A:

$$\frac{\Gamma \vdash A \Rightarrow^- t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^- [t] \mid \Gamma - r \cdot (\Gamma - \Delta)}\ \Box_R^-$$

A non-graded value t may be promoted to a graded value using the box syntactic construct. Recall that typing of a promotion $[t]$ scales all the graded assumptions used to type t by r. Therefore, to compute the output context we must "subtract" r-times the use of the variables in t. However, in the subtractive model Δ tells us what is left, rather than what is used. Thus we first compute the *context subtraction* of Γ and Δ yielding the variable usage information about t:

Definition 3.3.1 (Context subtraction)**.** For all Γ_1, Γ_2 where $\Gamma_2 \subseteq \Gamma_1$, $\Gamma_1 - \Gamma_2 =$

$$\begin{cases} \Gamma_1 & \Gamma_2 = \emptyset \\ (\Gamma_1', \Gamma_1'') - \Gamma_2' & \Gamma_2 = \Gamma_2', x : A \quad \wedge\, \Gamma_1 = \Gamma_1', x : A, \Gamma_1'' \\ ((\Gamma_1', \Gamma_1'') - \Gamma_2'), x :_q A & \Gamma_2 = \Gamma_2', x :_s A \quad \wedge\, \Gamma_1 = \Gamma_1', x :_r A, \Gamma_1'' \\ & \wedge\, \exists q.\, r \sqsupseteq q + s \ \wedge \forall q'.r \sqsupseteq q' + s \implies q \sqsupseteq q' \end{cases}$$

Note that this is an algorithmic definition of context subtraction. As in graded variable synthesis, context subtraction existentially quantifies a variable q to express the relationship between grades on the right being "subtracted" from those on the left. The last conjunct states q is the greatest element (wrt. to the pre-order) satisfying this constraint, i.e., for all other $q' \in \mathcal{R}$ satisfying the subtraction constraint then $q \sqsupseteq q'$ e.g., if $r = 2...3$ and $s = 0...1$ then $q = 2...2$ instead of, say, $0...1$. This *maximality* condition is important for soundness (i.e. that synthesised programs are well-typed); we discuss soundness further in Section 3.3.7.

Thus for $\Box_R^-$, $\Gamma - \Delta$ is multiplied by the goal type grade r to obtain how these variables are used in t after promotion. This is then subtracted from the original input context Γ giving an output context containing the left-over variables and grades. Context multiplication requires that $\Gamma - \Delta$ contains only graded variables, preventing the incorrect use of linear variables from Γ in t. For example, the derivation:

$$\frac{\dfrac{\exists s.0...2 \sqsupseteq s + 1}{x :_{0...2} A \vdash A \Rightarrow^- x \mid x :_s A}\ \textsc{GrVar}^-}{x :_{0...2} A \vdash \Box_{0...1} \Rightarrow^- [x] \mid x :_{0...2} A - 0...1 \cdot (x :_{0...2} A - x :_s A)}\ \Box_R^-$$

would be valid. Here, the subtraction $x :_{0...2} A - x :_s A$ yields a grade variable q_1 with the constraint $\exists q_1.0...2 \sqsupseteq q_1 + 0...1$ and q_1' with the maximality constraint $\forall q_1'.0...2 \sqsupseteq q_1' + 0...1 \Rightarrow q_1 \sqsupseteq q_1'$, satisfied by $q_1 = 0...1$, and $q_1' = q_1$, as s is satisfied by $0...1$. Finally, $0...1 \cdot x :_{q_1} A$ is subtracted from $x :_{0...2} A$ to obtain the left over usage: $0...1 \cdot x :_{q_1} A$ is $x :_{q_1} A$, thus the subtraction yields a grade q_2 with constraint $\exists q_2.0...2 \sqsupseteq q_2 + q_1$ and q_2' with maximality constraint $\forall q_2'.0...2 \sqsupseteq q_2' + q_1 \Rightarrow q_2 \sqsupseteq q_2'$, which is satisfied by $q_2 = 0...1$; the final output grade for x in the $\Box_R^-$ rule.

Synthesis of graded modality elimination is handled by the $\Box_L^-$ rule:

$$\frac{\Gamma, x_2 :_r A \vdash B \Rightarrow^- t \mid \Delta, x_2 :_s A \qquad 0 \sqsubseteq s}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^- \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid \Delta}\,\Box_L^-$$

Given an input context comprising Γ and a linear assumption x_1 of graded modal type, we can synthesise an unboxing of x_1 if we can synthesise a term t under Γ extended with a graded assumption $x_2 :_r A$. This returns an output context that must contain x_2 graded by s with the constraint that s must approximate 0, as in the following synthesis derivation where x_1 is an assumption with a graded modality of grade 0, i.e. it *must* be discarded after unboxing:

$$\frac{\dfrac{}{y : A, x_2 :_0 A \vdash A \Rightarrow^- y \mid x_2 :_0 A}\,\textsc{LinVar}^- \qquad 0 \sqsubseteq 0}{y : A, x_1 : \Box_0 A \vdash A \Rightarrow^- \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,y \mid \emptyset}\,\Box_L^-$$

This enforces that x_2 has been used as is permitted by the grade r.

3.3.5 *Products*

The right rule for products $\otimes_R^-$ behaves similarly to the $\multimap_L^-$ rule, again synthesising two sub-terms, and passing the entire input context Γ to the first premise. This is in then used to synthesise the first sub-term of the pair t_1, yielding an output context Δ_1, which is passed to the second premise. After synthesising the second sub-term t_2, the output context for this premise becomes the output context of the rule's conclusion. Although there is an implicit ordering present in this rule, the ability for the implementation to backtrack means the synthesis tool can always try alternative combinations of values for t_1 and t_2.

The left rule equivalent $\otimes_L^-$ binds two assumptions $x_1 : A\ x_2 : B$ in the premise, representing the constituent parts of the pair. As with $\multimap_L^-$, we also ensure that these bound assumptions must not be present in the premise's output context Δ.

$$\frac{\Gamma \vdash A \Rightarrow^- t_1 \mid \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow^- t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^- (t_1, t_2) \mid \Delta_2} \otimes_R^-$$

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^- t_2 \mid \Delta \qquad x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^- \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta} \otimes_L^-$$

3.3.5.1 *Sums*

The right rules for sum types, $\oplus 1_R^-$ and $\oplus 2_R^-$, are straightforward:

$$\frac{\Gamma \vdash A \Rightarrow^- t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^- \mathbf{inl}\,t \mid \Delta} \oplus 1_R^- \qquad \frac{\Gamma \vdash B \Rightarrow^- t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^- \mathbf{inr}\,t \mid \Delta} \oplus 2_R^-$$

The $\oplus_L^-$ rule synthesises the left and right branches of a case statement that may use resources differently:

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow^- t_1 \mid \Delta_1 \quad \Gamma, x_3 : B \vdash C \Rightarrow^- t_2 \mid \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2|}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^- \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcap \Delta_2} \oplus_L^-$$

The output context therefore takes the *greatest lower bound* ($\sqcap$) of Δ_1 and Δ_2, given by definition 3.3.2,

Definition 3.3.2 (Partial greatest-lower bounds of contexts)**.** For all Γ_1, Γ_2, $\Gamma_1 \sqcap \Gamma_2 =$

$$\begin{cases} \emptyset & \Gamma_1 = \emptyset & \wedge\ \Gamma_2 = \emptyset \\ (\emptyset \sqcap \Gamma_2'), x :_{0 \sqcap s} A & \Gamma_1 = \emptyset & \wedge\ \Gamma_2 = \Gamma_2', x :_s A \\ (\Gamma_1' \sqcap (\Gamma_2', \Gamma_2'')), x : A & \Gamma_1 = \Gamma_1', x : A & \wedge\ \Gamma_2 = \Gamma_2', x : A, \Gamma_2'' \\ (\Gamma_1' \sqcap (\Gamma_2', \Gamma_2'')), x :_{r \sqcap s} A & \Gamma_1 = \Gamma_1', x :_r A & \wedge\ \Gamma_2 = \Gamma_2', x :_s A, \Gamma_2'' \end{cases}$$

where $r \sqcap s$ is the greatest-lower bound of grades r and s if it exists, derived from the pre-order $\sqsubseteq$ (given by Definition 2.3.1). If the greatest lower bound of two grades does not exist, then the operation fails, terminating the branch of synthesis.

This operation is the dual to least-upper bound (Definition 3.1.1) operation used in the typing rule for Case, as here we are subtracting usages rather than adding them. As an example of $\sqcap$, consider the semiring of intervals over natural numbers and two judgements that could be used as premises for the ($\oplus_L^-$) rule:

$$\Gamma, y :_{0\ldots5} A', x_2 : A \vdash C \Rightarrow^- t_1 \mid y :_{2\ldots5} A'$$
$$\Gamma, y :_{0\ldots5} A', x_3 : B \vdash C \Rightarrow^- t_2 \mid y :_{3\ldots4} A'$$

where t_1 uses y such that there are 2-5 uses remaining and t_2 uses y such that there are 3-4 uses left. To synthesise **case** x_1 **of inl** $x_2 \to t_1$; **inr** $x_3 \to t_2$ the output context must be pessimistic about what resources are left, thus we take the greatest-lower bound yielding the interval $2 \ldots 4$ here: we know y can be used at least twice and at most 4 times in the rest of the synthesised program.

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^{-} x \mid \Gamma}\ \textsc{LinVar}^{-} \qquad \frac{\exists s.\, r \sqsupseteq s+1}{\Gamma, x :_r A \vdash A \Rightarrow^{-} x \mid \Gamma, x :_s A}\ \textsc{GrVar}^{-}$$

$$\frac{\Gamma, x : A \vdash B \Rightarrow^{-} t \mid \Delta \quad x \notin |\Delta|}{\Gamma \vdash A \multimap B \Rightarrow^{-} \lambda x.t \mid \Delta}\ \multimap^{-}_{R}$$

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1 \vdash A \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{-} [(x_1\, t_2)/x_2]t_1 \mid \Delta_2}\ \multimap^{-}_{L}$$

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^{-} t \mid \Delta, x :_{s'} A \qquad y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s+1}{\Gamma, x :_r A \vdash B \Rightarrow^{-} [x/y]t \mid \Delta, x :_{s'} A}\ \textsc{der}^{-}$$

$$\frac{\Gamma \vdash A \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^{-} [t] \mid \Gamma - r \cdot (\Gamma - \Delta)}\ \Box^{-}_{R}$$

$$\frac{\Gamma, x_2 :_r A \vdash B \Rightarrow^{-} t \mid \Delta, x_2 :_s A \qquad 0 \sqsubseteq s}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^{-} \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid \Delta}\ \Box^{-}_{L}$$

$$\frac{\Gamma \vdash A \Rightarrow^{-} t_1 \mid \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{-} (t_1, t_2) \mid \Delta_2}\ \otimes^{-}_{R}$$

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^{-} t_2 \mid \Delta \qquad x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^{-} \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta}\ \otimes^{-}_{L}$$

$$\frac{\Gamma \vdash A \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{-} \mathbf{inl}\,t \mid \Delta}\ \oplus 1^{-}_{R} \qquad \frac{\Gamma \vdash B \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{-} \mathbf{inr}\,t \mid \Delta}\ \oplus 2^{-}_{R}$$

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \quad \Gamma, x_3 : B \vdash C \Rightarrow^{-} t_2 \mid \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2|}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^{-} \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcap \Delta_2}\ \oplus^{-}_{L}$$

$$\frac{}{\Gamma \vdash \mathsf{Unit} \Rightarrow^{-} () \mid \Gamma}\ \mathsf{Unit}^{-}_{R} \qquad \frac{\Gamma \vdash C \Rightarrow^{-} t \mid \Delta}{\Gamma, x : \mathsf{Unit} \vdash C \Rightarrow^{-} \mathbf{let}\,() = x\,\mathbf{in}\,t \mid \Delta}\ \mathsf{Unit}^{-}_{L}$$

Figure 3.2: Collected rules of the subtractive synthesis calculus

3.3.6 *Unit*

The right and left rules for units are then as follows:

$$\frac{}{\Gamma \vdash \mathsf{Unit} \Rightarrow^{-} () \mid \Gamma}\ \mathsf{Unit}^{-}_{R} \qquad \frac{\Gamma \vdash C \Rightarrow^{-} t \mid \Delta}{\Gamma, x : \mathsf{Unit} \vdash C \Rightarrow^{-} \mathbf{let}\,() = x\,\mathbf{in}\,t \mid \Delta}\ \mathsf{Unit}^{-}_{L}$$

As no resources are used to synthesise a $()$ unit value, no subtraction need take place in the output contexts.

3.3.7 *Soundness of Subtractive Synthesis*

This completes subtractive synthesis calculus. We conclude with a key result, that synthesised terms are well-typed at the type from which they were synthesised:

Lemma 3.3.1 (Subtractive synthesis soundness)**.** For all Γ and A then:

$$\Gamma \vdash A \Rightarrow^{-} t \mid \Delta \quad \Longrightarrow \quad \Gamma - \Delta \vdash t : A$$

i.e. t has type A under context $\Gamma - \Delta$, that contains just those linear and graded variables with grades reflecting their use in t.

The proof of soundness can be found in Section B.1.1 of Appendix B.

The proof is by induction. As a simple example, consider the case of the $\multimap^{-}_{R}$ rule. By induction on the premise of the rule we have:

$$(\Gamma, x : A) - \Delta \vdash t : B$$

Then, since $x \notin |\Delta|$ then by the definition of context subtraction we have that $(\Gamma, x : A) - \Delta = (\Gamma - \Delta), x : A$. From this, we can construct the following derivation, matching the conclusion:

$$\frac{(\Gamma - \Delta), x : A \vdash t : B}{\Gamma - \Delta \vdash \lambda x.t : A \multimap B}\ \textsc{Abs}$$

3.3.8 *An Example of a Subtractive Synthesis Derivation*

We conclude this section with an example of a synthesis derivation tree in Figure 3.3 for the type $(\Box_{\mathsf{Lo}} A \otimes \Box_{\mathsf{Hi}} A) \multimap (\Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B) \multimap B$, (i.e. the *noLeak* program from Example 3.1.2). Due to space constraints, we use aliases for contexts and terms cases. The values corresponding to these aliases can be found in Table 3.1 for input contexts, terms, and output contexts, respectively. The number of each context/term also corresponds to the position indicated on the left hand side of the derivation tree.

We highlight in yellow the points at which a constraint is generated and solved, i.e. the choice points of the derivation where backtracking may occur. Note that some of these constraints are generated and solved in Δ_8.

The first occurrence of constraint generation takes place in the GrVar^{-} rule when synthesising e_{10} (i.e. a usage of x). Here, a constraint of the form $\exists s.\mathsf{Lo} \sqsupseteq s + \mathsf{Lo}$ is generated and solved, since $\mathsf{Lo} \sqsubseteq \mathsf{Lo} + \mathsf{Lo}$, for example. This grade variable s then becomes the grade of x in the output context Δ_{10}.

Next, we have the rather complex grade calculation that occurs in Δ_8, the output context of the $\Box^{-}_{R}$ rule. This rule requires us to subtract $\mathsf{Lo} \cdot ((x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A) - (x :_{s} A, y :_{\mathsf{Hi}} A))$ from the input context

$(x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A)$ (see Section 3.3.4). We do so using Definition 3.3.1 for context subtraction, which generates two constraints for each assumption involved in the subtraction. For x, substituting s with Lo (the value for s we chose before) we have the constraint:

$$\mathsf{Lo} \sqsupseteq \mathsf{Lo} + \mathsf{Lo} \wedge \forall q'.\mathsf{Lo} \sqsupseteq q' + \mathsf{Lo} \implies \mathsf{Lo} \sqsupseteq q'$$

and for y we have:

$$\exists r.\mathsf{Hi} \sqsupseteq r + \mathsf{Hi} \wedge \forall r'.\mathsf{Hi} \sqsupseteq r' + \mathsf{Hi} \implies r \sqsupseteq r'$$

which is satisfied by $r = \mathsf{Hi}$. Thus we have

$$(x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A) - (x :_{\mathsf{Lo}\cdot\mathsf{Lo}} A, y :_{\mathsf{Lo}\cdot\mathsf{Hi}} A)$$

yielding the constraint:

$$\exists q''.\mathsf{Lo} \sqsupseteq q'' + (\mathsf{Lo}\cdot\mathsf{Lo}) \wedge \forall q'''.\mathsf{Lo} \sqsupseteq q'' + (\mathsf{Lo}\cdot\mathsf{Lo}) \implies q'' \sqsupseteq q'''$$

for x which is satisfied by $q'' = \mathsf{Lo}$, and for y:

$$\exists r''.\mathsf{Hi} \sqsupseteq r'' + (\mathsf{Lo}\cdot\mathsf{Hi}) \wedge \forall r'''.\mathsf{Hi} \sqsupseteq r'' + (\mathsf{Lo}\cdot\mathsf{Hi}) \implies r'' \sqsupseteq r'''$$

which is satisfied by $r'' = \mathsf{Hi}$ (since $\mathsf{Hi} \sqcap (\mathsf{Lo}\cdot\mathsf{Hi}) = \mathsf{Hi}$).

Following this, the only remaining constraints are generated in the two occurrences of the $\Box_L^-$ rules, dealing first with y (which now has grade r'', i.e. Hi) and then x (which has grade q'', i.e., Lo). Since both $0 \sqsubseteq \mathsf{Lo}$ and $0 \sqsubseteq \mathsf{Hi}$, these constraints both succeed.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_7 \vdash B \Rightarrow^- e_7 \mid \Delta_7}\,\textsc{LinVar}^-_{7} \quad u' \notin |\Delta_7| \quad (3.1)}{\Gamma_6 \vdash B \Rightarrow^- e_6 \mid \Delta_6 \quad 0 \sqsubseteq r''}\,\multimap^-_{L,6}}{\Gamma_5 \vdash B \Rightarrow^- e_5 \mid \Delta_5 \quad 0 \sqsubseteq q''}\,\Box^-_{L,5}}{\Gamma_4 \vdash B \Rightarrow^- e_4 \mid \Delta_4 \quad x' \notin |\Delta_4| \quad y' \notin |\Delta_4|}\,\Box^-_{L,4}}{\Gamma_3 \vdash B \Rightarrow^- e_3 \mid \Delta_3 \quad u \notin |\Delta_3|}\,\otimes^-_{L,3}}{\Gamma_2 \vdash (\Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B) \multimap B \Rightarrow^- e_2 \mid \Delta_2 \quad z \notin |\Delta_2|}\,\multimap^-_{R,2}}{\Gamma_1 \vdash (\Box_{\mathsf{Lo}}A \otimes \Box_{\mathsf{Hi}}A) \multimap (\Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B) \multimap B \Rightarrow^- e_1 \mid \Delta_1}\,\multimap^-_{R,1}
$$

$$
\dfrac{\dfrac{\dfrac{\exists s.\mathsf{Lo} \sqsupseteq s + \mathsf{Lo}}{\Gamma_{10} \vdash A \Rightarrow^- e_{10} \mid \Delta_{10}}\,\textsc{GrVar}^-_{10} \qquad \dfrac{}{\Gamma_{11} \vdash \mathsf{Unit} \Rightarrow^- e_{11} \mid \Delta_{11}}\,\mathsf{Unit}^-_{R,11}}{\Gamma_9 \vdash A \otimes \mathsf{Unit} \Rightarrow^- e_9 \mid \Delta_9}\,\otimes^-_{R,9}}{\Gamma_8 \vdash \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \Rightarrow^- e_8 \mid \Delta_8}\,\Box^-_{R,8}
$$

(3.1)

No.	Γ (Input Context)	e (Synthesised Term)	Δ (Output Context)
1	$\emptyset$	$\lambda z.e_2$	$\emptyset$
2	$z : \Box_{\mathsf{Lo}}A \otimes \Box_{\mathsf{Hi}}A$	$\lambda u.e_3$	$\emptyset$
3	$z : \Box_{\mathsf{Lo}}A \otimes \Box_{\mathsf{Hi}}A$, $u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$	$\mathbf{let}\ (x', y') = z\ \mathbf{in}\ e_4$	$\emptyset$
4	$u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$, $x' : \Box_{\mathsf{Lo}}A$, $y' : \Box_{\mathsf{Hi}}A$	$\mathbf{let}\ [x] = x'\ \mathbf{in}\ e_5$	$\emptyset$
5	$u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$, $y' : \Box_{\mathsf{Hi}}A$, $x :_{\mathsf{Lo}} A$	$\mathbf{let}\ [y] = y'\ \mathbf{in}\ e_6$	$x :_{q''} A$
6	$u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$, $x :_{\mathsf{Lo}} A$, $y :_{\mathsf{Hi}} A$	$[u\ e_8/u']e_7$	$x :_{q''} A, y :_{r''}$
7	$x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A, u' : B$	u'	$x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A$
8	$x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A$	$[e_9]$	$x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}}$ $-\mathsf{Lo} \cdot (x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A - x :_s A, y :_{\mathsf{Hi}} A)$
9	$x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A$	(e_{10}, e_{11})	$x :_s A, y :_{\mathsf{Hi}} A$
10	$x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A$	x	$x :_s A, y :_{\mathsf{Hi}} A$
11	$x :_s A, y :_{\mathsf{Hi}} A$	$()$	$x :_s A, y :_{\mathsf{Hi}} A$

Table 3.1: Values for Γ, e, and Δ throughout the synthesis derivation tree

Figure 3.3: Subtractive synthesis derivation for the Example 2.3.1

3.4 AN ADDITIVE SYNTHESIS CALCULUS

We now present the dual to subtractive resource management — the *additive* approach. Additive synthesis also uses the input-output context approach, but where output contexts describe exactly which assumptions were used to synthesise a term, rather than which assumptions are still available. As with subtractive, additive synthesis rules are best read bottom-up, with $\Gamma \vdash A \Rightarrow^{+} t \mid \Delta$ meaning that from the type A we synthesise a term t using exactly the assumptions Δ that originate from the input context Γ. The rules are collected in Figure 3.4.

3.4.1 *Variables*

We unpack the rules, starting with variables:

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^{+} x \mid x : A}\ \textsc{LinVar}^{+} \qquad \frac{}{\Gamma, x :_r A \vdash A \Rightarrow^{+} x \mid x :_1 A}\ \textsc{GrVar}^{+}$$

For a linear assumption, the output context contains just the variable that was synthesised. For a graded assumption $x :_r A$, the output context contains the assumption graded by 1.

3.4.2 *Graded modalities*

The subtractive approach handled the $\textsc{GrVar}^{-}$ by a constraint $\exists s.\, r \sqsupseteq s + 1$. Here however, the point at which we check that a graded assumption has been used according to the grade takes place in the $\Box_L^{+}$ rule, where graded assumptions are bound:

$$\frac{\Gamma, x_2 :_r A \vdash B \Rightarrow^{+} t \mid \Delta \qquad \textit{if } x_2 :_s A \in \Delta \textit{ then } s \sqsubseteq r \textit{ else } 0 \sqsubseteq r}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^{+} \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid (\Delta \backslash x_2), x_1 : \Box_r A}\ \Box_L^{+}$$

Here, t is synthesised under a fresh graded assumption $x_2 :_r A$. This produces an output context containing x_2 with some grade s that describes how x_2 is used in t (if it was used at all). An additional premise requires that the original grade r approximates either s if x_2 appears in Δ or 0 if it does not, ensuring that x_2 has been used correctly. For the $\mathbb{N}$-semiring with equality as the ordering, this would ensure that a variable has been used exactly the number of times specified by the grade.

The synthesis of a promotion is considerably simpler in the additive approach. In subtractive resource management it was necessary to calculate how resources were used in the synthesis of t before then applying the scalar context multiplication by the grade r and subtracting this from the original input Γ. In additive resource management,

however, we can simply apply the multiplication directly to the output context Δ to obtain how our assumptions are used in $[t]$:

$$\frac{\Gamma \vdash A \Rightarrow^{+} t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^{+} [t] \mid r \cdot \Delta}\ \Box_R^{+}$$

3.4.3 *Functions*

Synthesis rules for $\multimap$ have a similar shape to the subtractive calculus:

$$\frac{\Gamma, x : A \vdash B \Rightarrow^{+} t \mid \Delta, x : A}{\Gamma \vdash A \multimap B \Rightarrow^{+} \lambda x.t \mid \Delta}\ \multimap_R^{+}$$

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1 \mid \Delta_1, x_2 : B \qquad \Gamma \vdash A \Rightarrow^{+} t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{+} [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap_L^{+}$$

Synthesising an abstraction ($\multimap_R^{+}$) requires that $x : A$ is in the output context of the premise, ensuring that linearity is preserved. Likewise for application ($\multimap_L^{+}$), the output context of the first premise must contain the linearly bound $x_2 : B$ and the final output context must contain the assumption being used in the application $x_1 : A \multimap B$. This output context computes the *context addition* (Def. 2.3.3 on page17) of both output contexts of the premises $\Delta_1 + \Delta_2$. If Δ_1 describes how assumptions were used in t_1 and Δ_2 respectively for t_2, then the addition of these two contexts describes the usage of assumptions for the entire subprogram. Recall, context addition ensures that a linear assumption may not appear in both Δ_1 and Δ_2, preventing us from synthesising terms that violate linearity.

3.4.4 *Dereliction*

As in the subtractive calculus, we avoid duplicating left rules to match graded assumptions by giving a synthesising version of dereliction:

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^{+} t \mid \Delta, y : A}{\Gamma, x :_s A \vdash B \Rightarrow^{+} [x/y]t \mid \Delta + x :_1 A}\ \textsc{der}^{+}$$

The fresh linear assumption $y : A$ must appear in the output context of the premise, ensuring it is used. The final context therefore adds to Δ an assumption of x graded by 1, accounting for this use of x (which was temporarily renamed to y). As with the subtractive case, $\textsc{der}^{+}$ makes $\textsc{GrVar}^{+}$ admissible.

3.4.5 *Products*

The right rule for products $\otimes_R^{+}$ follows the same structure as its subtractive equivalent, however, here Γ is passed to both premises. The

conclusion's output context is then formed by taking the context addition of the Δ_1 and Δ_2. The left rule, $\otimes_L^+$ follows fairly straightforwardly from the resource scheme.

$$\frac{\Gamma \vdash A \Rightarrow^+ t_1 \mid \Delta_1 \qquad \Gamma \vdash B \Rightarrow^+ t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^+ (t_1, t_2) \mid \Delta_1 + \Delta_2} \otimes_R^+$$

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^+ t_2 \mid \Delta, x_1 : A, x_2 : B}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^+ \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta, x_3 : A \otimes B} \otimes_L^+$$

3.4.6 *Sums*

In contrast to the subtractive rule, the rule $\oplus_L^+$ takes the least-upper bound of the premise's output contexts (see definition 3.1.1). Otherwise, the right and left rules for synthesising programs from sum types are straightforward.

$$\frac{\Gamma \vdash A \Rightarrow^+ t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inl}\,t \mid \Delta} \oplus 1_R^+ \qquad \frac{\Gamma \vdash B \Rightarrow^+ t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inr}\,t \mid \Delta} \oplus 2_R^+$$

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow^+ t_1 \mid \Delta_1, x_2 : A \qquad \Gamma, x_3 : B \vdash C \Rightarrow^+ t_2 \mid \Delta_2, x_3 : B}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^- \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcup \Delta_2, x_1 : A \oplus B} \oplus_L^+$$

3.4.7 *Unit*

As in the subtractive approach, the right and left rules for unit types, are as expected.

$$\frac{}{\Gamma \vdash \mathsf{Unit} \Rightarrow^+ () \mid \varnothing}\ \mathsf{Unit}_R^+ \qquad \frac{\Gamma \vdash C \Rightarrow^+ t \mid \Delta}{\Gamma, x : \mathsf{Unit} \vdash C \Rightarrow^+ \mathbf{let}\,() = x\,\mathbf{in}\,t \mid \Delta, x : \mathsf{Unit}}\ \mathsf{Unit}_L^+$$

3.4.8 *Soundness of Additive Synthesis*

Thus concludes the rules for additive synthesis. As with subtractive, we have prove that this calculus is sound.

Lemma 3.4.1 (Additive synthesis soundness)**.** Given a particular preordered semiring $\mathcal{R}$ parametrising the calculi, then, for all contexts Γ and Δ, types A and terms t:

$$\Gamma \vdash A \Rightarrow^+ t \mid \Delta \quad \Longrightarrow \quad \Delta \vdash t : A$$

Thus, the synthesised term t is well-typed at A using only the assumptions Δ whose grades capture variable use in t. i.e., synthesised terms are well typed at the type from which they were synthesised.

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^{+} x \mid x : A}\ \textsc{LinVar}^{+} \qquad \frac{}{\Gamma, x :_r A \vdash A \Rightarrow^{+} x \mid x :_1 A}\ \textsc{GrVar}^{+}$$

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^{+} t \mid \Delta, y : A}{\Gamma, x :_s A \vdash B \Rightarrow^{+} [x/y]t \mid \Delta + x :_1 A}\ \textsc{der}^{+}$$

$$\frac{\Gamma, x : A \vdash B \Rightarrow^{+} t \mid \Delta, x : A}{\Gamma \vdash A \multimap B \Rightarrow^{+} \lambda x.t \mid \Delta}\ \multimap^{+}_{R}$$

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1 \mid \Delta_1, x_2 : B \qquad \Gamma \vdash A \Rightarrow^{+} t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{+} [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap^{+}_{L}$$

$$\frac{\Gamma \vdash A \Rightarrow^{+} t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^{+} [t] \mid r \cdot \Delta}\ \Box^{+}_{R}$$

$$\frac{\Gamma, x_2 :_r A \vdash B \Rightarrow^{+} t \mid \Delta \qquad \textit{if } x_2 :_s A \in \Delta \textit{ then } s \sqsubseteq r \textit{ else } 0 \sqsubseteq r}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^{+} \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid (\Delta \backslash x_2), x_1 : \Box_r A}\ \Box^{+}_{L}$$

$$\frac{\Gamma \vdash A \Rightarrow^{+} t_1 \mid \Delta_1 \qquad \Gamma \vdash B \Rightarrow^{+} t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{+} (t_1, t_2) \mid \Delta_1 + \Delta_2}\ \otimes^{+}_{R}$$

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^{+} t_2 \mid \Delta, x_1 : A, x_2 : B}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^{+} \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta, x_3 : A \otimes B}\ \otimes^{+}_{L}$$

$$\frac{\Gamma \vdash A \Rightarrow^{+} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{+} \mathbf{inl}\,t \mid \Delta}\ \oplus 1^{+}_{R} \qquad \frac{\Gamma \vdash B \Rightarrow^{+} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{+} \mathbf{inr}\,t \mid \Delta}\ \oplus 2^{+}_{R}$$

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow^{+} t_1 \mid \Delta_1, x_2 : A \qquad \Gamma, x_3 : B \vdash C \Rightarrow^{+} t_2 \mid \Delta_2, x_3 : B}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^{-} \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcup \Delta_2, x_1 : A \oplus B}\ \oplus^{+}_{L}$$

$$\frac{}{\Gamma \vdash \mathsf{Unit} \Rightarrow^{+} () \mid \emptyset}\ \mathsf{Unit}^{+}_{R} \qquad \frac{\Gamma \vdash C \Rightarrow^{+} t \mid \Delta}{\Gamma, x : \mathsf{Unit} \vdash C \Rightarrow^{+} \mathbf{let}\,() = x\,\mathbf{in}\,t \mid \Delta, x : \mathsf{Unit}}\ \mathsf{Unit}^{+}_{L}$$

Figure 3.4: Collected rules of the additive synthesis calculus

In the additive calculus, the soundness on its own does not guarantee that a synthesised program t is *well resourced*, i.e., the grades in Δ may not be approximated by the grades in Γ. For example, a valid judgement under semiring $\mathbb{N}_{\equiv}$ is:

$$x :_2 A \vdash A \Rightarrow^{+} x \mid x :_1 A$$

i.e., for goal A, if x has type A in the context then we synthesis x as the result program, regardless of the grades. A synthesis judgement such as this may be part of a larger derivation in which the grades eventually match, i.e., this judgement forms part of a larger derivation which has a further sub-derivation in which x is used again and thus the total usage for x is eventually 2 as prescribed by the input context. However, at the level of an individual judgement we do not

guarantee that the synthesised term is well-resourced. A reasonable *pruning condition* that could be used to assess whether any synthesis judgement is *potentially* well-resourced is $\exists \Delta'.(\Delta + \Delta') \sqsubseteq \Gamma$, i.e., there is some additional usage Δ' (that might come from further on in the synthesis process) that 'fills the gap' in resource use to produce $\Delta + \Delta'$ which is overapproximated by Γ. In this example, $\Delta' = x :_1 A$ would satisfy this constraint, explaining that there is some further possible single usage which will satisfy the incoming grade. We apply this pruning condition at the synthesis of terms which are binders. Therefore, synthesised closed terms are always well-resourced.

The proof of soundness can be found in Section B.1.2 of Appendix B.

As in Section 3.3.7, the proof is fairly straightforward, by induction. Again, we consider the $\multimap^+_{\mathrm{R}}$ rule. By induction we have that:

$$(\Gamma, x : A) - \Delta \vdash t : B$$

Since $x \notin |\Delta|$ then by the definition of context subtraction we have that $(\Gamma, x : A) - \Delta = (\Gamma - \Delta), x : A$. From this, we can construct the following derivation, matching the conclusion:

$$\frac{(\Gamma - \Delta), x : A \vdash t : B}{\Gamma - \Delta \vdash \lambda x.t : A \multimap B}\ \textsc{Abs}$$

Remark 2. An observation that can be made of this soundness property for the additive calculus is that it only guarantees soundness for synthesis of complete programs. Synthesis of a partial program, i.e. where some context of graded assumptions is provided to the initial synthesis rule, cannot guarantee that the usages of those assumptions in the synthesised program will complement their usages in the rest of the program such that the grade constraints will be satisfied (as grade usages are only checked in the $\Box^+_L$ rule). For example, consider the following scenario (where ? represents a program hole and the entry point of synthesis):

$$\begin{aligned} &\mathit{badPair} : \Box_1 A \multimap (A \otimes A) \\ &\mathit{badPair} = \lambda x.\mathbf{let}\ [x] = x'\ \mathbf{in}\ (x,\ ?) \end{aligned}$$

Although this type is in fact uninhabited, a valid synthesis judgment for the partial program at ? would have the form:

$$\frac{}{x :_1 A \vdash A \Rightarrow x \mid x :_1 A}\ \textsc{GrVar}^+$$

yielding the complete program:

$$\begin{aligned} &\mathit{badPair} : \Box_1 A \multimap (A \otimes A) \\ &\mathit{badPair} = \lambda x.\mathbf{let}\ [x] = x'\ \mathbf{in}\ (x,\ x) \end{aligned}$$

which is clearly invalid, as the total usage of x exceeds the grade constraint of 1.

In practice, this can be resolved in the implementation by type-checking the enclosing program containing the synthesised partial program and re-synthesising if grade constraints fail. In this thesis, however, we only consider the synthesis of complete programs.

Note that in the subtractive calculus this issue does not arise, as the grade constraint is checked at the point of variable usage.

3.4.9 *An Example of an Additive Synthesis Derivation*

We now repeat the example synthesis derivation from Section 3.3.8, only here using the additive calculus to synthesise the same program from the same type. The derivation tree can be found in Figure 3.5 for the type $(\Box_{\mathsf{Lo}} A \otimes \Box_{\mathsf{Hi}} A) \multimap (\Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B) \multimap B$. Again, due to space constraints, we use aliases for contexts and terms cases. The values corresponding to these aliases can be found in Table 3.2 for input contexts, terms, and output contexts, respectively. The number of each context/term also corresponds to the position indicated on the left hand side of the derivation tree.

Again, we highlight in yellow the points at which a constraint is generated and solved, i.e. the choice points of the derivation where backtracking may occur. In comparison to the subtractive example, the additive is considerable easier to follow. In fact, there are only two decision points where backtracking can occur, involving relatively straightforward constraints. Both of these occur in the $\Box_L^+$ rules, for x' and y'. In the case of y', the first constraint to be considered, we have the condition *if* $y :'_s A \in \Delta_5$ *then* $s' \sqsubseteq \mathsf{Hi}$ *else* $\mathsf{Hi} \sqsubseteq \mathsf{Hi}$. Here, s' is a potential grade that may appear on y if it were to appear in the output context Δ_5. However y is not present in Δ_5, so we default to the else clause with the trivial constraint that $\mathsf{Hi} \sqsubseteq \mathsf{Hi}$. Likewise, in the case of x', we have the condition *if* $x :_s A \in \Delta_5$ *then* $s \sqsubseteq \mathsf{Lo}$ *else* $\mathsf{Hi} \sqsubseteq \mathsf{Lo}$. In this case, x does appear in Δ_5 with the grade Lo, giving us the constraint $\mathsf{Lo} \sqsubseteq \mathsf{Lo}$, which again is trivially true.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_7 \vdash B \Rightarrow^+ e_7 \mid \Delta_7}\,7\ \textsc{LinVar}^+ \quad (3.2)}{\Gamma_6 \vdash B \Rightarrow^+ e_6 \mid \Delta_6 \quad \textit{if } y :'_s A \in \Delta_5 \textit{ then } s' \sqsubseteq \mathsf{Hi} \textit{ else } \mathsf{Hi} \sqsubseteq \mathsf{Hi}}\,6\ \multimap^+_L}{\Gamma_5 \vdash B \Rightarrow^+ e_5 \mid \Delta_5 \quad \textit{if } x :_s A \in \Delta_5 \textit{ then } s \sqsubseteq \mathsf{Lo} \textit{ else } \mathsf{Hi} \sqsubseteq \mathsf{Lo}}\,5\ \Box^+_L}{\Gamma_4 \vdash B \Rightarrow^+ e_4 \mid \Delta_4}\,4\ \Box^+_L}{\Gamma_3 \vdash B \Rightarrow^+ e_3 \mid \Delta_3}\,3\ \otimes^+_L}{\Gamma_2 \vdash (\Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B) \multimap B \Rightarrow^+ e_2 \mid \Delta_2}\,2\ \multimap^+_R}{\Gamma_1 \vdash (\Box_{\mathsf{Lo}}A \otimes \Box_{\mathsf{Hi}}A) \multimap (\Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B) \multimap B \Rightarrow^+ e_1 \mid \Delta_1}\,1\ \multimap^+_R
$$

$$
\dfrac{\dfrac{\dfrac{}{\Gamma_{10} \vdash A \Rightarrow^+ e_{10} \mid \Delta_{10}}\,10\ \textsc{GrVar}^+ \qquad \dfrac{}{\Gamma_{11} \vdash \mathsf{Unit} \Rightarrow^+ e_{11} \mid \Delta_{11}}\,11\ \mathsf{Unit}^+_R}{\Gamma_9 \vdash A \otimes \mathsf{Unit} \Rightarrow^+ e_9 \mid \Delta_9}\,9\ \otimes^+_R}{\Gamma_8 \vdash \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \Rightarrow^+ e_8 \mid \Delta_8}\,8\ \Box^+_R
$$

(3.2)

No.	Γ (Input Context)	e (Synthesised Term)	Δ (Output Context)
1	$\emptyset$	$\lambda z.e_2$	$\emptyset$
2	$z : \Box_{\mathsf{Lo}}A \otimes \Box_{\mathsf{Hi}}A$	$\lambda u.e_3$	$z : \Box_{\mathsf{Lo}}A \otimes \Box_{\mathsf{Hi}}A$
3	$z : \Box_{\mathsf{Lo}}A \otimes \Box_{\mathsf{Hi}}A$, $u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$	$\mathbf{let}\ (x', y') = z\ \mathbf{in}\ e_4$	$u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$, $z : \Box_{\mathsf{Lo}}A \otimes \Box_{\mathsf{Hi}}A$
4	$u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$, $x' : \Box_{\mathsf{Lo}}A$, $y' : \Box_{\mathsf{Hi}}A$	$\mathbf{let}\ [x] = x'\ \mathbf{in}\ e_5$	$u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$, $y' : \Box_{\mathsf{Hi}}A$, $x' : \Box_{\mathsf{Lo}}A$
5	$u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$, $y' : \Box_{\mathsf{Hi}}A$, $x :_{\mathsf{Lo}} A$	$\mathbf{let}\ [y] = y'\ \mathbf{in}\ e_6$	$x :_{\mathsf{Lo}\cdot\mathsf{Lo}}$, $u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$, $y' : \Box_{\mathsf{Hi}}A$
6	$u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$, $x :_{\mathsf{Lo}} A$, $y :_{\mathsf{Hi}} A$	$[u\ e_8 / u']e_7$	$x :_{\mathsf{Lo}\cdot\mathsf{Lo}}$, $u : \Box_{\mathsf{Lo}}(A \otimes \mathsf{Unit}) \multimap B$
7	$x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A, u' : B$	u'	$u' : B$
8	$x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A$	$[e_9]$	$\mathsf{Lo} \cdot x :_{\mathsf{Lo}}$
9	$x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A$	(e_{10}, e_{11})	$x :_{\mathsf{Lo}}$
10	$x :_{\mathsf{Lo}} A, y :_{\mathsf{Hi}} A$	x	$x :_{\mathsf{Lo}}$
11	$x :_s A, y :_{\mathsf{Hi}} A$	$()$	$\emptyset$

Table 3.2: Values for Γ, e, and Δ throughout the synthesis derivation tree

Figure 3.5: Additive synthesis derivation for the Example 2.3.1

3.4.9.1 *Additive pruning*

As seen above, the additive approach delays checking whether a variable is used according to its linearity/grade until it is bound, i.e. in the $\multimap^+_R$, $\multimap^+_L$, DER$^+$, and $\Box^+_L$ rules. We hypothesise that this can lead additive synthesis to explore many ultimately ill-resourced paths for too long. For example, say we have a partial synthesis derivation for synthesising a pair introduction form of type $A \otimes A$, but our context contains only one linear assumption of type A. Clearly we can use this assumption to synthesise a term for the left part of the pair:

$$\dfrac{\dfrac{}{x : A \vdash A \Rightarrow^+ x \mid x : A}\,\textsc{LinVar}^+ \qquad x : A \vdash A \Rightarrow^+ ? \mid ?}{x : A \vdash A \otimes A \Rightarrow^+ ? \mid ?}\,\otimes^+_R$$

However, after synthesising the left part we no longer have the available usage of x to synthesise a term for right part of the pair. Currently, $\otimes^+_R$ will allow the synthesis of the right part of the pair to take place using x, which will then be discarded when the context addition in the rule's output context fails: conclusion

$$\dfrac{\dfrac{}{x : A \vdash A \Rightarrow^+ x \mid x : A}\,\textsc{LinVar}^+ \qquad x : A \vdash A \Rightarrow^+ x \mid x : A}{x : A \vdash A \otimes A \Rightarrow^+ (x, x) \mid x : A + x : A}\,\otimes^+_R$$

(invalid)

Subsequently, we define a "pruning" variant of any additive rules with multiple sequenced premises. For $\otimes^+_R$ this is:

$$\dfrac{\Gamma \vdash A \Rightarrow^\pm t_1 \mid \Delta_1 \qquad \Gamma - \Delta_1 \vdash B \Rightarrow^\pm t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^\pm (t_1, t_2) \mid \Delta_1 + \Delta_2}\,\otimes^\pm_R$$

Instead of passing Γ to both premises, Γ is the input only for the first premise. This premise outputs context Δ_1 that is subtracted from Γ (highlighted in yellow) to give the input context of the second premise. This provides an opportunity to terminate the current branch of synthesis early if $\Gamma - \Delta_1$ does not contain the necessary resources to attempt the second premise, based on Definition 3.3.1 which may fail. The $\multimap^+_L$ rule is similarly adjusted:

$$\dfrac{\Gamma, x_2 : B \vdash C \Rightarrow^\pm t_1 \mid \Delta_1, x_2 : B \qquad \Gamma - \Delta_1 \vdash A \Rightarrow^\pm t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^\pm [(x_1\ t_2)/x_2] \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B}\,\multimap^\pm_L$$

Lemma 3.4.2 (Additive pruning synthesis soundness)**.** For all Γ and A:

$$\Gamma \vdash A \Rightarrow^\pm t \mid \Delta \quad \Longrightarrow \quad \Delta \vdash t : A$$

The proof of soundness can be found in Section B.1.3 of Appendix B

3.5 FOCUSING

The additive and subtractive calculi presented in Sections 3.3 and 3.4 provide the foundations for the implementations of a synthesis tool for Granule programs. Implementing the rules as they currently stand, however, would yield a highly inefficient tool. In their current form, the rules of both calculi exhibit a high degree of non-determinism with regard to order in which rules can be applied.

This leads to us exploring a large number of redundant search branches: something which can be avoided through the application of a technique from linear logic proof theory called *focusing* [Andreoli, 1992]. Focusing is based on the observation that some of the synthesis rules are invertible, i.e. whenever the conclusion of the rule is derivable, then so are its premises (e.g. the $\multimap^{-}_{R}$ and $\multimap^{+}_{R}$ rules). In other words, the order in which we apply invertible rules doesn't matter. By fixing a particular ordering on the application of these rules, we eliminate much of the non-determinism that arises from trying branches which differ only in the order in which invertible rules are applied. This focused approach restricts synthesis to generating programs in β-normal form.

We take both of our calculi and apply this focusing technique to them, yielding two *focusing* calculi. To do so, we augment our previous synthesis judgement with an additional input context Ω:

$$\Gamma; \Omega \vdash A \Rightarrow t \mid \Delta$$

Unlike Γ and Δ, Ω is an *ordered* context, which behaves like a stack. Assumptions with types that can be broken down further are then bound into Ω.

Using the terminology of Andreoli [1992], we refer to rules that are invertible as *asynchronous* and rules that are not as *synchronous*. The intuition is that of asynchronous communication: asynchronous rules can be applied eagerly, while the non-invertible synchronous rules require us to *focus* on a particular part of the judgement: either on the assumption (if we are in an elimination rule) or on the goal (for an introduction rule). When focusing we apply a chain of synchronous rules until:

- We reach a position where no rules may be applied (at which point the branch terminates).
- We have synthesised a term for our goal.
- We have exposed an asynchronous connective at which point we switch back to applying asynchronous rules.

We divide our synthesis rules into five categories, each with their own judgement form, which refines the focusing judgement above with an

arrow indicating which part of the judgement is currently in focus. An $\Uparrow$ indicates an asynchronous phase, while a $\Downarrow$ indicates a synchronous (focused) phase. The location of the arrow in the judgement indicates whether we are focusing on the left or right:

1. RIGHT ASYNC: $\multimap_R$ rule with the judgement:

$$\Gamma; \Omega \vdash A \Uparrow \Rightarrow t \mid \Delta$$

2. LEFT ASYNC: $\otimes_L$, $\oplus_L$, Unit_L, DER, and $\Box_L$ rules with the judgement:

$$\Gamma; \Omega \Uparrow \vdash A \Rightarrow t \mid \Delta$$

3. RIGHT SYNC: $\otimes_R$, $\oplus 1_R$, $\oplus 2_R$, Unit_R, and $\Box_R$ rules with the judgement:

$$\Gamma; \Omega \vdash A \Downarrow \Rightarrow t \mid \Delta$$

4. LEFT SYNC: $\multimap_L$ rule with the judgement:

$$\Gamma; \Omega \Downarrow \vdash A \Rightarrow t \mid \Delta$$

5. VAR: LINVAR and GRVAR rules (i.e. terminal rules) with the judgement:

$$\Gamma; \Omega \Downarrow \vdash A \Rightarrow t \mid \Delta$$

The complete calculi of focusing synthesis rules are given in Figures 3.6-3.11 for the subtractive calculus, and 3.12-3.16 for the additive, divided into focusing phases. The focusing rules for the additive pruning calculus are identical to the additive calculus, save for the $\otimes_R^+$ and $\multimap_L^+$ rules, which are given in Figure 3.18.

For the most part, the translation from non-focused to focused rules is straightforward. The most notable change occurs in rules in which assumptions are bound. In the cases where a fresh assumption's type falls into the LEFT ASYNC category (i.e. $\otimes$, $\oplus$, etc.), then it is bound in the ordered context Ω instead of Γ. LEFT ASYNC rules operate on assumptions in Ω, rather than Γ. This results in invertible elimination rules being applied as fully as possible before *focusing* on non-invertible rules when Ω is empty.

In addition to the focused forms of the original synthesis calculi, each calculus has a set of rules which determine which part of the synthesis judgement will be focused on: the FOCUS rules. These rules are given by Figures 3.8, and 3.14 for the subtractive and additive calculi, respectively.

We briefly describe each set of rules for the focusing phases of the subtractive synthesis calculus.

$$\frac{\Gamma; \Omega, x : A \vdash C \Uparrow \Rightarrow^- t \mid \Delta \quad x \notin |\Delta|}{\Gamma; \Omega \vdash A \multimap B \Uparrow \Rightarrow^- \lambda x.t \mid \Delta} \multimap^-_R$$

$$\frac{\Gamma; \Omega \Uparrow \vdash C \Rightarrow^- t \mid \Delta \qquad C \text{ not RIGHT ASYNC}}{\Gamma; \Omega \vdash C \Uparrow \Rightarrow^- t \mid \Delta} \Uparrow^-_R$$

Figure 3.6: RIGHT ASYNC rules of the focused subtractive synthesis calculus

The RIGHT ASYNC phase of focusing (Figure 3.6) contains the focused form of the $\multimap^-_R$ rule, which simply binds the variable from the λ term in Ω instead of Γ. The $\Uparrow^-_R$ rule handles the transition from a RIGHT ASYNC phase to a LEFT ASYNC phase. It is applied when the goal type C is no longer right asynchronous (i.e. not a $\multimap$), thus we switch to breaking down the assumptions in Ω by applying a sequence of LEFT ASYNC rules.

$$\frac{\Gamma; \Omega, x_1 : A, x_2 : B \Uparrow \vdash C \Rightarrow^- t_2 \mid \Delta \qquad x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|}{\Gamma; \Omega, x_3 : A \otimes B \Uparrow \vdash C \Rightarrow^- \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta} \otimes^-_L$$

$$\frac{\Gamma; \Omega, x_2 : A \Uparrow \vdash C \Rightarrow^- t_1 \mid \Delta_1 \quad \Gamma; \Omega, x_3 : B \Uparrow \vdash C \Rightarrow^- t_2 \mid \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2|}{\Gamma; \Omega, x_1 : A \oplus B \Uparrow \vdash C \Rightarrow^- \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcap \Delta_2} \oplus^-_L$$

$$\frac{\Gamma; \Omega, x_2 :_r A \Uparrow \vdash B \Rightarrow^- t \mid \Delta, x_2 :_s A \qquad 0 \sqsubseteq s}{\Gamma; \Omega, x_1 : \Box_r A \Uparrow \vdash B \Rightarrow^- \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid \Delta} \Box^-_L$$

$$\frac{\Gamma; \emptyset \vdash C \Rightarrow^- t \mid \Delta}{\Gamma; x : \mathsf{Unit} \vdash C \Rightarrow^- \mathbf{let}\,() = x\,\mathbf{in}\,t \mid \Delta} \mathsf{Unit}^-_L$$

$$\frac{\Gamma, x : A; \Omega \Uparrow \vdash C \Rightarrow^- t \mid \Delta \qquad A \text{ not LEFT ASYNC}}{\Gamma; \Omega, x : A \Uparrow \vdash C \Rightarrow^- t \mid \Delta} \Uparrow^-_L$$

Figure 3.7: LEFT ASYNC rules of the focused subtractive synthesis calculus

The LEFT ASYNC phase of focusing (Figure 3.7) breaks down the assumptions in Ω via the application of left rules. The $\Uparrow^-_L$ rule moves an assumption from Γ to Ω if the type is not left asynchronous.

$$\frac{\Gamma; \emptyset \vdash C \Downarrow \Rightarrow^- t \mid \Delta \qquad C \text{ not atomic}}{\Gamma; \emptyset \Uparrow \vdash C \Rightarrow^- t \mid \Delta} \text{FOCUS}^-_R \qquad \frac{\Gamma; x : A \Downarrow \vdash C \Rightarrow^- t \mid \Delta}{\Gamma, x : A; \emptyset \Uparrow \vdash C \Rightarrow^- t \mid \Delta} \text{FOCUS}^-_L$$

Figure 3.8: FOCUS rules of the focused subtractive synthesis calculus

The next set of rules are the FOCUS rules (Figure 3.8) which are used to determine what synchronous phase should be entered based on the type. In both rules, Ω is empty as a consequence of repeatedly

applying left rules in the LEFT ASYNC phase until each assumption is broken into assumptions without left asynchronous types. The FOCUS^-_R rule requires that the goal type not be atomic (i.e. a type variable), as the RIGHT SYNC phase comprises right rules for types. Similarly, the FOCUS^-_L requires that Γ not be empty.

$$\frac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow^- t_1 \mid \Delta_1 \qquad \Delta_1;\varnothing \vdash B \Downarrow \Rightarrow^- t_2 \mid \Delta_2}{\Gamma;\varnothing \vdash A \otimes B \Downarrow \Rightarrow^- (t_1, t_2) \mid \Delta_2}\otimes^-_R$$

$$\frac{\Gamma;\varnothing \vdash B \Downarrow \Rightarrow^- t \mid \Delta}{\Gamma;\varnothing \vdash A \oplus B \Downarrow \Rightarrow^- \mathbf{inr}\, t \mid \Delta}\oplus 2^+_L \qquad \frac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow^- t \mid \Delta}{\Gamma;\varnothing \vdash A \oplus B \Downarrow \Rightarrow^- \mathbf{inl}\, t \mid \Delta}\oplus 1^+_L$$

$$\frac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow^- t \mid \Delta}{\Gamma;\varnothing \vdash \Box_r A \Downarrow \Rightarrow^- t \mid \Gamma - r \cdot (\Gamma - \Delta)}\Box^-_R$$

$$\frac{}{\Gamma;\varnothing \vdash \mathsf{Unit} \Downarrow \Rightarrow^- () \mid \Gamma}\mathsf{Unit}^-_R \qquad \frac{\Gamma;\varnothing \vdash A \Uparrow \Rightarrow^- t \mid \Delta}{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow^- t \mid \Delta}\Downarrow^-_R$$

Figure 3.9: RIGHT SYNC rules of the focused subtractive synthesis calculus

The RIGHT SYNC rules are given by Figure 3.9, comprising the right rules for $\otimes$, $\oplus$, $\Box$, and Unit. The rule $\Downarrow^-_\text{R}$ switches back to a RIGHT ASYNC phase if the other rules cannot be applied.

$$\frac{\Gamma;x_2 : B \Downarrow \vdash C \Rightarrow^- t_1 \mid \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1;\varnothing \vdash A \Downarrow \Rightarrow^- t_2 \mid \Delta_2}{\Gamma;x_1 : A \multimap B \Downarrow \vdash C \Rightarrow^- [(x_1\, t_2)/x_2]t_1 \mid \Delta_2}\multimap^-_L$$

$$\frac{\Gamma;x :_s A, y : A \Downarrow \vdash B \Rightarrow^- t \mid \Delta, x :_{s'} A \qquad y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s+1}{\Gamma;x :_r A \Downarrow \vdash B \Rightarrow^- [x/y]t \mid \Delta, x :_{s'} A}\textsc{der}^-$$

$$\frac{\Gamma;x : A \Uparrow \vdash C \Rightarrow^- t \mid \Delta \qquad A \text{ not atomic and not LEFT SYNC}}{\Gamma;x : A \Downarrow \vdash C \Rightarrow^- t \mid \Delta}\Downarrow^-_L$$

Figure 3.10: LEFT SYNC rules of the focused subtractive synthesis calculus

$$\frac{}{\Gamma;x : A \Downarrow \vdash A \Rightarrow^- x \mid \Gamma}\textsc{LinVar}^- \qquad \frac{\exists s.\, r \sqsubseteq s+1}{\Gamma;x :_r A \Downarrow \vdash A \Rightarrow^- x \mid \Gamma, x :_s A}\textsc{GrVar}^-$$

Figure 3.11: VAR rules of the focused subtractive synthesis calculus

Finally the LEFT SYNC set (Figure 3.10) contains the focused $\multimap^-_\text{L}$ rule while the VAR set (Figure 3.11) contains the focused forms of $\textsc{LinVar}^-$ and $\textsc{GrVar}^-$. Both sets allow for transition to LEFT ASYNC via the $\Downarrow^-_\text{L}$ rule.

This focused form of the additive synthesis calculi follows an identical scheme, given by Figures 3.12-3.17.

$$\frac{\Gamma;\Omega, x : A \vdash B \Uparrow \Rightarrow t \mid \Delta, x : A}{\Gamma;\Omega \vdash A \multimap B \Uparrow \Rightarrow \lambda x.t \mid \Delta} \multimap^+_R \qquad \frac{\Gamma;\Omega \Uparrow \vdash C \Rightarrow t \mid \Delta \qquad C \text{ not RIGHT ASYNC}}{\Gamma;\Omega \vdash C \Uparrow \Rightarrow t \mid \Delta} \Uparrow^+_R$$

Figure 3.12: RIGHT ASYNC rules of the focused additive synthesis calculus

$$\frac{\Gamma;\Omega, x_1 : A, x_2 : B \vdash C \Rightarrow t_2 \mid \Delta, x_1 : A, x_2 : B}{\Gamma;\Omega, x_3 : A \otimes B \vdash C \Rightarrow \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta, x_3 : A \otimes B} \otimes^+_L$$

$$\frac{\Gamma;\Omega, x_2 : A \Uparrow \vdash C \Rightarrow t_1 \mid \Delta_1, x_2 : A \qquad \Gamma;\Omega, x_3 : B \Uparrow \vdash C \Rightarrow t_2 \mid \Delta_2, x_3 : B}{\Gamma;\Omega, x_1 : A \oplus B \Uparrow \vdash C \Rightarrow^- \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcup \Delta_2, x_1 : A \oplus B} \oplus^+_L$$

$$\frac{\Gamma;\Omega, x_2 :_r A \Uparrow \vdash B \Rightarrow t \mid \Delta \qquad \textit{if } x_2 :_s A \in \Delta \textit{ then } s \sqsubseteq r \textit{ else } 0 \sqsubseteq r}{\Gamma;\Omega, x_1 : \Box_r A \vdash B \Rightarrow \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid (\Delta \backslash x_2), x_1 : \Box_r A} \Box^+_L$$

$$\frac{\Gamma;\emptyset \vdash C \Rightarrow t \mid \Delta}{\Gamma; x : \mathsf{Unit} \vdash C \Rightarrow \mathbf{let}\,() = x\,\mathbf{in}\,t \mid \Delta, x : \mathsf{Unit}}\ \mathsf{Unit}^+_L$$

$$\frac{\Gamma, x : A;\Omega \Uparrow \vdash C \Rightarrow t \mid \Delta \qquad A \text{ not LEFT ASYNC}}{\Gamma;\Omega, x : A \Uparrow \vdash C \Rightarrow t \mid \Delta} \Uparrow^+_L$$

Figure 3.13: LEFT ASYNC rules of the focused additive synthesis calculus

$$\frac{\Gamma;\emptyset \vdash C \Downarrow \Rightarrow t \mid \Delta \qquad C \text{ not atomic}}{\Gamma;\emptyset \Uparrow \vdash C \Rightarrow t \mid \Delta}\ \textsc{focus}^+_R$$

$$\frac{\Gamma; x : A \Downarrow \vdash C \Rightarrow t \mid \Delta}{\Gamma, x : A;\emptyset \Uparrow \vdash C \Rightarrow t \mid \Delta}\ \textsc{focus}^+_L$$

Figure 3.14: FOCUS rules of the focused additive synthesis calculus

$$\frac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t_1 \mid \Delta_1 \qquad \Gamma;\varnothing \vdash B \Downarrow \Rightarrow t_2 \mid \Delta_2}{\Gamma;\varnothing \vdash A \otimes B \Downarrow \Rightarrow (t_1, t_2) \mid \Delta_1 + \Delta_2}\ \otimes_R^+$$

$$\frac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta}{\Gamma;\varnothing \vdash A \oplus B \Downarrow \Rightarrow \mathbf{inl}\, t \mid \Delta}\ \oplus 1_L^+ \qquad \frac{\Gamma;\varnothing \vdash B \Downarrow \Rightarrow t \mid \Delta}{\Gamma;\varnothing \vdash A \oplus B \Downarrow \Rightarrow \mathbf{inr}\, t \mid \Delta}\ \oplus 2_L^+$$

$$\frac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta}{\Gamma;\varnothing \vdash \Box_r A \Downarrow \Rightarrow [t] \mid r \cdot \Delta}\ \Box_R^+ \qquad \frac{}{\Gamma;\varnothing \vdash \mathsf{Unit} \Rightarrow () \mid \varnothing}\ \mathsf{Unit}_R^+$$

$$\frac{\Gamma;\varnothing \vdash A \Uparrow \Rightarrow t \mid \Delta}{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta}\ \Downarrow_R^+$$

Figure 3.15: RIGHT SYNC rules of the focused additive synthesis calculus

LEFTSYNC

$$\frac{\Gamma;x_2 : B \Downarrow \vdash C \Rightarrow t_1 \mid \Delta_1, x_2 : B \qquad \Gamma;\varnothing \vdash A \Downarrow \Rightarrow t_2 \mid \Delta_2}{\Gamma;x_1 : A \multimap B \Downarrow \vdash C \Rightarrow [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap_L^+$$

$$\frac{\Gamma;x :_s A, y : A \Downarrow \vdash B \Rightarrow t \mid \Delta, y : A}{\Gamma;x :_s A \Downarrow \vdash B \Rightarrow [x/y]t \mid \Delta + x :_1 A}\ \textsc{der}^+$$

$$\frac{\Gamma;x : A \Uparrow \vdash C \Rightarrow t \mid \Delta \qquad A \text{ not atomic and not LEFT SYNC}}{\Gamma;x : A \Downarrow \vdash C \Rightarrow t \mid \Delta}\ \Downarrow_L^+$$

Figure 3.16: LEFT SYNC rules of the focused additive synthesis calculus

$$\frac{}{\Gamma;x : A \vdash A \Rightarrow x \mid x : A}\ \textsc{LinVar}^+ \qquad \frac{}{\Gamma;x :_r A \vdash A \Rightarrow x \mid x :_1 A}\ \textsc{GrVar}^+$$

Figure 3.17: VAR rules of the focused additive synthesis calculus

$$\frac{\Gamma;x_2 : B \Downarrow \vdash C \Rightarrow^{\pm} t_1 \mid \Delta_1, x_2 : B \qquad \Gamma - \Delta_1;\varnothing \vdash A \Downarrow \Rightarrow^{\pm} t_2 \mid \Delta_2}{\Gamma;x_1 : A \multimap B \Downarrow \vdash C \Rightarrow^{\pm} [(x_1\, t_2)/x_2] \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap_L^{\pm}$$

$$\frac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow^{\pm} t_1 \mid \Delta_1 \qquad \Gamma - \Delta_1 \vdash B \Downarrow \Rightarrow^{\pm} t_2 \mid \Delta_2}{\Gamma;\varnothing \vdash A \otimes B \Downarrow \Rightarrow^{\pm} (t_1, t_2) \mid \Delta_1 + \Delta_2}\ \otimes_R^{\pm}$$

Figure 3.18: Rules of the focused additive pruning synthesis calculus

One way to view focusing is in terms of a finite state machine given in Figure 3.19. States comprise the four phases of focusing, plus

two additional states, FOCUS, and VAR. Edges are then the synthesis rules that direct the transition between focusing phases. The transitions between these focusing phases are handled by dedicated focusing rules for each transition. For the asynchronous phases, the $\Uparrow_R$ handles the transition between RIGHT ASYNC to LEFT ASYNC phases, while the $\Uparrow_L$ handles the transition from LEFT ASYNC to FOCUS phases. Conversely, the $\Downarrow R$ rule deals with the transition from a RIGHT SYNC phase back to a RIGHT ASYNC phase, with the $\Downarrow L$ rule likewise transitioning to a LEFT ASYNC phase. Depending on the current phase of focusing, these rules consider the goal type, the type of the assumption currently being focused on, as well as the size of Ω, to decide when to transition between FOCUS phases.

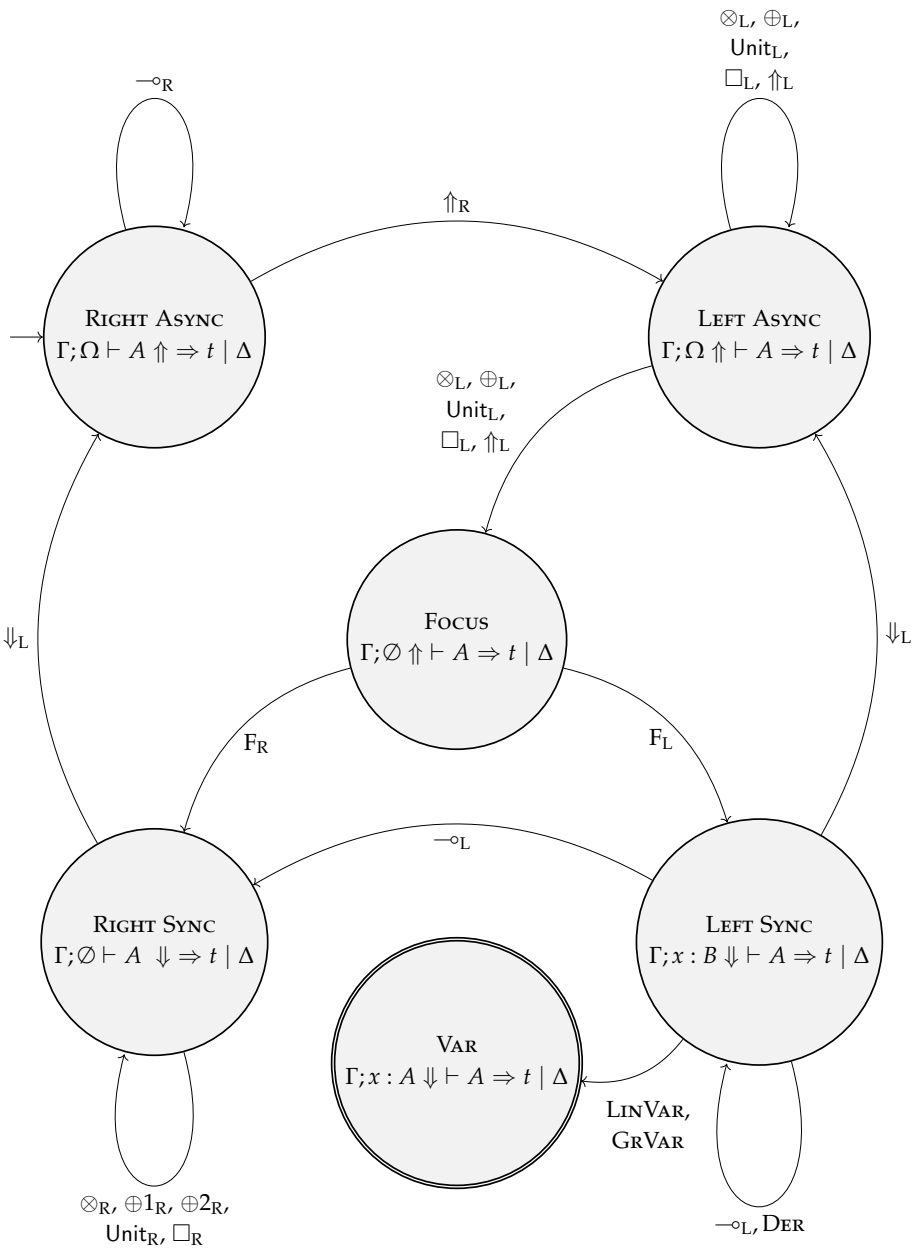

Figure 3.19: Focusing State Machine

We conclude with three key results: that applying focusing is sound for the subtractive (Lemma 3.5.1) and additive (Lemma 3.5.2), and additive pruning (Lemma 3.5.3) synthesis calculi. The proofs are contained in Sections B.1.4, B.1.5, and B.1.6 of Appendix B, respectively.

Lemma 3.5.1 (Soundness of focusing for subtractive synthesis)**.** For all contexts Γ, Ω and types A, B then:

1. RIGHT ASYNC: $\Gamma; \Omega \vdash A \Uparrow \Rightarrow^{-} t \mid \Delta \iff \Gamma, \Omega \vdash A \Rightarrow^{-} t \mid \Delta$
2. LEFT ASYNC: $\Gamma; \Omega \Uparrow \vdash B \Rightarrow^{-} t \mid \Delta \iff \Gamma, \Omega \vdash B \Rightarrow^{-} t \mid \Delta$
3. RIGHT SYNC: $\Gamma; \varnothing \vdash A \Downarrow \Rightarrow^{-} t \mid \Delta \iff \Gamma \vdash A \Rightarrow^{-} t \mid \Delta$
4. LEFT SYNC: $\Gamma; x : A \Downarrow \vdash B \Rightarrow^{-} t \mid \Delta \iff \Gamma, x : A \vdash B \Rightarrow^{-} t \mid \Delta$
5. FOCUS RIGHT: $\Gamma; \varnothing \Uparrow \vdash B \Rightarrow^{-} t \mid \Delta \iff \Gamma \vdash B \Rightarrow^{-} t \mid \Delta$
6. FOCUS LEFT: $\Gamma, x : A; \varnothing \Uparrow \vdash C \Rightarrow^{-} t \mid \Delta \iff \Gamma, x : A \vdash B \Rightarrow^{-} t \mid \Delta$

i.e. t has type A under context Δ, which contains assumptions with grades reflecting their use in t.

Lemma 3.5.2 (Soundness of focusing for additive synthesis)**.** For all contexts Γ, Ω and types A, B then:

1. RIGHT ASYNC: $\Gamma; \Omega \vdash A \Uparrow \Rightarrow^{+} t \mid \Delta \iff \Gamma, \Omega \vdash A \Rightarrow^{+} t \mid \Delta$
2. LEFT ASYNC: $\Gamma; \Omega \Uparrow \vdash A \Rightarrow^{+} t \mid \Delta \iff \Gamma, \Omega \vdash B \Rightarrow^{+} t \mid \Delta$
3. RIGHT SYNC: $\Gamma; \varnothing \vdash A \Downarrow \Rightarrow^{+} t \mid \Delta \iff \Gamma \vdash A \Rightarrow^{+} t \mid \Delta$
4. LEFT SYNC: $\Gamma; x : A \Downarrow \vdash B \Rightarrow^{+} t \mid \Delta \iff \Gamma, x : A \vdash B \Rightarrow^{+} t \mid \Delta$
5. FOCUS RIGHT: $\Gamma; \varnothing \Uparrow \vdash A \Rightarrow^{+} t \mid \Delta \iff \Gamma \vdash B \Rightarrow^{+} t \mid \Delta$
6. FOCUS LEFT: $\Gamma, x : A; \varnothing \Uparrow \vdash B \Uparrow \Rightarrow^{+} t \mid \Delta \iff \Gamma, x : A \mid B \Rightarrow^{+} t \mid \Delta$

i.e. t has type A under context Δ, which contains assumptions with grades reflecting their use in t.

Lemma 3.5.3 (Soundness of focusing for additive pruning synthesis)**.** For all contexts Γ, Ω and types A, B then:

1. RIGHT ASYNC: $\Gamma; \Omega \vdash A \Uparrow \Rightarrow^{\pm} t \mid \Delta \iff \Gamma, \Omega \vdash A \Rightarrow^{\pm} t \mid \Delta$
2. LEFT ASYNC: $\Gamma; \Omega \Uparrow \vdash A \Rightarrow^{\pm} t \mid \Delta \iff \Gamma, \Omega \vdash B \Rightarrow^{\pm} t \mid \Delta$
3. RIGHT SYNC: $\Gamma; \varnothing \vdash A \Downarrow \Rightarrow^{\pm} t \mid \Delta \iff \Gamma \vdash A \Rightarrow^{\pm} t \mid \Delta$
4. LEFT SYNC: $\Gamma; x : A \Downarrow \vdash B \Rightarrow^{\pm} t \mid \Delta \iff \Gamma, x : A \vdash B \Rightarrow^{\pm} t \mid \Delta$
5. FOCUS RIGHT: $\Gamma; \varnothing \Uparrow \vdash A \Rightarrow^{\pm} t \mid \Delta \iff \Gamma \vdash B \Rightarrow^{\pm} t \mid \Delta$
6. FOCUS LEFT: $\Gamma, x : A; \varnothing \Uparrow \vdash B \Uparrow \Rightarrow^{\pm} t \mid \Delta \iff \Gamma, x : A \mid B \Rightarrow^{\pm} t \mid \Delta$

i.e. t has type A under context Δ, which contains assumptions with grades reflecting their use in t.

3.6 IMPLEMENTATION

We implemented our approach as a synthesis tool for Granule, integrated with its core tool. Granule features ML-style polymorphism (rank-1 polymorphism) but we do not address polymorphism here (Chapter 4 offers a treatment of synthesis of polymorphic programs). Instead, programs are synthesised from type schemes treating universal type variables as logical atoms. Multiplicative products are primitive in Granule, although additive coproducts are provided via ADTs, from which we define a core sum type to use defined:

```
data Either a b = Left a | Right b
```

Constraints on resource usage are handled via Granule's existing symbolic engine, which compiles constraints on grades (for various semirings) to the SMT-lib format for Z3 [de Moura and Bjørner, 2008]. In the case of graded variable synthesis in the subtractive scheme, the kind of the assumption's grade (i.e., what semiring it belongs to) is inferred using Granule's type checker, which is used to generate an existential variable representing the remaining available usage of the graded assumption.

We use the LogicT monad for backtracking *enumerative* search of the space of candidate programs [Kiselyov et al., 2005] and the Scrap Your Reprinter library for splicing synthesised code into syntactic "holes" (represented by `?` in Granule), preserving the rest of the program text [Clarke et al., 2017].

The synthesis procedure can terminate in one of three ways: either (i) a type-and-grade correct program term is synthesised and returned, (ii) the search space of potential programs is exhausted without identifying a solution, or (iii) the synthesis procedure times out after some running for some user configurable length of time. As a type may have multiple inhabitants, the user may also request the "next" program after candidate solution is returned. The tool will discard the current solution and proceed until another candidate is found (if one exists).

3.6.1 *Post-Synthesis Resugaring*

A synthesised term often contains some artefacts of the fact that it was constructed automatically. The structure of our synthesis rules means aspects of our synthesised programs are not representative in some stylistic ways of the kind of programs functional programmers typically write. We consider two examples of these below using Granule code, and show how we apply a refactoring (or "re-sugaring") procedure to any synthesised term to rewrite them in a more idiomatic style.

3.6.1.1 *Abstractions*

A function definition synthesised from a function type using the $\multimap_R$ will take the form of a sequence of nested abstractions which bind the function's arguments, with the sub-term of the innermost abstraction containing the function body, e.g.

```
pair : ∀ { a b : Type } . a → b → (a, b)
pair = λx → λy → (x, y)
```

In most cases, a programmer would write a function definition as a series of equations with the function arguments given as patterns. Our refactoring procedure collects the outermost abstractions of a

synthesised term and transforms them into equation-level patterns with the innermost abstraction body forming the equation body:

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

3.6.1.2 *Unboxing*

An unboxing term is synthesised via the $\Box_L^-$ abd $\Box_L^+$ rules rule as a let expression which pattern matches over a box pattern, yielding an assumption with the grade's usage. Such terms can also be refactored both into function equations and to avoid excessive use of let bindings:

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

which we can then refactor into

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

This procedure includes programs which perform a nested unboxing:

```
comp : ∀ {k : Semiring, n m : k, a b c : Type}
     . (a [m] → b) [n]
     → (b [n] → c)
     → a [n ∗ m]
     → c
comp x y z = let [u] = x in let [v] = z in y [ u [v] ]
```

is refactored into:

```
comp : ∀ {k : Semiring, n m : k, a b c : Type}
     . (a [m] → b) [n]
     → (b [n] → c)
     → a [n ∗ m]
     → c
comp [u] y [v] = y [ u [v] ]
```

3.7 EVALUATING THE SYNTHESIS CALCULI

Prior to evaluation, we made the following hypotheses about the relative performance of the additive versus subtractive approaches:

H1. (**Solving; Additive requires less**) Additive synthesis should make fewer calls to the solver, with lower complexity theorems (fewer quantifiers). Dually, subtractive synthesis makes more calls to the solver with higher complexity theorems.

H2. (**Paths; Subtractive explores fewer**) For complex problems, additive will explore more paths as it cannot tell whether a variable is not well-resourced until closing a binder; additive pruning and subtractive will explore fewer paths as they can fail sooner.

H3. (**Performance; additive faster on simpler examples**) A corollary of the above two: simple examples will likely be faster in additive mode, but more complex examples will be faster in subtractive.

3.7.1 *Methodology*

To evaluate our synthesis tool we developed a suite of benchmarks comprising Granule type schemes for a variety of operations using linear and graded modal types. We divide our benchmarks into several classes of problem:

- **Hilbert**: the Hilbert-style axioms of intuitionistic logic (including SKI combinators), with appropriate $\mathbb{N}$ and $\mathbb{N}$-intervals grades where needed (see, e.g., S combinator in Example 2.3.1 or coproduct elimination in Example 3.1.1).
- **Comp**: various translations of function composition into linear logic: multiplicative, call-by-value and call-by-name using ! [Girard, 1987], o/1 translation of intuitionistic logic using ! [Liang and Miller, 2009], and coKleisli composition over $\mathbb{N}$ and arbitrary semirings: e.g. $\forall r, s \in \mathcal{R}$:

$$\mathit{comp\text{-}coK}_{\mathcal{R}} : \Box_r(\Box_s A \multimap B) \multimap (\Box_r B \multimap C) \multimap \Box_{r \cdot s} A \multimap C$$

- **Dist**: distributive laws of various graded modalities over functions, sums, and products, e.g., $\forall r \in \mathbb{N}$, or $\forall r \in \mathcal{R}$ in any semiring, or $r = 0...\infty$:

$$\mathit{pull}_{\oplus} : (\Box_r A \oplus \Box_r B) \multimap \Box_r (A \oplus B)$$

$$\mathit{push}_{\multimap} : \Box_r (A \multimap B) \multimap \Box_r A \multimap \Box_r B$$

- **Vec**: map operations on fixed size vectors encoded as products, e.g.:

$$\begin{aligned} \mathit{vmap}_5 : \quad & \Box_5 (A \multimap B) \\ \multimap \quad & ((((A \otimes A) \otimes A) \otimes A) \otimes A) \\ \multimap \quad & ((((B \otimes B) \otimes B) \otimes B) \otimes B) \end{aligned}$$

- **Misc**: includes Example 3.1.2 (information-flow security) and functions which must share resources between graded modalities, e.g.:

$$\begin{aligned} \mathit{share} : \quad & \Box_4 A \\ \multimap \quad & \Box_6 A \\ \multimap \quad & \Box_2 (((((A \otimes A) \otimes A) \otimes A) \otimes A) \multimap B) \\ \multimap \quad & (B \otimes B) \end{aligned}$$

Hilbert	
⊗Intro	$\otimes_i : \forall a, b.\ a \multimap b \multimap (a \otimes b)$
⊗Elim	$\otimes_{e1} : \forall a, b.\ (a \otimes \Box_0 b) \multimap a$
	$\otimes_{e2} : \forall a, b.\ (\Box_0 a \otimes b) \multimap b$
⊕Intro	$\oplus_{i1} : \forall a, b.\ a \multimap a \oplus b$
	$\oplus_{i2} : \forall a, b.\ b \multimap a \oplus b$
⊕Elim	$\oplus_e : \forall a, b, c.\ \Box_{0\ldots1}(a \multimap c) \multimap \Box_{0\ldots1}(b \multimap c) \multimap (a \oplus b) \multimap c$
SKI	$s : \forall a, b, c.\ (a \multimap (b \multimap c)) \multimap (a \multimap b) \multimap (\Box_2 a \multimap c)$
	$k : \forall a, b.\ a \multimap \Box_0 b \multimap a$
	$i : \forall a.\ a \multimap a$
Comp	
0/1	$\circ_{I/O} : \forall a, b, c.\ \Box(\Box a \multimap \Box b) \multimap \Box(\Box b \multimap \Box c) \multimap \Box(\Box a \multimap c)$
CBN	$\circ_{\text{CBN}} : \forall a, b, c.\ \Box(\Box a \multimap b) \multimap \Box(\Box b \multimap c) \multimap \Box a \multimap c$
CBV	$\circ_{\text{CBV}} : \forall a, b, c.\ \Box(\Box a \multimap \Box b) \multimap \Box(\Box b \multimap \Box c) \multimap \Box\Box a \multimap \Box c$
coK-$\mathcal{R}$	$\circ_{\mathcal{R}} : \forall \mathcal{R}, r, s \in \mathcal{R}, a, b, c.\ \Box_r(\Box_s a \multimap b) \multimap (\Box_r b \multimap c) \multimap \Box_{r\cdot s} a \multimap c$
lin	$\circ : \forall a, b, c.\ (a \multimap b) \multimap (b \multimap c) \multimap (a \multimap c)$
coK-$\mathbb{N}$	$\circ_{\mathbb{N}} : \forall r, s \in \mathbb{N}, a, b, c.\ \Box_r(\Box_s a \multimap b) \multimap (\Box_r b \multimap c) \multimap \Box_{r\cdot s} a \multimap c$
Dist	
⊕-$\mathbb{N}$	$pull_{\oplus} : \forall r : \mathbb{N}, a, b.\ (\Box_r a \oplus \Box_r b) \multimap \Box_r (a \oplus b)$
⊕-!	$pull_{\oplus} : \forall a, b.\ (\Box a \oplus \Box b) \multimap \Box (a \oplus b)$
⊕-$\mathcal{R}$	$pull_{\oplus} : \forall \mathcal{R}, r \in \mathcal{R}, a, b.\ (\Box_r a \oplus \Box_r b) \multimap \Box_r (a \oplus b)$
⊗-$\mathbb{N}$	$pull_{\otimes} : \forall r : \mathbb{N}, a, b.\ (\Box_r a \otimes \Box_r b) \multimap \Box_r (a \otimes b)$
⊗-!	$pull_{\otimes} : \forall a, b.\ (\Box a \otimes \Box b) \multimap \Box (a \otimes b)$
⊗-$\mathbb{R}$	$pull_{\otimes} : \forall \mathcal{R}, r, a, b.\ (\Box_r a \otimes \Box_r b) \multimap \Box_r (a \otimes b)$
—∘-$\mathbb{N}$	$push_{\circ} : \forall r : \mathbb{N}, a, b.\ \Box_r (a \multimap b) \multimap \Box_r a \multimap \Box_r b$
—∘-!	$push_{\circ} : \forall a, b.\ \Box (a \multimap b) \multimap \Box a \multimap \Box b$
—∘-$\mathcal{R}$	$push_{\circ} : \forall \mathcal{R}, r : \mathcal{R}, a, b.\ \Box_r (a \multimap b) \multimap \Box_r a \multimap \Box_r b$
Vec	
vec5	$vmap_5 : \forall a, b.\ \Box_5 (a \multimap b) \multimap ((((a \otimes a) \otimes a) \otimes a) \otimes a) \multimap ((((b \otimes b) \otimes b) \otimes b) \otimes b)$
vec10	$vmap_{10} : \forall a, b.$ *as above but for 10-tuples*
vec15	$vmap_{15} : \forall a, b.$ *as above but for 15-tuples*
vec20	$vmap_{20} : \forall a, b.$ *as above but for 20-tuples*
Misc	
split⊕	$split : \forall a, b, c. \Box_{2\ldots3} b \multimap (a \oplus c) \multimap ((a \otimes \Box_{2..2} b) \oplus (c \otimes \Box_{3\ldots3} b))$
split⊗	$split : \forall a, b. \Box_{0\ldots2} (a \multimap a \multimap a) \multimap \Box_{10\ldots10} a \multimap (\Box_{2\ldots2} a \otimes \Box_{6\ldots6} a)$
share	$share : \forall a, b. \Box_4 a \multimap \Box_6 a \multimap \Box_2 (((((a \otimes a) \otimes a) \otimes a) \otimes a) \otimes a) \multimap b) \multimap (b \otimes b)$
Exm. 3.1.2	$noLeak : \forall a, b. (\Box_{\text{Lo}} a \otimes \Box_{\text{Hi}} a) \multimap (\Box_{\text{Lo}} (a \otimes 1) \multimap b) \multimap b$

Table 3.3: List of benchmark synthesis problems

Table 3.3 provides the complete list of Granule type schemes used for these synthesis problems (32 in total). Note that these are type schemes which quantify over type variables (a, and b), however, we simply treat each type variable as a logical atom, unifiable only with itself. Chapter 4 provides a proper treatment of synthesis from Rank-1 polymorphic type schemes. Note also that $\Box A$ is used as shorthand for $\Box_{0\ldots\infty} A$ (graded modality with indices drawn from intervals over $\mathbb{N} \cup \{\infty\}$). The complete synthesised program code for the benchmarking problems can be found in Section A.1 of Appendix A

We found that Z3 is highly variable in its solving time, so timing measurements are computed as the mean of 20 trials. We used Z3 version 4.8.8 on a Linux laptop with an Intel i7-8665u @ 4.8 Ghz and 16 Gb of RAM.

		Additive			Additive (pruning)			Subtractive		
	Problem		μT (ms)	N		μT (ms)	N		μT (ms)	N
Hilbert	$\otimes$Intro	✓	6.69 (0.05)	2	✓	9.66 (0.23)	2	✓	10.93 (0.31)	2
	$\otimes$Elim	✓	0.22 (0.01)	0	✓	0.05 (0.00)	0	✓	0.06 (0.00)	0
	$\oplus$Intro	✓	0.08 (0.00)	0	✓	0.07 (0.00)	0	✓	0.07 (0.00)	0
	$\oplus$Elim	✓	7.26 (0.30)	2	✓	13.25 (0.58)	2	✓	204.50 (8.78)	15
	SKI	✓	8.12 (0.25)	2	✓	24.98 (1.19)	2	✓	41.92 (2.34)	4
Comp	01	✓	28.31 (3.09)	5	✓	41.86 (0.38)	5	×	Timeout	-
	cbn	✓	13.12 (0.84)	3	✓	26.24 (0.27)	3	×	Timeout	-
	cbv	✓	19.68 (0.98)	5	✓	34.15 (0.98)	5	×	Timeout	-
	$\circ coK_{\mathcal{R}}$	✓	33.37 (2.01)	2	✓	27.37 (0.78)	2	×	92.71 (2.37)	8
	$\circ coK_{\mathbb{N}}$	✓	27.59 (0.67)	2	✓	21.62 (0.59)	2	×	95.94 (2.21)	8
	mult	✓	0.29 (0.02)	0	✓	0.12 (0.00)	0	✓	0.11 (0.00)	0
Dist	$\otimes$-!	✓	12.96 (0.48)	2	✓	32.28 (1.32)	2	✓	10487.92 (4.38)	7
	$\otimes$-$\mathbb{N}$	✓	24.83 (1.01)	2	×	32.18 (0.80)	2	×	31.33 (0.65)	2
	$\otimes$-$\mathcal{R}$	✓	28.17 (1.01)	2	×	29.72 (0.90)	2	×	31.91 (1.02)	2
	$\oplus$-!	✓	7.87 (0.23)	2	✓	16.54 (0.43)	2	✓	160.65 (2.26)	4
	$\oplus$-$\mathbb{N}$	✓	22.13 (0.70)	2	✓	30.30 (1.02)	2	×	23.82 (1.13)	1
	$\oplus$-$\mathcal{R}$	✓	22.18 (0.60)	2	✓	31.24 (1.40)	2	×	16.34 (0.40)	1
	$\multimap$-!	✓	6.53 (0.16)	2	✓	10.01 (0.25)	2	✓	342.52 (2.64)	4
	$\multimap$-$\mathbb{N}$	✓	29.16 (0.82)	2	✓	28.71 (0.67)	2	×	54.00 (1.53)	4
	$\multimap$-$\mathcal{R}$	✓	29.31 (1.84)	2	✓	27.44 (0.60)	2	×	61.33 (2.28)	4
Vec	vec5	✓	4.72 (0.07)	1	✓	14.93 (0.21)	1	✓	78.90 (2.25)	6
	vec10	✓	5.51 (0.36)	1	✓	20.81 (0.77)	1	✓	142.87 (5.86)	11
	vec15	✓	9.75 (0.25)	1	✓	22.09 (0.24)	1	✓	195.24 (3.20)	16
	vec20	✓	13.40 (0.46)	1	✓	30.18 (0.20)	1	✓	269.52 (4.25)	21
Misc	split$\oplus$	✓	3.79 (0.04)	1	✓	5.10 (0.16)	1	✓	10732.65 (8.01)	6
	split$\otimes$	✓	14.07 (1.01)	3	✓	46.27 (2.04)	3	×	Timeout	-
	share	✓	292.02 (11.37)	44	✓	100.85 (2.44)	6	✓	193.33 (4.46)	17
	Exm. 3.1.2	✓	8.09 (0.46)	2	✓	26.03 (1.21)	2	✓	284.76 (0.31)	3

Table 3.4: Results. μT in *ms* to 2 d.p. with standard sample error in brackets

3.7.2 *Results and Analysis*

For each synthesis problem, we recorded whether synthesis was successful or not (denoted ✓ or ×), the mean total synthesis time (μT), and the number of calls made to the SMT solver (N). Table 3.4 summarises the results with the fastest case for each benchmark highlighted. For all benchmarks that used the SMT solver, the solver accounted for 91.73% – 99.98% of synthesis time, so we report only the mean total synthesis time μT, rather than showing the SMT solver time separately as this gives a good proxy of the solver time. We set a timeout of 120 seconds.

3.7.2.1 *Additive versus subtractive*

As conjectured in H1, the additive approach generally synthesises programs with fewer calls to the solver than subtractive. Our first hypothesis holds for almost all benchmarks, with the subtractive approach often far exceeding the number made by the additive. This is explained by the difference in graded variable synthesis between approaches. In the additive, a constant grade 1 is given for graded assumptions in the output context, whereas in the subtractive, a fresh grade variable is created with a constraint on its usage which is checked immediately. As the total synthesis time is almost entirely spent in the SMT solver (more than 90%), solving constraints is by far the most costly part of synthesis leading to the additive approach synthesising most examples in a shorter amount of time.

Graded variable synthesis in the subtractive case also results in several examples failing to synthesise due to timeout. In some cases, e.g., the first three *comp* benchmarks, the subtractive approach times-out as synthesis diverges with constraints growing in size due to the maximality condition and absorbing behaviour of $0...\infty$ interval. In the case of *coK*-$\mathcal{R}$ and *coK*-$\mathbb{N}$, the generated constraints have the form $\forall r.\exists s.r \sqsupseteq s + 1$ which is not valid $\forall r \in \mathbb{N}$ (e.g., when $r = 0$), which suggests that the subtractive approach does not work well for polymorphic grades. As further work, we are considering an alternate rule for synthesising promotion with constraints of the form $\exists s.s = s' * r$, i.e., a multiplicative inverse constraint.

In more complex examples we see evidence to support our second hypothesis. The *share* problem requires a lot of graded variable synthesis which is problematic for the additive approach, for the reason described in H2 that synthesis may explore many paths which are incorrect in their eventual resource usage. In contrast, the subtractive approach performs better, with $\mu T = 193.3ms$ as opposed to additive's $292.02ms$. However, additive pruning outperforms both.

Notably, on examples which are purely linear such as *andElim* from Hilbert's axioms or *mult* for function composition, the subtractive approach generally performs better. Linear programs without graded modalities can be synthesised without the need to interface with Z3 at all, making the differences here somewhat negligible as solver time generally makes up for the vast proportion of total synthesis time in the graded benchmarks.

3.7.2.2 *Additive pruning*

The pruning variant of additive synthesis (where subtraction takes place in the premises of multiplicative rules) had mixed results compared to the default. In simpler examples, the overhead of pruning (requiring SMT solving) outweighs the benefits obtained from reduc-

ing the space. However, in more complex examples which involve synthesising many graded variables (e.g. *share*), pruning is especially powerful, performing better than the subtractive approach. However, additive pruning failed to synthesis two examples which are polymorphic in their grade ($\otimes$-$\mathbb{N}$) and in the semiring ($\otimes$-$\mathcal{R}$).

Overall, the additive approach outperforms the subtractive and is able to synthesise more examples, including ones polymorphic in grades and even the semiring. Given that the literature on linear logic theorem proving is typically subtractive, this is an interesting result.

3.7.3 *Completeness of Synthesis*

While we do not provide lemmas for completeness of synthesis, we hypothesise that this holds for the declarative focused forms of each of the calculi. The proof of this remains as future work.

In Table 3.4 one can observe examples which succeed in the Additive column yet fail to synthesise in the Subtractive and Additive Pruning columns even though they do not timeout ($\otimes$-$\mathbb{N}$, $\otimes$-$\mathcal{R}$, $\oplus$-$\mathbb{N}$, etc.). Rather than being the result of the synthesis rules themselves lacking completeness, this is instead due to limitations in how the implementation of the synthesis tool interacts with the SMT solver. The method of compiling constraints which existentially quantify over grades which are polymorphic (in both the natural numbers semiring and over semirings in general) is not expressive enough in the current system to convey the correct information to the solver. Correcting this is ongoing work which requires co-ordination with how Granule's type checker compiles constraints to the SMT-lib format.

3.8 CONCLUSION

At this point we have constructed a simple program synthesis tool for Granule, parameterised by a resource management scheme, which effectively deals with the problems of treating data as a resource inside a program. Both schemes would be a reasonable choice for further development of a synthesis tool for our language based on the graded linear λ-calculus.

Going forward, however, we focus primarily on the additive resource management scheme, using this as the basis for our more feature-rich fully-graded synthesis calculus in Chapter 4. The evaluation in Section 3.7 showed that the additive approach generally yields smaller and simpler theorems than the subtractive, requiring less time to solve. Theorem proving becomes even more prevalent in synthesis for a fully graded typing calculus - potentially every rule introduces new constraints that require solving, thus the speed at which this can be carried out is especially important.

While the tool presented in this chapter allows users to synthesise a considerable subset of Granule programs, our language is still limited in its expressivity when compared to the core of Granule. Data types comprise only product, sum, and unit types, while synthesis of recursive function definitions or functions which make use of other in-scope values such as top-level definitions is not permitted. One notable limitation of our typing calculi is the inability to express (and therefore synthesise) programs which perform a deep pattern match over a graded data type. A clear example of this can be found in the synthesis of programs which distribute a graded modality over a data type. Consider an example of a distributive program, *push*:

$$push : \Box_r (A \otimes B) \multimap \Box_r A \otimes \Box_r B$$

which takes a data type graded by r (in this case the product type $A \otimes B$), and distributes r over the constituent elements of the product A and B. This example is representative of a common class of graded programs known as *distribute laws*, however, we are unable to express such programs in our simplified language.

In Granule, however, we can write the above as:

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

by pattern matching on both the graded modality and the product in a single nested pattern match. The ability to perform this *deep* pattern matching greatly increases the expressivity of our language, and allows us to both type and potentially synthesis much more interesting and realistic Granule programs.

In the next chapter we show how we can synthesise such programs and other, more complex examples by incorporating pattern matching, algebraic data types (ADTs), and recursion into our typing calculus and synthesis tool.

4

AN EXTENDED SYNTHESIS CALCULUS

In Chapter 3 we showed two ways in which a calculus of type-directed synthesis rules can incorporate grades to ensure that synthesised programs are both well-typed and well-resourced. We compared the performance of these calculi against each other, however, we have yet to show that synthesis with grades makes the task of program synthesis easier, by reducing the search space of programs.

So far, we have considered a language with a several basic types, but which falls short of the full features that one would expect in a practical functional language. In this chapter, we consider a target language which is significantly more expressive than what we have seen thus far, constituting a fully-fledged functional programming language, with (recursive) ADTs, recursion, and polymorphic definitions.

We also take this chapter as an opportunity to explore synthesis in a fully graded type system. As mentioned in Chapter 2, the fully graded λ-calculus is one of the two dominant flavours of quantitative type system. This approach is common amongst implementations of quantitative type systems, such as Idris 2, the *Linear Types* language extension to GHC, and the *GradedBase* language extension of Granule (which forms the target language of our implementation). The majority of content of this chapter is derived from Hughes and Orchard [2024].

ROADMAP We begin by extending the calculus of Section 2.3.2, adding polymorphic recursive user-defined algebraic data types, as well as recursive function definitions with polymorphic type schemes. Section 4.1 provides a full formal description of our target language.

We then provide a program synthesis calculus with an additive resource management scheme for this language in Section 4.2, describing the rules in turn, as well as additional post-synthesis refactoring procedures as an extension of the procedures in Section 3.6.1 (Section 4.3). We then apply focusing to this calculus in Section 4.4, as we did in Chapter 3, using this focused form as the basis of the implementation of our synthesis tool.

We evaluate our implementation on a set of 46 benchmarks (Section 4.5), including several non-trivial programs which use algebraic data types and recursion.

In Section 4.6, to demonstrate the practicality and versatility of our approach, we apply our algorithm to synthesising programs in Haskell from type signatures that use GHC's *Linear Types* extension, bringing our approach to a mainstream language.

4.1 A FULLY GRADED TARGET LANGUAGE

We now formally define our target language, extending the fully graded λ-calculus of Chapter 2. Our language comprises the λ-calculus extended with grades and a graded necessity modality, arbitrary user-defined recursive algebraic data types (ADTs), as well as rank-1 polymorphism. The syntax of types is given by:

$$A, B ::= A^r \to B \mid K\,\vec{A} \mid \Box_r A \mid \mu X.A \mid X \mid \alpha \qquad \textit{(types)}$$
$$K ::= \mathsf{Unit} \mid \otimes \mid \oplus \qquad \textit{(type constructors)}$$
$$\kappa ::= \mathsf{Type} \mid \kappa_1 \to \kappa_2 \qquad \textit{(kinds)}$$
$$\tau ::= \forall \overline{\alpha : \kappa}.A \qquad \textit{(type schemes)}$$

Recursive types $\mu X.A$ are equi-recursive (although we also provide explicit typing rules) with type recursion variables X. Data constructors and other top-level definitions are typed by type schemes τ (rank-1 polymorphic types), which bind a set of kind-annotated universally quantified type variables $\overline{\alpha : \kappa}$ à la ML [Milner, 1978]. Thus, types may contain type variables α. Kinds κ are standard, given by Figure 4.1.

The syntax of terms is given by:

$$t ::= x \mid \lambda x.t \mid t_1\, t_2 \mid [t] \mid C\, t_1 \ldots t_n \mid \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; \ldots; p_n \mapsto t_n \qquad \textit{(terms)}$$
$$p ::= x \mid \_ \mid [p] \mid C\, p_1 \ldots p_n \qquad \textit{(patterns)}$$

Terms consist of a graded λ-calculus, a promotion construct $[t]$ which introduces a graded modality explicitly, as well as data constructor introduction ($C\, t_1 \ldots t_n$) and elimination via **case** expressions with patterns, which are defined via the syntax of patterns p. Top-level recursive function definitions are instead assumed to be in-scope during synthesis, with the user providing the set of definitions that the synthesis algorithm may use.

Typing judgements have the form $\Sigma; \Gamma \vdash t : A$ assigning a type A to a term t under a type variable context Σ and variable context Γ:

$$\Sigma ::= \emptyset \mid \Sigma, \alpha : \kappa \mid \Sigma, X : \mathsf{Type} \qquad \textit{(type variable contexts)}$$
$$\Delta, \Gamma ::= \emptyset \mid \Gamma, x :_r A \qquad \textit{(variable contexts)}$$

That is, a type variable context may be empty $\emptyset$, extended with a kind-annotated type variable $\alpha : \kappa$ or extended with a recursion variable

$$\frac{}{\Sigma, \alpha : \kappa \vdash \alpha : \kappa}\ \kappa_{\text{VAR}} \qquad \frac{\Sigma \vdash A : \text{Type}}{\Sigma \vdash \Box_r A : \text{Type}}\ \kappa_{\Box}$$

$$\frac{\Sigma \vdash A : \text{Type} \qquad \Sigma \vdash B : \text{Type}}{\Sigma \vdash A^r \to B : \text{Type}}\ \kappa_{\to}$$

$$\frac{\Sigma \vdash A : \kappa_1 \to \kappa_2 \qquad \Sigma \vdash B : \kappa_1}{\Sigma \vdash A\, B : \kappa_2}\ \kappa_{\text{APP}}$$

$$\frac{}{\Sigma \vdash \text{Unit} : \text{Type}}\ \kappa_{\text{UNIT}} \qquad \frac{\Sigma \vdash A : \text{Type} \qquad \Sigma \vdash B : \text{Type}}{\Sigma \vdash A \otimes B : \text{Type}}\ \kappa_{\otimes}$$

$$\frac{\Sigma \vdash A : \text{Type} \qquad \Sigma \vdash B : \text{Type}}{\Sigma \vdash A \oplus B : \text{Type}}\ \kappa_{\oplus}$$

$$\frac{}{\Sigma, X : \text{Type} \vdash X : \text{Type}}\ \kappa_X \qquad \frac{\Sigma, X : \text{Type} \vdash A : \text{Type}}{\Sigma \vdash \mu X.A : \text{Type}}\ \kappa_{\mu}$$

Figure 4.1: Kinding rules for the fully graded typing calculus

X. Recursion variables always have kind Type. Similarly, a variable context may be empty or extended with a *graded* assumption $x :_r A$. Graded assumptions must be used in a way which adheres to the constraints of the grade r. Structural exchange is permitted, allowing a context to be arbitrarily reordered. A global context D parametrises the system, containing top-level definitions and data constructors annotated with type schemes. This context is elided in the rules as it never changes.

Another judgment types top-level terms (definitions) with polymorphic type schemes:

$$\frac{\overline{\alpha : \kappa}; \emptyset \vdash t : A}{\emptyset; \emptyset \vdash t : \forall \overline{\alpha : \kappa}.A}\ \textsc{TopLevel}$$

This rule takes the type scheme and adds its universally quantified type variables to Σ, where they can be used subsequently in the typing rules. This rule corresponds to the generalisation step of typing polymorphic definitions [Milner, 1978]. The rule's premise then types the body at A, using the typing rules for terms of Figures 4.2, and 4.3 whose rules help explain the meaning of the syntax with reference to their static semantics. Figure 4.2 simply gives the rules from the fully graded λ-calculus of Section 2.3.2 adapted to include polymorphism, while Figure 4.3 gives the typing rules for the new additions.

The use of top-level definitions is typed by the DEF rule. The definition x must be present in the global definition context D, with the type scheme $\forall \overline{\alpha : \kappa}.A'$. The type A results from instantiating all of the

$$\frac{\Sigma \vdash A : \text{Type}}{\Sigma; 0 \cdot \Gamma, x :_1 A \vdash x : A}\ \textsc{Var}$$

$$\frac{\Sigma; \Gamma, x :_r A \vdash t : B}{\Sigma; \Gamma \vdash \lambda x.t : A^r \to B}\ \textsc{Abs} \qquad \frac{\Sigma; \Gamma_1 \vdash t_1 : A^r \to B \quad \Gamma_2 \vdash t_2 : A}{\Sigma; \Gamma_1 + r \cdot \Gamma_2 \vdash t_1\, t_2 : B}\ \textsc{App}$$

$$\frac{\Sigma; \Gamma \vdash t : A}{\Sigma; r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{Pr} \qquad \frac{\Sigma; \Gamma, x :_r A, \Gamma' \vdash t : B \quad r \sqsubseteq s}{\Sigma; \Gamma, x :_s A, \Gamma' \vdash t : B}\ \textsc{Approx}$$

Figure 4.2: Typing rules for the fully graded polymorphic λ-calculus

$$\frac{(x : \forall \overline{\alpha : \kappa}.A') \in D \quad \Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')}{\Sigma; 0 \cdot \Gamma \vdash x : A}\ \textsc{Def}$$

$$\frac{\begin{array}{l}(C : \forall \overline{\alpha : \kappa}.B_1'^{\,q_1} \to \ldots \to B_n'^{\,q_n} \to K\,\vec{A}') \in D \\ \Sigma \vdash B_1{}^{q_1} \to \ldots \to B_n{}^{q_n} \to K\,\vec{A} = \text{inst}(\forall \overline{\alpha : \kappa}.B_1'^{\,q_1} \to \ldots \to B_n'^{\,q_n} \to K\,\vec{A}')\end{array}}{\Sigma; 0 \cdot \Gamma \vdash C : B_1{}^{q_1} \to \ldots \to B_n{}^{q_n} \to K\,\vec{A}}\ \textsc{Con}$$

$$\frac{\Sigma; \Gamma \vdash t : A \quad \Sigma; r \vdash p_i : A \rhd \Delta_i \quad \Sigma; \Gamma', \Delta_i \vdash t_i : B}{\Sigma; r \cdot \Gamma + \Gamma' \vdash \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; \ldots; p_n \mapsto t_n : B}\ \textsc{Case}$$

$$\frac{\Sigma; \Gamma \vdash t : A[\mu X.A/X]}{\Sigma; \Gamma \vdash t : \mu X.A}\ \mu_1 \qquad \frac{\Sigma; \Gamma \vdash t : \mu X.A}{\Sigma; \Gamma \vdash t : A[\mu X.A/X]}\ \mu_2$$

Figure 4.3: Typing rules for the fully graded polymorphic calculus extensions

universal variables to types via the judgment $\Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')$ in a standard way as in Algorithm W [Milner, 1978].

The graded λ-calculus fragment of our language remains mostly the same as in Section 2.3.2, with the addition of pattern typing. Furthermore, the Var rule also checks the kind of the assumption x's type in the premise.

Recursion is typed via the μ_1 and μ_2 rules, in a standard way for equi-recursive types.

Introduction and elimination of data constructors is given by the Con and Case rules respectively, with Case also handling graded modality elimination via pattern matching, as in Section 5.2. For Con, we may type a data constructor C of some data type $K\,\vec{A}$ (with zero or more type parameters represented by $\vec{A}$) if it is present in the global context of data constructors D. Data constructors are closed requiring our context Γ to have zero-use grades, thus we scale Γ by 0. Elimination of data constructors take place via pattern matching over a constructor. Patterns p are typed by the judgement $r \vdash p : A \rhd \Delta$ which states that a pattern p has type A and produces a context of typed binders Δ. The grade r to the left of the turnstile represents the grade information arising from usage in the context generated by this pattern match. This is similar to the approach used for pattern typing

$$\frac{0 \sqsubseteq r \qquad \Sigma \vdash A : \text{Type}}{\Sigma; r \vdash \_ : A \rhd \emptyset}\ \textsc{PWild}$$

$$\frac{\Sigma \vdash A : \text{Type}}{\Sigma; r \vdash x : A \rhd x :_r A}\ \textsc{PVar} \qquad \frac{\Sigma; r \cdot s \vdash p : A \rhd \Gamma}{\Sigma; r \vdash [p] : \Box_s A \rhd \Gamma}\ \textsc{PBox}$$

$$\frac{\begin{array}{l}(C : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\,\vec{A'}) \in D \\ \Sigma \vdash {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K\,\vec{A} = \text{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\,\vec{A'}) \\ \Sigma; q_i \cdot r \vdash p_i : B_i \rhd \Gamma_i \qquad |K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq r\end{array}}{\Sigma; r \vdash C\,p_1 \ldots p_n : K\,\vec{A} \rhd \overrightarrow{\Gamma_i}}\ \textsc{PCon}$$

Figure 4.4: Pattern typing rules for the fully graded typing calculus

in Section 5.2, however, we note that we no longer have the linear forms of the rules.

The pattern typing rules are given by Figure 4.4.

Variable patterns are typed by PVar, which simply produces a singleton context containing an assumption $x :_r A$ from the variable pattern with any grade r. A wildcard pattern _, typed by the PWild rule, is only permissible with grades that allow for weakening, i.e., where $0 \sqsubseteq r$.

Pattern matching over data constructors is handled by the PCon rule. A data constructor may have up to zero or more sub-patterns ($p_1 \ldots p_n$), each of which is typed under the grade $q_i \cdot r$ (where q_i is the grade of corresponding argument type for the constructor, as defined in D). Additionally, we have the constraint $|K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq r$ which witnesses the fact that if there is more than one data constructor for the data type (written $|K\,\vec{A}| > 1$), then r must approximate 1 because pattern matching on a data constructor incurs some usage since it reveals information about that constructor.[1]

By contrast, pattern matching on a type with only one constructor cannot convey any information by itself and so no usage requirement is imposed. Finally, elimination of a graded modality (often called *unboxing*) takes place via the PBox rule, with syntax $[p]$. Like PCon, this rule propagates the grade information of the box pattern's type s to the enclosed sub-pattern p, yielding a context with the grades $r \cdot s$.

One may observe that PBox (and by extension Pr) could be considered as special cases of PCon (and Con respectively), if we were to treat our promotion construct as a data constructor with the type $A^r \to \Box_r A$. We find it helpful to keep explicit modality introduction and elimination distinct from constructors, however, particularly with regard to synthesis.

1 A discussion of this additional constraint on grades for case expressions is given in Section 6.3.4 of 5 comparing how this manifests in various approaches.

Note that Granule does not support as-patterns: each function argument contains at most one variable referencing it in the function body.

4.1.1 *Metatheory*

Lastly we note that the fully graded system also enjoys admissibility of substitution [Abel and Bernardy, 2020, Liepelt et al., 2024] which is critical in type preservation proofs, and is needed in our proof of soundness for synthesis:

Lemma 4.1.1 (Admissibility of substitution)**.** Let $\Delta \vdash t' : A$, then: If $\Gamma, x :_r A, \Gamma' \vdash t : B$ then $\Gamma + (r \cdot \Delta) + \Gamma' \vdash [t'/x]t : B$

4.2 A FULLY GRADED SYNTHESIS CALCULUS

Having defined the target language, we define our synthesis calculus, which uses the *additive* approach to resource management (see Section 3.4), with judgements:

$$\Sigma; \Gamma \vdash A \Rightarrow t \mid \Delta$$

That is, given an input context Γ, for goal type A we can synthesise the term t with the output context Δ describing how variables were used in t. As with the typing rules, top-level definitions and data constructors in scope are contained in a set D, which parametrises the system and is elided in the rules. Σ is a context of kind-annotated type variables, which we elide in rules where it is passed inductively to the premise(s). As in the additive scheme of Chapter 3, the graded context Δ need not use all the variables in Γ, nor with exactly the same grades.

We next present the synthesis calculus in stages, similarly to Chapter 3. Each type former of the core calculus (with the exception of type variables) has two corresponding synthesis rules: a right rule for introduction (labelled R) and a left rule for elimination (labelled L). We frequently apply the algorithmic reading of the judgements, where meta-level terms to the left of $\Rightarrow$ are inputs (i.e., context Γ and goal type A) and terms to the right of $\Rightarrow$ are outputs (i.e., the synthesised term t and the usage context Δ). The synthesis calculus is non-deterministic, i.e., for any Γ and A there may be many possible t and Δ such that $\Gamma \vdash A \Rightarrow^+ t \mid \Delta$.

4.2.1 *Core Synthesis Rules*

4.2.1.1 *Top-level*

We begin with the TopLevel rule, which is the entry-point to synthesis, for a judgment form with a type scheme goal instead of a type:

$$\frac{\overline{\alpha : \kappa}; \emptyset \vdash A \Rightarrow t \mid \emptyset}{\emptyset; \emptyset \vdash \forall \overline{\alpha : \kappa}.A \Rightarrow t \mid \emptyset} \quad \textsc{TopLevel}$$

This rule takes the universally quantified type variables $\overline{\alpha : \kappa}$ from the type scheme and adds them to the type variable context Σ; type variables are only equal to themselves.

4.2.1.2 *Variables*

For any goal type A, if there is a variable in the context matching this type then it can be synthesised for the goal, given by the terminal rule:

$$\frac{\Sigma \vdash A : \text{Type}}{\Sigma; \Gamma, x :_r A \vdash A \Rightarrow x \mid 0 \cdot \Gamma, x :_1 A} \quad \textsc{Var}$$

Said another way, to synthesise the use of a variable x, we require that x be present in the input context Γ. The output context here then explains that only variable x is used: it consists of the entirety of the input context Γ scaled by grade 0 (using Definition 2.3.6), extended with $x :_1 A$, i.e. a single usage of x as denoted by the 1 element of the semiring. Maintaining this zeroed Γ in the output context simplifies subsequent rules by avoiding excessive context membership checks.

The Var rule permits the synthesis of terms which may not be well-resourced, e.g., if $r = 0$, the rule still synthesises a use of x. This is locally ill-resourced, but is acceptable at the global level as we check that an assumption has been used correctly in the rule where the assumption is bound. This leads us to consider some branches of synthesis that are guaranteed to fail: at the point of synthesising a usage of a variable in the additive scheme, isolated from information about how else the variable is used, there is no way of knowing if such a usage will be permissible in the final synthesised program. However, it also reduces the amount of intermediate theorems that need solving, which can significantly effect performance as shown in Chapter 3, especially since the variable rule is applied very frequently.

4.2.1.3 *Functions*

Synthesis of programs from function types is handled by the $\to_{\mathrm{R}}$ and $\to_{\mathrm{L}}$ rules, which synthesise abstraction and application terms, respectively. An abstraction is synthesised like so:

$$\frac{\Gamma, x :_q A \vdash B \Rightarrow t \mid \Delta, x :_r A \qquad r \sqsubseteq q}{\Gamma \vdash A^q \to B \Rightarrow \lambda x.t \mid \Delta} \;\to_{\mathrm{R}}$$

Reading bottom up, to synthesise a term of type $A^q \to B$ in context Γ we first extend the context with a fresh variable assumption $x :_q A$ and synthesise a term of type B that will ultimately become the body of the function. The type $A^q \to B$ conveys that A must be used according to q in our term for B. The fresh variable x is passed to the premise of the rule using the grade of the binder: q. The variable x must then be used to synthesise a term t with q usage. In the premise, after synthesising t we obtain an output context $\Delta, x :_r A$. The Var and $\mathrm{C_R}$ rules (i.e. terminal rules) ensure that x is present in this context, even if it was not used in the synthesis of t (e.g., $r = 0$). The rule ensures the usage of bound term (r) in t does not violate the input grade q via the requirement that $r \sqsubseteq q$ i.e. that q *approximates* r. If met, Δ becomes the output context of the rule's conclusion.

The counterpart to abstraction synthesises an application from the occurrence of a function in the context (a left rule):

$$\frac{\begin{array}{l}\Gamma, x_1 :_{r_1} A^q \to B, x_2 :_{r_1} B \vdash C \Rightarrow t_1 \mid \Delta_1, x_1 :_{s_1} A^q \to B, x_2 :_{s_2} B \\ \Gamma, x_1 :_{r_1} A^q \to B \vdash A \Rightarrow t_2 \mid \Delta_2, x_1 :_{s_3} A^q \to B\end{array}}{\Gamma, x_1 :_{r_1} A^q \to B \vdash C \Rightarrow [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + s_2 \cdot q \cdot \Delta_2), x_1 :_{s_2 + s_1 + (s_2 \cdot q \cdot s_3)} A^q \to B} \;\to_{\mathrm{L}}$$

Reading bottom up again, the input context contains an assumption with a function type $x_1 :_{r_1} A^q \to B$. We may attempt to use this assumption in the synthesis of a term with the goal type C, by applying some argument to it. We do this by synthesising the argument from the input type of the function A, and then binding the result of this application as an assumption of type B in the synthesis of C. This is decomposed into two steps corresponding to the two premises (though in the implementation the first premise is considered first):

1. The first premise synthesises a term t_1 from the goal type C under the assumption that the function x_1 has been applied and its result is bound to x_2. This placeholder assumption is bound with the same grade as x_1.

2. The second premise synthesises an argument t_2 of type A for the function x_1. In the implementation, this synthesis step occurs only after a term t_1 is found for the goal C as a heuristic to avoid possibly unnecessary work if no term can be synthesised for C.

In the conclusion of the rule, a term is synthesised which substitutes in t_1 the result placeholder variable x_2 for the application $x_1\, t_2$.

The first premise yields an output context $\Delta_1, x_1 :_{s_1} A^q \to B, x_2 :_{s_2} B$. The output context of the conclusion is obtained by taking the context addition of Δ_1 and $s_2 \cdot q \cdot \Delta_2$. The output context Δ_2 is first scaled by q since t_2 is used according to q when applied to x_1 (as per the type of x_1). We then scale this again by s_2 which represents the usage of the entire application $x_1\, t_2$ inside t_1.

The output grade of x_1 follows a similar pattern since this rule permits the re-use of x_1 inside both premises of the application (which differs from our treatment of synthesis in a linear setting). As x_1's input grade r_1 may permit multiple uses both inside the synthesis of the application argument t_2 and in t_1 itself, the total usage of t_1 across both premises must be calculated.

In the first premise x_1 is used according to s_1, and in the second according to s_3. As with Δ_2, we take the semiring multiplication of s_3 and q and then multiply this by s_2 to yield the final usage of x_1 in t_2. We then add this to $s_2 + s_1$ to yield the total usage of x_1 in t_1.

The reasoning behind this grade calculation is illustrated in the soundness proof for $\to_{\mathrm{L}}$. As in the additive synthesis calculus of Chapter 3, a synthesis judgment is sound if we can type the synthesised term against the goal type using the assumptions of the output context. We give the full details of soundness in Section 4.2.2 but use the $\to_{\mathrm{L}}$ to aid in understanding how the output grades are used when typing the synthesised term.

The total usage of x_1 in the application $x_1\, t_2$ is thus given by $1 + (q \cdot s_3)$. Since we use x_1 once on the left hand side of the application and according to s_3 inside t_2, while application scales the argument grade by the function type's grade q. This means we can construct the following typing derivation:

$$\dfrac{\dfrac{\Sigma \vdash A^q \to B : \mathrm{Type}}{\Sigma; x_1 :_1 A^q \to B \vdash x_1 : A^q \to B}\,\textsc{Var} \qquad \Sigma; \Delta_2, x_1 :_{s_3} A^q \to B \vdash t_2 : A}{\Sigma; q \cdot \Delta_2, x_1 :_{1+(q \cdot s_3)} A^q \to B \vdash x_1\, t_2 : B}\,\textsc{App}$$

where

$$\Sigma; \Delta_2, x_1 :_{s_3} A^q \to B \vdash t_2 : A$$

is an inductive hypothesis obtained from the second premise of the $\to_{\mathrm{L}}$ rule. The rest of x's output grade is then left to type its usage in t_1.

4.2.1.4 *Using polymorphic definitions*

Programs can be synthesised from a polymorphic type scheme (the previously shown TopLevel rule), treating universally-quantified type

variables at the top-level of our goal type as logical atoms which cannot be unified with and are only equal to themselves. The DEF rule synthesises a *use* of a top-level polymorphic function via instantiation:

$$\frac{(x : \forall \overline{\alpha : \kappa}.A') \in D \qquad \Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')}{\Sigma; \Gamma \vdash A \Rightarrow x \mid 0 \cdot \Gamma} \ \text{DEF}$$

For example, in the following we have a polymorphic function `flip` that we want to use to synthesise a monomorphic function:

```
flip : ∀ c d  . (c, d) %1 → (d, c)
flip (x, y) = (y, x)
f : (Int, Int) %1 → (Int, Int)
f x = ? -- synthesis to flip x trivially
```

To synthesise the term `flip x`, the type scheme of `flip` is instantiated via DEF with $\emptyset \vdash (Int \otimes Int)^1 \to (Int \otimes Int) = \text{inst}(\forall c : \text{Type}, d : \text{Type}.(c \otimes d)^1 \to (d \otimes c))$.

4.2.1.5 *Graded modalities*

Graded modalities are introduced and eliminated explicitly through the $\Box_R$ and $\Box_L$ rules, respectively. In the $\Box_R$ rule, we synthesise a promotion $[t]$ for some graded modal goal type $\Box_r A$:

$$\frac{\Gamma \vdash A \Rightarrow t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow [t] \mid r \cdot \Delta} \ \Box_R$$

In the premise, we synthesise from A, yielding the sub-term t and an output context Δ. In the conclusion, Δ is scaled by the grade of the goal type r: as $[t]$ must use t as r requires.

Grade elimination (*unboxing*) takes place via pattern matching in a **case** statement:

$$\frac{\begin{array}{l}\Gamma, y :_{r \cdot q} A, x :_r \Box_q A \vdash B \Rightarrow t \mid \Delta, y :_{s_1} A, x :_{s_2} \Box_q A \\ \exists s_3.\, s_1 \sqsubseteq s_3 \cdot q \sqsubseteq r \cdot q\end{array}}{\Gamma, x :_r \Box_q A \vdash B \Rightarrow \textbf{case } x \textbf{ of } [y] \to t \mid \Delta, x :_{s_3 + s_2} \Box_q A} \ \Box_L$$

To eliminate the assumption x of graded modal type $\Box_q A$, we bind a fresh assumption in the synthesis of the premise: $y :_{r \cdot q} A$. This assumption is graded with $r \cdot q$: the grade from the assumption's type multiplied by the grade of the assumption itself. As with previous elimination rules, x is rebound in the rule's premise. A term t is then synthesised resulting in the output context $\Delta, y :_{s_1} A, x :_{s_2} \Box_q A$, where s_1 and s_2 describe how y and x were used in t. The second premise ensures that the usage of y is well-resourced. The grade s_3 represents how much the usage of y inside t contributes to the overall usage of x. The constraint $s_1 \sqsubseteq s_3 \cdot q$ conveys the fact that q uses of y constitutes a single use of x, with the constraint $s_3 \cdot q \sqsubseteq r \cdot q$ ensuring that the overall

usage does not exceed the binding grade. For the output context of the conclusion, we simply remove the bound y from Δ and add x, with the grade $s_2 + s_3$: representing the total usage of x in t.

4.2.1.6 *Data types*

The synthesis of introduction forms for data types is by the ConRule:

$$\frac{\begin{array}{l}(C : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\vec{A'}) \in D \\ \Sigma \vdash {B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A} = \mathsf{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\vec{A'}) \\ \Sigma; \Gamma \vdash B_i \Rightarrow t_i \mid \Delta_i\end{array}}{\Sigma; \Gamma \vdash K\,\vec{A} \Rightarrow C\,t_1\,...\,t_n \mid 0 \cdot \Gamma + (q_1 \cdot \Delta_1) + ... + (q_n \cdot \Delta_n)}\ \mathrm{C_R}$$

where D is the set of data constructors in global scope, e.g., coming from ADT definitions, including here products, unit, and coproducts with $(,) : A^1 \to B^1 \to A \otimes B$, $Unit : \mathsf{Unit}$, $\mathsf{inl} : A^1 \to A \oplus B$, and $\mathsf{inr} : B^1 \to A \oplus B$.

For a goal type $K\,\vec{A}$ where K is a data type with zero or more type arguments (denoted by the vector $\vec{A}$), then a constructor term $C\,t_1\,..\,t_n$ for $K\,\vec{A}$ is synthesised. The type scheme of the constructor in D is first instantiated (similar to Def rule), yielding a type ${B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A}$. A sub-term is then synthesised for each of the constructor's arguments t_i in the third premise (which is repeated for each instantiated argument type B_i), yielding output contexts Δ_i. The output context for the rule's conclusion is obtained by performing a context addition across all the output contexts generated from the premises, where each context Δ_i is scaled by the corresponding grade q_i from the data constructor in D capturing the fact that each argument t_i is used according to q_i.

Dual to the above, constructor elimination synthesises **case** statements with branches pattern matching on each data constructor of the target data type $K\,\vec{A}$, with various associated constraints on grades which require some explanation:

$$\frac{\begin{array}{l}(C_i : \forall \overline{\alpha : \kappa}.{B'_1}^{q^i_1} \to ... \to {B'_n}^{q^i_n} \to K\,\vec{A'}) \in D \qquad \Sigma \vdash K\,\vec{A} : \mathrm{Type} \\ \Sigma \vdash {B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A} = \mathsf{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\vec{A'}) \\ \Sigma; \Gamma, x :_r K\,\vec{A}, y^i_1 :_{r \cdot q^i_1} B_1, ..., y^i_n :_{r \cdot q^i_1} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\,\vec{A}, y^i_1 :_{s^i_1} B_1, ..., y^i_n :_{s^i_n} B_n \\ \exists {s'}^i_j . s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j \qquad s_i = {s'}^i_1 \sqcup ... \sqcup {s'}^i_n \qquad |K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup ... \sqcup s_m\end{array}}{\Sigma; \Gamma, x :_r K\,\vec{A} \vdash B \Rightarrow \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ y^i_1 ... y^i_n \mapsto t_i} \mid (\Delta_1 \sqcup ... \sqcup \Delta_m), x :_{(r_1 \sqcup ... \sqcup r_m) + (s_1 \sqcup ... \sqcup s_m)} K\,\vec{A}}\ \mathrm{C_L}$$

where $1 \leq i \leq m$ is used to index the data constructors of which there are m (i.e., $m = |K\,\vec{A}|$) and $1 \leq j \leq n$ is used to index the arguments of the i^{th} data constructor. For brevity, the rule focuses n-ary data constructors where $n > 0$.

As with constructor introduction, the relevant data constructors are retrieved from the global scope D in the first premise. A data constructor type is a function type from the constructor's arguments

$B_1 \ldots B_n$ to a type constructor applied to zero or more type parameters $K\,\vec{A}$. However, in the case of nullary data constructors (e.g., for the unit type), the data constructor type is simply the type constructor's type with no arguments. For each data constructor C_i, we synthesise a term t_i from the result type of the data constructor's type in D, binding the data constructor's argument types as fresh assumptions to be used in the synthesis of t_i.

To synthesise the body for each branch i, the arguments of the data constructor are bound to fresh variables in the premise, with the grades from their respective argument types in D multiplied by the r. This follows the pattern typing rule for constructors; a pattern match under some grade r must bind assumptions that have the capability to be used according to r.

The assumption being eliminated $x :_r K\,\vec{A}$ is also included in the premise's context (as in $\to_{\mathrm{L}}$) as we may perform additional eliminations on the current assumption subsequently if the grade r allows us. If successful, this will yield both a term t_i and an output context for the pattern match branch. The output context can be broken down into three parts:

1. Δ_i contains any assumptions from Γ were used to construct t_i
2. $x :_{r_i} K\,A$ describes how the assumption x was used
3. $y^i_1 :_{s^i_1} B_1, \ldots, y^i_n :_{s^i_n} B_n$ describes how each assumption y^i_j bound in the pattern match was used in t_i.

This leaves the question of how we calculate the final grade to attribute to x in the output context of the rule's conclusion. For each bound assumption, we generate a fresh grade variable s'^i_j which represents how that variable was used in t_i after factoring out the multiplication by q^i_j. This is done via the constraint in the third premise that $\exists s'^i_j.\, s^i_j \sqsubseteq s'^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j$. The join of each s'^i_j (for each assumption) is then taken to form a grade variable s_i which represents the total usage of x for this branch that arises from the use of assumptions which were bound via the pattern match (i.e. not usage that arises from reusing x explicitly inside t_i). For the output context of the conclusion, we then take the join of output context from the constructors used. This is extended with the original x assumption with the output grade consisting of the join of each r_i (the usages of x directly in each branch) plus the join of each s_i (the usages of the assumptions that were bound from matching on a constructor of x).

Example 4.2.1 (Example of **case** synthesis)**.** Consider two possible synthesis results:

$$x :_r \mathsf{Unit} \oplus A, y :_s A, z :_{r \cdot q_1} A \vdash A \Rightarrow^+ z \mid x :_0 \mathsf{Unit} \oplus A, y :_0 A, z :_1 A \tag{4.1}$$

$$x :_r \mathsf{Unit} \oplus A, y :_s A \vdash A \Rightarrow^+ y \mid x :_0 \mathsf{Unit} \oplus A, y :_1 A \tag{4.2}$$

We will plug these into the rule for generating case expressions as follows where in the following instead of using the above concrete grades we have used the abstract form of the rule (the two will be linked by equations after):

$$\frac{\begin{array}{c}\mathsf{Just} : \forall \overline{\alpha : \kappa}.A'^{1} \to A' \oplus \mathsf{Unit} \in D \\ \mathsf{Nothing} : \forall \overline{\alpha : \kappa}.\mathsf{Unit}^{1} \to A' \oplus \mathsf{Unit} \in D \\ \Sigma \vdash A^{1} \to A \oplus \mathsf{Unit} = \mathsf{inst}(\forall \overline{\alpha : \kappa}.A'^{1} \to A' \oplus \mathsf{Unit}) \\ \Sigma \vdash \mathsf{Unit}^{1} \to A \oplus \mathsf{Unit} = \mathsf{inst}(\forall \overline{\alpha : \kappa}.\mathsf{Unit}^{1} \to A' \oplus \mathsf{Unit}) \\ (4.1) \quad \Sigma; x :_{r} \mathsf{Unit} \oplus A, y :_{s} A, z :_{r \cdot q_1} A \vdash A \Rightarrow z \mid x :_{0} \mathsf{Unit} \oplus A, y :_{0} A, z :_{s_1} A \\ (4.2) \quad \Sigma; x :_{r} \mathsf{Unit} \oplus A, y :_{s} A \vdash A \Rightarrow y \mid x :_{0} \mathsf{Unit} \oplus A, y :_{1} A \\ \exists s'_1.\, s_1 \sqsubseteq s'_1 \cdot q_1 \sqsubseteq r \cdot q_1 \qquad s' = s'_1\end{array}}{\Sigma; x :_{r} \mathsf{Unit} \oplus A, y :_{s} A \vdash A \Rightarrow (\mathbf{case}\ x\ \mathbf{of}\ \mathsf{Just}\, z \to z; \mathsf{Nothing} \to y) \mid x :_{(0 \sqcup 0) + s'} \mathsf{Unit} \oplus A, y :_{0 \sqcup 1} A}\ \textsc{Case}$$

Thus, to unify (4.1) and (4.2) with the rule format we have that $s_1 = 1$ and $q_1 = 1$. Applying these two equalities as rewrites to the remaining constraint, we have:

$$\exists s'_1.\, 1 \sqsubseteq s'_1 \cdot 1 \sqsubseteq r \cdot 1 \quad \Longrightarrow \quad \exists s'_1.\, 1 \sqsubseteq s'_1 \sqsubseteq r$$

These constraints can be satisfied with the natural-number intervals semiring where y has grade 0..1 and x has grade 1..1.

Deep pattern matching, over nested data constructors, is handled via inductively applying the Case rule but with a post-synthesis refactoring procedure substituting the pattern match of the inner case statement into the outer pattern match. For example, nested matching on pairs becomes a single **case** with nested pattern matching, simplifying the program:

$$\begin{array}{rl} & \mathbf{case}\ x\ \mathbf{of}\ (y_1, y_2) \to \mathbf{case}\ y_1\ \mathbf{of}\ (z_1, z_2) \to z_2 \\ \textit{(rewritten to)} \ \rightsquigarrow & \mathbf{case}\ x\ \mathbf{of}\ ((z_1, z_2), y_2) \to z_2 \end{array}$$

4.2.1.7 *Recursion*

Synthesis permits recursive definitions, as well as programs which may make use of calls to functions from a user-supplied context of function definitions in scope (using the `spec` construct). Synthesis of non-recursive function applications may take place arbitrarily, however, synthesising a recursive function definition application requires more care. To ensure that a synthesised programs terminates, we only permit synthesis of terms which are *structurally recursive*, i.e., those which apply the recursive definition to a sub-term of the function's inputs [Osera, 2015].

Synthesis rules for recursive types (μ-types) are straightforward:[2]

$$\frac{\Gamma \vdash A[\mu X.A/X] \Rightarrow t \mid \Delta}{\Gamma \vdash \mu X.A \Rightarrow t \mid \Delta}\ \mu_{\mathrm{R}} \qquad \frac{\Gamma, x :_{r} A[\mu X.A/X] \vdash B \Rightarrow t \mid \Delta}{\Gamma, x :_{r} \mu X.A \vdash B \Rightarrow t \mid \Delta}\ \mu_{\mathrm{L}}$$

2 Though μ types are equi-recursive, we make explicit the synthesis rules here which maps more closely to the implementation where iterative deepening information needs to be tracked at the points of using μ_{L} and μ_{R}.

This μ_R rule states that to synthesise a recursive data structure of type $\mu X.A$, we must be able to synthesise A with $\mu X.A$ substituted for the recursion variables X in A. For example, if we wish to synthesise a list data type `List a` with constructors `Nil` and `Cons a (List a)`, then when choosing the `Cons` constructor in the μ_R rule, the type of this constructor requires us to re-apply the μ_R rule, to synthesise the recursive part of `Cons`. Elimination of a recursive data structure may be synthesised using the μ_L rule. In this rule, we have some recursive data type $\mu X.A$ in our context which we may wish to pattern match on via the C_L rule. To do this, the assumption is bound in the premise with the type A, substituting $\mu X.A$ for the recursion variables X in A.

Recursive data structures present a challenge in the implementation. For our list data type, how do we prevent our synthesis tool from simply applying the μ_L rule, followed by the C_L rule on the `Cons` constructor ad infinitum? We resolve this issue using an *iterative deepening* approach to synthesis similar to the approach used by Myth [Osera, 2015]. Programs are synthesised with elimination (and introduction) forms of constructors restricted up to a given depth. If no program is synthesised within these bounds, then the depth limits are incremented. Combined with focusing (see Section 4.4), this provides the basis for an efficient implementation of the above rules.

$$\frac{(x : \forall \overline{\alpha : \kappa}.A') \in D \qquad \Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')}{\Sigma; \Gamma \vdash A \Rightarrow x \mid 0 \cdot \Gamma} \ \textsc{Def}$$

$$\frac{\overline{\alpha : \kappa}; \emptyset \vdash A \Rightarrow t \mid \emptyset}{\emptyset; \emptyset \vdash \forall \overline{\alpha : \kappa}.A \Rightarrow t \mid \emptyset} \ \textsc{TopLevel} \qquad \frac{\Sigma \vdash A : \text{Type}}{\Sigma; \Gamma, x :_r A \vdash A \Rightarrow x \mid 0 \cdot \Gamma, x :_1 A} \ \textsc{Var}$$

$$\frac{\Sigma; \Gamma, x :_q A \vdash B \Rightarrow t \mid \Delta, x :_r A \qquad r \sqsubseteq q}{\Sigma; \Gamma \vdash A^q \to B \Rightarrow \lambda x.t \mid \Delta} \ \to_{\text{R}}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma, x_1 :_{r_1} A^q \to B, x_2 :_{r_1} B \vdash C \Rightarrow t_1 \mid \Delta_1, x_1 :_{s_1} A^q \to B, x_2 :_{s_2} B \\ \Sigma; \Gamma, x_1 :_{r_1} A^q \to B \vdash A \Rightarrow t_2 \mid \Delta_2, x_1 :_{s_3} A^q \to B \qquad \Sigma \vdash A^q \to B : \text{Type}\end{array}}{\Sigma; \Gamma, x_1 :_{r_1} A^q \to B \vdash C \Rightarrow [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + s_2 \cdot q \cdot \Delta_2), x_1 :_{s_2 + s_1 + (s_2 \cdot q \cdot s_3)} A^q \to B} \ \to_{\text{L}}$$

$$\frac{\begin{array}{l}(C : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\,\vec{A}') \in D \\ \Sigma \vdash {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K\,\vec{A} = \text{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\,\vec{A}') \\ \Sigma; \Gamma \vdash B_i \Rightarrow t_i \mid \Delta_i\end{array}}{\Sigma; \Gamma \vdash K\,\vec{A} \Rightarrow C\,t_1 \ldots t_n \mid 0 \cdot \Gamma + (q_1 \cdot \Delta_1) + \ldots + (q_n \cdot \Delta_n)} \ \text{C}_{\text{R}}$$

$$\frac{\begin{array}{l}(C_i : \forall \overline{\alpha : \kappa}.{B'_1}^{q^i_1} \to \ldots \to {B'_n}^{q^i_n} \to K\,\vec{A}') \in D \qquad \Sigma \vdash K\,\vec{A} : \text{Type} \\ \Sigma \vdash {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K\,\vec{A} = \text{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\,\vec{A}') \\ \Sigma; \Gamma, x :_r K\,\vec{A}, y^i_1 :_{r \cdot q^i_1} B_1, \ldots, y^i_n :_{r \cdot q^i_1} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\,\vec{A}, y^i_1 :_{s^i_1} B_1, \ldots, y^i_n :_{s^i_n} B_n \\ \exists s'^i_j.\, s^i_j \sqsubseteq s'^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j \qquad s_i = s'^i_1 \sqcup \ldots \sqcup s'^i_n \qquad |K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup \ldots \sqcup s_m\end{array}}{\Sigma; \Gamma, x :_r K\,\vec{A} \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y^i_1 \ldots y^i_n \mapsto t_i} \mid (\Delta_1 \sqcup \ldots \sqcup \Delta_m), x :_{(r_1 \sqcup \ldots \sqcup r_m) + (s_1 \sqcup \ldots \sqcup s_m)} K\,\vec{A}} \ \text{C}_{\text{L}}$$

$$\frac{\Sigma; \Gamma \vdash A \Rightarrow t \mid \Delta}{\Sigma; \Gamma \vdash \Box_r A \Rightarrow [t] \mid r \cdot \Delta} \ \Box_{\text{R}}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma, y :_{r \cdot q} A, x :_r \Box_q A \vdash B \Rightarrow t \mid \Delta, y :_{s_1} A, x :_{s_2} \Box_q A \\ \exists s_3.\, s_1 \sqsubseteq s_3 \cdot q \sqsubseteq r \cdot q \qquad \Sigma \vdash \Box_q A : \text{Type}\end{array}}{\Sigma; \Gamma, x :_r \Box_q A \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ [y] \to t \mid \Delta, x :_{s_3 + s_2} \Box_q A} \ \Box_{\text{L}}$$

$$\frac{D; \Sigma; \Gamma \vdash A[\mu X.A/X] \Rightarrow t \mid\ \Delta}{D; \Sigma; \Gamma \vdash \mu X.A \Rightarrow t \mid\ \Delta}\ \mu_{\text{R}} \qquad \frac{D; \Sigma; \Gamma, x :_r A[\mu X.A/X] \vdash B \Rightarrow t \mid\ \Delta}{D; \Sigma; \Gamma, x :_r \mu X.A \vdash B \Rightarrow t \mid\ \Delta}\ \mu_{\text{L}}$$

Figure 4.5: Collected rules of the fully graded synthesis calculus

4.2.2 *Soundness of Synthesis*

The relationship between synthesis and typing is given by the central soundness result:

Theorem 4.2.1 (Soundness of synthesis)**.** Given a particular pre-ordered semiring $\mathcal{R}$ parametrising the calculi, then:

1. For all contexts Γ and Δ, types A, terms t:

$$\Sigma; \Gamma \vdash A \Rightarrow t \mid \Delta \quad \Longrightarrow \quad \Sigma; \Delta \vdash t : A$$

   i.e. t has type A under context Δ whose grades capture variable use in t.

2. At the top-level, for all type schemes $\forall \overline{\alpha : \kappa}.A$ and terms t then:

$$\emptyset; \emptyset \vdash \forall \overline{\alpha : \kappa}.A \Rightarrow t \mid \emptyset \quad \Longrightarrow \quad \emptyset; \emptyset \vdash t : \forall \overline{\alpha : \kappa}.A$$

The first part of soundness takes the same form as soundness for the additive calculus of Chapter 3 (see Section 3.4.8), i.e. it does not on its own guarantee well-resourcedness. The grades in the output context Δ may not be approximated by the grades in the input context Γ. Instead, an individual synthesis judgement forms part of a larger derivation which is well-resourced. As in Chapter 3 we apply pruning judiciously, only requiring that variable use is well-resourced at the point of synthesising binders. Therefore synthesised closed terms are always well-resourced (second part of the soundness theorem).

For open terms, the implementation checks that from a user-given top-level goal A for which $\Gamma \vdash A \Rightarrow t \mid \Delta$ is derivable then t is only provided as a valid (well-typed and well-resourced) result if $\Delta \sqsubseteq \Gamma$.

Section B.3.1 of Appendix B provides the soundness proof, which in part resembles a translation from sequent calculus to natural deduction, but also with the management of grades between synthesis and type checking.

4.3 POST-SYNTHESIS REFACTORING

In Section 3.6.1, we considered how our synthesised terms could be refactored into a more idiomatic programming style. Those same refactoring transformations also apply to our calculus here, along with the additional treatment relating to case statements.

Recall that the C_L binds a data constructor's patterns as a series of variables. Synthesising a pattern match over a nested data structure therefore yields a term such as:

```
case x of
  C₁ y →
    case y of
      D₁ z → ...
      D₂ z → ...
  C₂ y →
    case y of
      D₁ z → ...
      D₂ z → ...
```

which would be rather unnatural for a programmer to write. Nested case statements are therefore folded together to yield a single case statement which pattern matches over all combination of patterns from each statement. The above cases are then transformed into the much more compact and readable single case:

```
case x of
  C₁ (D₁ z) → ...
  C₁ (D₂ z) → ...
  C₂ (D₁ z) → ...
  C₂ (D₂ z) → ...
```

Furthermore, pattern matches over a function's arguments in the form of case statements are refactored such that a new function equation is created for each unique combination of pattern match. In this way, a refactored program should only contain case statements that arise from pattern matching over the result of an application.

```
neg : Bool %1 → Bool %1
neg x = case x of
          True → False;
          False → True
```

is refactored into:

```
neg : Bool %1 → Bool %1
neg True = False;
neg False = True
```

The exception to this is where the scrutinee of a case statement is re-used inside one of the case branches, in which case refactoring would cause us to throw away the binding of the scrutinee's name and so it cannot be folded into the head pattern match, for example:

```
last : ∀ { a : Type } . (List a) %0..∞ → Maybe a
spec
    last Nil = Nothing;
    last (Cons 1 Nil) = Just 1;
    last (Cons 1 (Cons 2 Nil)) = Just 2;
    last %0..∞
last Nil = Nothing;
last (Cons y z) =
    (case z of
      Nil → Just y;
      Cons u v → last z)
```

A final minor refactoring procedure is to refactor a variable pattern into a wildcard pattern, in the case that the bound variable is not used inside the body of the case branch:

```
throwAway : ∀ { a : Type } . a %0..∞ → ()
throwAway x = ()
```

is refactored into:

```
throwAway : ∀ { a : Type } . a %0..∞ → ()
throwAway _ = ()
```

For such a scenario to occur a pattern must typed with a grade that is approximatable by 0.

4.4 FOCUSING

As in Chapter 3, the synthesis rules as we have presented them thus far would be too non-deterministic to form the basis of an implementation through direct translation into code. Again, we apply the technique of focusing [Andreoli, 1992] to our calculus, yielding a *focused* synthesis calculus which imposes an ordering on when rules may be applied at each stage of synthesis.

We have already outlined the principles behind focusing in Section 3.5, and thus opt not to repeat ourselves here. Instead, we merely present the focused form of the fully graded synthesis calculus in Figures 4.6, 4.7, 4.8, 4.9, 4.10, and 4.11. and state that focusing is sound (Lemma 4.4.1). The proof of soundness of focusing can be found in Section B.3.2 of Appendix B.

Lemma 4.4.1 (Soundness of focusing for graded-base synthesis)**.** For all contexts Γ, Ω and types A:

1. RIGHT ASYNC :	$D;\Sigma;\Gamma;\Omega \vdash A \Uparrow \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma,\Omega \vdash A \Rightarrow t \mid \Delta$
2. LEFT ASYNC :	$D;\Sigma;\Gamma;\Omega \Uparrow \vdash B \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma,\Omega \vdash B \Rightarrow t \mid \Delta$
3. RIGHT SYNC :	$D;\Sigma;\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma \vdash A \Rightarrow t \mid \Delta$
4. LEFT SYNC :	$D;\Sigma;\Gamma;x :_r A \Downarrow \vdash B \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma,x :_r A \vdash B \Rightarrow t \mid \Delta$
5. FOCUS RIGHT :	$D;\Sigma;\Gamma;\varnothing \Uparrow \vdash B \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma \vdash B \Rightarrow t \mid \Delta$
6. FOCUS LEFT :	$D;\Sigma;\Gamma,x :_r A;\varnothing \Uparrow \vdash B \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma,x :_r A \vdash B \Rightarrow t \mid \Delta$

i.e. t has type A under context Δ, which contains variables with grades reflecting their use in t.

$$\frac{D;\Sigma;\Gamma;\Omega,x :_q A \vdash B \Uparrow \Rightarrow t \mid \Delta, x :_r A \qquad r \sqsubseteq q}{D;\Sigma;\Gamma;\Omega \vdash A^q \to B \Uparrow \Rightarrow \lambda x.t \mid \Delta} \to_R$$

$$\frac{D;\overline{\alpha : \kappa};\varnothing;\varnothing \vdash A \Uparrow \Rightarrow t \mid \varnothing}{D;\varnothing;\varnothing;\varnothing \vdash \forall \overline{\alpha : \kappa}.A \Uparrow \Rightarrow t \mid \varnothing} \text{ TOPLEVEL}$$

$$\frac{D;\Sigma;\Gamma;\Omega \Uparrow \vdash B \Rightarrow t \mid \Delta \qquad B \text{ not right async}}{D;\Sigma;\Gamma;\Omega \vdash B \Uparrow \Rightarrow t \mid \Delta} \Uparrow_R$$

Figure 4.6: Right Async rules of the focused fully graded synthesis calculus

$$\frac{\begin{array}{l}(C_i : \forall\overline{\alpha : \kappa}.{B'_1}^{q_1^i} \to ... \to {B'_n}^{q_n^i} \to K\,\vec{A'}) \in D \qquad \Sigma \vdash K\,\vec{A} : \text{Type} \\ \Sigma \vdash {B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A} = \text{inst}(\forall\overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\vec{A'}) \\ D; \Sigma; \Gamma; \Omega, x :_r K\,\vec{A}, y_1^i :_{r \cdot q_1^i} B_1, ..., y_n^i :_{r \cdot q_1^i} B_n \Uparrow \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\,\vec{A}, y_1^i :_{s_1^i} B_1, ..., y_n^i :_{s_n^i} B_n \\ \exists {s'}_j^i.\, s_j^i \sqsubseteq {s'}_j^i \cdot q_j^i \sqsubseteq r \cdot q_j^i \qquad s_i = {s'}_1^i \sqcup ... \sqcup {s'}_n^i \qquad |K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup ... \sqcup s_m\end{array}}{D; \Sigma; \Gamma; \Omega, x :_r K\,\vec{A} \Uparrow \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y_1^i ... y_n^i \mapsto t_i} \mid (\Delta_1 \sqcup ... \sqcup \Delta_m), x :_{(r_1 \sqcup ... \sqcup r_m) + (s_1 \sqcup ... \sqcup s_m)} K\,\vec{A}}\ \text{C}_\text{L}$$

$$\frac{D; \Sigma; \Gamma; \Omega, x :_r A[\mu X.A/X] \Uparrow \vdash B \Rightarrow t \mid\ \Delta}{D; \Sigma; \Gamma; \Omega, x :_r \mu X.A \Uparrow \vdash B \Rightarrow t \mid\ \Delta}\ \mu_\text{L}$$

$$\frac{\begin{array}{l}D; \Sigma; \Gamma; \Omega, y :_{r \cdot q} A, x :_r \Box_q A \Uparrow \vdash B \Rightarrow t \mid \Delta, y :_{s_1} A, x :_{s_2} \Box_q A \\ \exists s_3.\, s_1 \sqsubseteq s_3 \cdot q \sqsubseteq r \cdot q \qquad \Sigma \vdash \Box_q A : \text{Type}\end{array}}{D; \Sigma; \Gamma; \Omega, x :_r \Box_q A \Uparrow \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ [y] \to t \mid \Delta, x :_{s_3 + s_2} \Box_q A}\ \Box_\text{L}$$

$$\frac{D; \Sigma; \Gamma, x :_r A; \Omega \Uparrow \vdash B \Rightarrow t \mid \Delta \qquad \text{A not left async}}{D; \Sigma; \Gamma; \Omega, x :_r A \Uparrow \vdash B \Rightarrow t \mid \Delta}\ \Uparrow_\text{L}$$

Figure 4.7: Left Async rules of the focused fully graded synthesis calculus

$$\frac{D; \Sigma\Gamma; \varnothing \vdash B \Downarrow \Rightarrow t \mid\ \Delta \qquad \text{B not atomic}}{D; \Sigma; \Gamma; \varnothing \Uparrow \vdash B \Rightarrow t \mid \Delta}\ \text{Foc}_\text{R}$$

$$\frac{D; \Sigma; \Gamma; x :_r A \Downarrow \vdash B \Rightarrow t \mid \Delta}{D; \Sigma; \Gamma, x :_r A; \varnothing \Uparrow \vdash B \Rightarrow t \mid \Delta}\ \text{Foc}_\text{L}$$

Figure 4.8: Focus rules of the focused fully graded synthesis calculus

$$\frac{\begin{array}{c}(C : \forall\overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\vec{A'}) \in D \\ \Sigma \vdash {B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A} = \text{inst}(\forall\overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\vec{A'}) \\ D; \Sigma; \Gamma; \varnothing \vdash B_i \Downarrow \Rightarrow t_i \mid\ \Delta_i\end{array}}{D; \Sigma; \Gamma; \varnothing \vdash K\,\vec{A} \Downarrow \Rightarrow C\,t_1 ... t_n \mid\ \Delta_1 + ... + \Delta_n}\ \text{Con}_\text{R}$$

$$\frac{D; \Sigma; \Gamma; \varnothing \vdash A \Downarrow \Rightarrow t \mid\ \Delta}{D; \Sigma; \Gamma; \varnothing \vdash \Box_r A \Downarrow \Rightarrow [t] \mid\ r \cdot \Delta}\ \Box_\text{R}$$

$$\frac{D; \Sigma; \Gamma; \varnothing \vdash A[\mu X.A/X] \Downarrow \Rightarrow t \mid\ \Delta}{D; \Sigma; \Gamma; \varnothing \vdash \mu X.A \Downarrow \Rightarrow t \mid\ \Delta}\ \mu_\text{R}$$

$$\frac{D; \Sigma; \Gamma; \varnothing \vdash A \Uparrow \Rightarrow t \mid \Delta}{D; \Sigma; \Gamma; \varnothing \vdash A \Downarrow t \mid\ \Delta}\ \Downarrow_\text{R}$$

Figure 4.9: Right Sync rules of the focused fully graded synthesis calculus

$$\frac{\begin{array}{c} D;\Sigma;\Gamma;x_1 :_{r_1} A^q \to B, x_2 :_{r_1} B \Downarrow \vdash C \Rightarrow t_1 \mid \Delta_1, x_1 :_{s_1} A^q \to B, x_2 :_{s_2} B \\ D;\Sigma;\Gamma;x_1 :_{r_1} A^q \to B \Downarrow \vdash A \Rightarrow t_2 \mid \Delta_2, x_1 :_{s_3} A^q \to B \qquad \Sigma \vdash A^q \to B : \text{Type} \end{array}}{D;\Sigma;\Gamma;x_1 :_{r_1} A^q \to B \Downarrow \vdash C \Rightarrow [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + s_2 \cdot q \cdot \Delta_2), x_1 :_{s_2+s_1+(s_2 \cdot q \cdot s_3)} A^q \multimap B} \to_{\mathrm{L}}$$

$$\frac{D;\Sigma;\Gamma;x :_r A \Uparrow \vdash B \Rightarrow t \mid \Delta \qquad \text{A not atomic and not left sync}}{D;\Sigma;\Gamma;x :_r A \Downarrow \vdash B \Rightarrow t \mid \Delta} \Downarrow_{\mathrm{L}}$$

Figure 4.10: Left Sync rules of the focused fully graded synthesis calculus

$$\frac{\Sigma \vdash A : \text{Type}}{D;\Sigma;\Gamma;x :_r A \Downarrow \vdash A \Rightarrow x \mid 0 \cdot \Gamma, x :_1 A} \textsc{Var}$$

$$\frac{\Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')}{D, x : \forall \overline{\alpha : \kappa}.A';\Sigma;\Gamma;\emptyset \Downarrow \vdash A \Rightarrow x \mid 0 \cdot \Gamma} \textsc{Def}$$

Figure 4.11: Var and Def rules of the focused fully graded synthesis calculus

4.5 EVALUATING THE SYNTHESIS CALCULUS

In evaluating our fully graded synthesis approach and tool, we made the following hypotheses:

H1. (**Expressivity; less consultation**) The use of grades in synthesis results in a synthesised program that is more likely to have the behaviour desired by the user; the user needs to request fewer alternate synthesised results (*retries*) and thus is consulted less in order to arrive at the desired program.

H2. (**Expressivity; fewer examples**) Grade-and-type directed synthesis requires fewer input-output examples to arrive at the desired program compare with a purely type-driven approach.

H3. (**Performance; more pruning**) The ability to prune resource-violating candidate programs from the search tree leads to a synthesised program being found more quickly when synthesised from a graded type compared with the same type but without grades (purely type-driven approach).

4.5.1 *Methodology*

To evaluate our approach, we collected a suite of benchmarks comprising graded type signatures for common transformations on data structures such as lists, streams, booleans, option ('maybe') types, unary natural numbers, and binary trees. We draw many of these from the benchmark suite of the MYTH synthesis tool [Osera and Zdancewic,

2015]. Benchmarks are divided into classes based on the main data type, with an additional category of miscellaneous programs. The type schemes for the full suite of benchmarks can be found in Table 4.2 while 4.1 lists the data types used by the benchmarking problems. The complete synthesised programs, including examples used and synthesis contexts can be found in Section A.2 of Appendix A.

To compare, in various ways, our grade-and-type-directed synthesis to traditional type-directed synthesis, each benchmark signature is also "de-graded" by replacing all grades in the goal with `Any` which is the only element of the singleton `Cartesian` semiring in Granule. When synthesising in this semiring, we can forgo discharging grade constraints in the SMT solver entirely. Thus, synthesis for Cartesian grades degenerates to type-directed synthesis following our rules.

To assess hypothesis 1 (grade-and-type directed leads to less consultation / more likely to synthesise the intended program) we perform grade-and-type directed synthesis on each benchmark problem and type-directed synthesis on the corresponding de-graded version. For the de-graded versions, we record the number of retries N needed to arrive at a well-resourced answer by type checking the output programs against the original graded type signature, retrying if the program is not well-typed (essentially, not well-resourced). This provides a means to check whether a program may be as intended without requiring user input. In each case we also compared whether the resulting programs from synthesis via graded-and-type directed vs. type-directed with retries (on non-well-resourced outputs) were equivalent.

To assess hypothesis 2 (graded-and-type directed requires fewer examples than type-directed), we run the de-graded (Cartesian) synthesis with the smallest set of examples which leads to the model program being synthesised (without any retries). To compare across approaches to the state-of-the-art type-directed approach, we also run a separate set of experiments comparing the minimal number of examples required to synthesise in Granule (with grades) vs. Myth.

To assess hypothesis 3 (grade-and-type-directed faster than type-directed) we compare performance in the graded setting to the non-graded Cartesian setting. Comparing our tool for speed against another type-directed (but not graded-directed) synthesis tool is likely to be largely uninformative due to differences in implementation approach obscuring any meaningful comparison. Thus, we instead compare timings for the graded approach and de-graded approach within Granule. We also record the number of search paths taken (over all retries) to assess the level of pruning in graded vs non-graded.

We ran our synthesis tool on each benchmark for both the graded type and the de-graded Cartesian case, computing the mean after 10 trials for timing data. Benchmarking was carried out using version

4.12.1 of Z3 on an M1 MacBook Air with 16 GB of RAM. A timeout limit of 10 seconds was set for synthesis.

4.5.2 *Results and Analysis*

Table 4.3 records the results comparing grade-and-type synthesis vs. the Cartesian (de-graded) type-directed synthesis. The left column gives the benchmark name, number of top-level definitions in scope that can be used as components (size of the synthesis context) labelled CTXT, and the minimum number of examples needed (#/Exs) to synthesise the Graded and Cartesian programs. In the Cartesian setting, where grade information is not available, if we forgo type checking a candidate program against the original graded type then additional input-output examples are required to provide a strong enough specification such that the correct program is synthesised (see H3). The number of additional examples is given in parentheses for those benchmarks which required these additional examples to synthesise a program in the Cartesian setting.

Each subsequent results column records: whether a program was synthesised successfully ✓ or not × (due to timeout or no solution found), the mean synthesis time (μT) or if timeout occurred, and the number branching paths (Paths) explored in the search space.

The first results column (Graded) contains the results for graded synthesis. The second results column (Cartesian + Graded type-check) contains the results for synthesising a program in the Cartesian (de-graded) setting, using the same examples set as the Graded column, and recording the number of retries (consultations of the type-checker) N needed to reach a well-resourced program. In all cases, this resulting program in the Cartesian column was equivalent to that generated by the graded synthesis, none of which needed any retries (i.e., implicitly $N = 0$ for graded synthesis). H1 is confirmed by the fact that N is greater than 0 in 29 out of 46 benchmarks (60%), i.e., the Cartesian case does not synthesise the correct program on the first attempt and needs multiple retries to reach a well-resource program, with a mean of 19.60 retries and a median of 4 retries.

For each row, we highlight the column which synthesised a result the fastest in yellow. The results show that in 17 of the 46 benchmarks (37%) the graded approach out-performed non-graded synthesis. This contradicts hypothesis 3 somewhat: whilst type-directed synthesis often requires multiple retries (versus no retries) it still outperforms the graded equivalent. This appears to be due to the cost of our SMT solving procedure which must first compile a first-order theorem on grades into the SMT-lib file format, start up Z3, and then run the solver. Considerable amounts of system overhead are incurred in this procedure. A more efficient implementation calling Z3 directly

(e.g., via a dynamic library call) may give more favourable results here. However, H3 is still somewhat supported: the cases in which the graded does outperform the Cartesian are those which involve considerable complexity in their use of grades, such as `stutter`, `inc`, and `bind` for lists, as well as `sum` for both lists and trees. In each of these cases, the Cartesian column is significantly slower, even timing out for `stutter`; this shows the power of the graded approach. Furthermore, we highlight the column with the smallest number of synthesis paths explored in blue, observing that the number of paths in the graded case is always the same or less than those in the Cartesian+graded type check case (apart from Tree stutter).

Interestingly the paths explored are sometimes the same because we use backtracking search in the Cartesian+Graded type check case where, if an output program fails to type check against the graded type signature, the search backtracks rather than starting again.

Confirming H2, we find that for the non-graded setting without graded type checking, further examples are required to synthesise the same program as the graded in 20 out of 46 (43%) cases. In these cases, an average of 1.25 additional examples was required.

Data Type		Type Scheme
List	Cons	$\forall a.a^1 \to \text{List } a^1 \to \text{List } a$
	Nil	$\forall a.\text{List } a$
Stream	Next	$\forall a.a^1 \to \text{Stream } a^1 \to \text{Stream } a$
Bool	True	Bool
	False	Bool
Maybe	Just	$\forall a.a^1 \to \text{Maybe } a$
	Nothing	$\forall a.\text{Maybe } a$
N	S	$\text{N}^1 \to \text{N}$
	Z	N
Tree	Node	$\forall a.\text{Tree } a^1 \to a^1 \to \text{Tree } a^1 \to \text{Tree } a$
	Leaf	$\forall a.\text{Tree } a$

Table 4.1: Data types used in synthesis benchmarking problems

	Problem	Type Scheme
List	append	$\forall a.\text{List } a^1 \to a^1 \to \text{List } a$
	concat	$\forall a.\text{List } a^{1..\infty} \to \text{List } a^{1..\infty} \to \text{List } a$
	empty	$\forall a.\text{Unit}^1 \to \text{List } a$
	snoc	$\forall a.\text{List}^{1..\infty} \to a^{1..\infty} \to \text{List } a$
	drop	$\forall a.\text{N}^{0..\infty} \to \text{List}^{0..\infty} \to \text{List } a$
	flatten	$\forall a.\text{List } (\text{List } a)^{0..\infty} \to \text{List } a$
	bind	$\forall a\ b.\text{List } a^{1..\infty} \to (a^{1..\infty} \to \text{List } b)^{0..\infty} \to \text{List } b$
	return	$\forall a.a^1 \to \text{List } a$
	inc	$\text{List N}^{1..\infty} \to \text{List N}$
	head	$\forall a.\text{List } a^{0..1} \to a^{0..1} \to at$
	tail	$\forall a.\text{List } a^{0..1} \to \text{List } a$
	last	$\forall a.\text{List } a^{0..\infty} \to \text{Maybe } a$
	length	$\forall a.\text{List } a \to \text{N}$
	map	$\forall a\ b.(a^{1..\infty} \to b)^{0..\infty} \to \text{List } a^{1..\infty} \to \text{List } b$
	replicate5	$\forall a.a^5 \to \text{List } a$
	replicate10	$\forall a.a^{10} \to \text{List } a$
	replicateN	$\forall a.\text{N}^{0..\infty} \to a^{0..\infty} \to \text{List}a$
	stutter	$\forall a.\text{List } (a\ [2])^{1..\infty} \to \text{List } a$
	sum	$\text{List N}^{0..\infty} \to \text{N}$
Stream	build	$\forall a.a^{1..1} \to \text{Stream } a^{1..1} \to \text{Stream } a$
	map	$\forall a\ b.\text{Stream } a^{1..\infty} \to (a^{1..\infty} \to b)^{1..\infty} \to \text{Stream } b$
	take1	$\forall a.\text{Stream } a^{0..1} \to a$
	take2	$\forall a.\text{Stream } a^{0..1} \to (a,\ a)$
	take3	$\forall a.\text{Stream } a^{0..1} \to (a,\ (a,\ a))$
Bool	neg	$\text{Bool}^1 \to \text{Bool}$
	and	$\text{Bool}^1 \to \text{Bool}^1 \to \text{Bool}$
	impl	$\text{Bool}^1 \to \text{Bool}^1 \to \text{Bool}$
	or	$\text{Bool}^1 \to \text{Bool}^1 \to \text{Bool}$
	xor	$\text{Bool}^1 \to \text{Bool}^1 \to \text{Bool}$
Maybe	bind	$\forall a\ b.\text{Maybe } a^{1..1} \to (a^{1..1} \to \text{Maybe } b)^{0..1} \to \text{Maybe } b$
	fromMaybe	$\forall a.\text{Maybe } a^{1..1} \to a^{0..1} \to a$
	return	$\forall a.a^1 \to \text{Maybe } a$
	isJust	$\forall a.\text{Maybe } a^1 \to \text{Bool}$
	isNothing	$\forall a.\text{Maybe } a^1 \to \text{Bool}$
	map	$\forall a\ b.(a^{1..1} \to b)^{0..1} \to \text{Maybe } a^{1..1} \to \text{Maybe } b$
	mplus	$\forall a\ b.\text{Maybe } a^1 \to \text{Maybe } b^1 \to \text{Maybe } (a,\ b)$
Nat	isEven	$\text{N}^{1..\infty} \to \text{Bool}$
	pred	$\text{N}^1 \to \text{N}$
	succ	$\text{N}^1 \to \text{N}$
	sum	$\text{N}^{1..\infty} \to \text{N}^{1..\infty} \to \text{N}$
Tree	map	$\forall a\ b.(a^{1..\infty} \to b)^{0..\infty} \to \text{Tree } a^{1..\infty} \to \text{Tree } b$
	stutter	$\forall a.\text{Tree } (a\ [2])^{1..\infty} \to \text{Tree } (a,\ a)$
	sum	$\text{Tree N}^{0..\infty} \to \text{N}$
Misc	compose	$\forall k : \text{Coeffect}, n\ m : k, a\ b\ c : \text{Type}.(a^m \to b)^n \to (b^n \to c)^{1:k} \to a^{n \cdot m} \to c$
	copy	$\forall a.a^2 \to (a,\ a)$
	push	$\forall k : \text{Coeffect}, c : k, a\ b : \text{Type}.(a^1 \to b)^c \to a^c \to b\ [c]$

Table 4.2: Type schemes for synthesis benchmarking results

				Graded			Cartesian + Graded type-check			
	Problem	Ctxt	#/Exs.		μT (ms)	Paths		μT (ms)	N	Paths
List	append	0	0 (+1)	✓	115.35 (5.13)	130	✓	105.24 (0.36)	8	130
	concat	1	0 (+3)	✓	1104.76 (1.60)	1354	✓	615.29 (1.43)	12	1354
	empty	0	0	✓	5.31 (0.02)	17	✓	1.20 (0.01)	0	17
	snoc	1	1	✓	2137.28 (2.14)	2204	✓	1094.03 (4.75)	8	2278
	drop	1	1	✓	1185.03 (2.53)	1634	✓	445.95 (1.71)	8	1907
	flatten	2	1	✓	1369.90 (2.60)	482	✓	527.64 (1.04)	8	482
	bind	2	0 (+2)	✓	62.20 (0.21)	129	✓	622.84 (0.95)	18	427
	return	0	0 (+1)	✓	19.71 (0.18)	49	✓	22.00 (0.08)	4	49
	inc	1	1	✓	708.23 (0.69)	879	✓	2835.53 (7.69)	24	1664
	head	0	1	✓	68.23 (0.53)	34	✓	20.78 (0.10)	4	34
	tail	0	1	✓	84.23 (0.20)	33	✓	38.59 (0.06)	8	33
	last	1	1 (+1)	✓	1298.52 (1.17)	593	✓	410.60 (6.25)	4	684
	length	1	1	✓	464.12 (0.90)	251	✓	127.91 (0.58)	4	251
	map	1	0 (+1)	✓	550.10 (0.61)	3075	✓	249.42 (0.73)	4	3075
	replicate5	0	0 (+1)	✓	372.23 (0.70)	1295	✓	435.78 (1.06)	4	1295
	replicate10	0	0 (+1)	✓	2241.87 (4.74)	10773	✓	2898.93 (1.47)	4	10773
	replicateN	1	1	✓	593.86 (1.68)	772	✓	108.98 (0.65)	4	772
	stutter	1	0	✓	1325.36 (1.77)	1792	×	Timeout	-	-
	sum	2	1 (+1)	✓	84.09 (0.25)	208	✓	3236.74 (0.87)	192	3623
Stream	build	0	0 (+1)	✓	61.27 (0.45)	75	✓	84.44 (0.49)	4	75
	map	1	0 (+1)	✓	351.93 (0.91)	1363	✓	153.01 (0.37)	0	1363
	take1	0	0 (+1)	✓	34.02 (0.23)	22	✓	19.32 (0.05)	0	22
	take2	0	0 (+1)	✓	110.18 (0.31)	204	✓	89.10 (0.18)	0	208
	take3	0	0 (+1)	✓	915.39 (1.42)	1139	✓	631.47 (1.14)	0	1172
Bool	neg	0	2	✓	209.09 (0.31)	42	✓	168.37 (0.56)	0	42
	and	0	4	✓	3129.30 (2.82)	786	✓	7069.14 (15.91)	0	2153
	impl	0	4	✓	1735.09 (4.31)	484	✓	3000.48 (4.65)	0	1214
	or	0	4	✓	1213.86 (1.02)	374	✓	2867.74 (3.52)	0	1203
	xor	0	4	✓	2865.79 (4.33)	736	✓	7251.38 (32.06)	0	2229
Maybe	bind	0	0 (+1)	✓	159.87 (0.52)	237	✓	55.33 (0.33)	0	237
	fromMaybe	0	0 (+2)	✓	54.27 (0.35)	18	✓	11.58 (0.10)	0	18
	return	0	0	✓	9.89 (0.02)	17	✓	11.49 (0.04)	4	17
	isJust	0	2	✓	69.33 (0.17)	48	✓	22.07 (0.09)	0	48
	isNothing	0	2	✓	102.42 (0.32)	49	✓	31.89 (0.22)	0	49
	map	0	0 (+1)	✓	54.90 (0.22)	120	✓	22.01 (0.10)	0	120
	mplus	0	1	✓	319.64 (0.47)	318	✓	70.98 (0.05)	0	318
Nat	isEven	1	2	✓	1027.79 (1.28)	466	✓	313.77 (0.92)	8	468
	pred	0	1	✓	46.20 (0.18)	33	✓	48.04 (0.13)	8	33
	succ	0	1	✓	115.16 (0.91)	76	✓	156.02 (0.50)	8	76
	sum	1	1 (+2)	✓	1582.23 (3.60)	751	✓	734.38 (1.41)	12	751
Tree	map	1	0 (+1)	✓	1168.60 (1.21)	4259	✓	525.47 (1.31)	4	4259
	stutter	1	0 (+1)	✓	693.44 (1.21)	832	✓	219.91 (1.02)	4	674
	sum	2	3	✓	1477.83 (1.28)	3230	✓	3532.24 (7.19)	192	3623
Misc	compose	0	0	✓	40.27 (0.08)	38	✓	14.53 (0.09)	2	38
	copy	0	0	✓	5.24 (0.04)	21	✓	6.16 (0.10)	2	21
	push	0	0	✓	26.66 (0.18)	45	✓	14.23 (0.13)	2	45

Table 4.3: Results. μT in *ms* to 2 d.p. with standard sample error in brackets

We briefly examine some of the more complex benchmarks which make use of almost all of our synthesis rules in one program. The `stutter` case from the List class of benchmarks is specified as:

```
stutter : ∀ a . List (a [2]) %1..∞ → List a
spec
    stutter % 0..∞
stutter = ?
```

This is a function which takes a list of values of type `a`, where each element in the list is explicitly graded by `2`, indicating that each element must be used twice. The return type of `stutter` is a list of type `a`. The argument list itself must be used at least once with potential usage extending up to infinity, suggesting that some recursion will be necessary in the program. This is further emphasised by the `spec` , which states that we can use the definition of `stutter` inside the function body in an unrestricted way. From this, we synthesise:

```
stutter Nil = Nil;
stutter (Cons [u] z) = (Cons u) ((Cons u) (stutter z))
```

in 1325ms (∼1.3 seconds). We also have a `stutter` case in the Tree class of benchmarks, which instead performs the above transformation over a binary tree data type, which yields the following program in 693ms (∼0.7 seconds):

```
stutter : ∀ a b . Tree (a [2]) % 1..∞ → Tree (a, a)
spec
    stutter %0..∞
stutter Leaf = Leaf;
stutter (Node y [v] u) = ((Node (stutter y)) (v, v)) (stutter
     u)
```

Lastly, we compare between the number of examples required by Granule (using grades) and the Myth program synthesis tool (based purely on pruning by examples). We take the cases from our benchmark set which have an equivalent in the Myth benchmark suite [Osera and Zdancewic, 2015]. Table 4.4 shows the number of examples used in Granule, and the number required for the equivalent Myth case. In both Granule and Myth this number represents the minimal number of examples required to synthesise the correct program.

		Granule	Myth
Problem		#/Exs	#/Exs
List	append	0	6
	concat	1	6
	snoc	1	8
	drop	1	13
	inc	1	4
	head	1	3
	tail	1	3
	last	1	6
	length	1	3
	map	0	8
	stutter	0	3
	sum	1	3
Bool	neg	2	2
	and	4	4
	impl	4	4
	or	4	4
	xor	4	4
Nat	isEven	2	4
	pred	1	3
Tree	map	0	7

Table 4.4: Number of examples needed for synthesis, comparing Granule vs. Myth

For most of the problems (15 out of 20), Granule required fewer examples to identify the desired program in synthesis. The disparity in the number of examples required is quite significant in some cases: with 13 examples required by Myth to synthesise the *concat* problem but only 1 example for Granule. This shows the pruning power of graded information in synthesis, confirming H2.

4.6 SYNTHESIS OF LINEAR HASKELL PROGRAMS

As part of a growing trend of resourceful types being added to more mainstream languages, Haskell has introduced support for linear types as of GHC 9, using an underlying graded type system which can be enabled as a language extension of GHC's existing type system [Bernardy et al., 2018] (the `LinearType` extension). This system is closely related to Granule, but limited only to one semiring for its grades. This however presents an exciting opportunity: can we leverage our tool to synthesise (linear) Haskell programs?

Like Granule, grades in Linear Haskell can be expressed as "multiplicities" on function types: `a %r -> b`. The multiplicity r can be either 1 or ω (or polymorphic), with 1 denoting linear usage (also written as `'One`) and ω for (`'Many`) unrestricted usage. Likewise, in Granule, we can model linear types using the 0-1-ω semiring [Hughes et al., 2021].

Synthesising Linear Haskell programs then simply becomes a task of parsing a Haskell type into a Granule equivalent, synthesising a term from this, and compiling the synthesised term back into Haskell. The close syntactic correspondence between Granule and Haskell makes this translation straightforward.

Our implementation includes a prototype synthesis tool using this approach. A synthesis problem takes the form of a Linear Haskell program with a hole, e.g.

```
{-# LANGUAGE LinearTypes #-}
swap :: (a, b) %One -> (b, a)
swap = _
```

We invoke the tool with `gr --linear-haskell swap.hs` producing:

```
swap (z, y) = (y, z)
```

Haskell tuples are converted into a Granule data type, generated based on the tuple's arity. This data type is given unique name based on this size, e.g. `,4` for a tuple of arity 4. Using the special character `,` in the constructor name prevents name clashes with other in-scope data constructors, as this character would not be permitted in a typical user-defined constructor.

Users may make use of lists, tuples, `Maybe` and `Either` data types from Haskell's prelude, as well as user-defined ADTs. Further integration of the tool, as well as support for additional Haskell features such as GADTs is left as future work.

This tool can be especially useful as a programming aid when writing linear versions of Haskell libraries. To conclude this section, we showcase some of the programs synthesised via our Linear Haskell tool with the goal of writing a Linear Haskell library for Haskell's `Maybe` data type.

```
{-# LANGUAGE LinearTypes #-}
```

```
maybe :: b %Many -> (a %One -> b) %Many -> (Maybe a) %One -> b
maybe x y Nothing = x
maybe x y (Just u) = y u

bind :: Maybe a %One -> (a %One -> Maybe b) %Many -> Maybe b
bind Nothing y = Nothing
bind (Just z) y = y z

map :: (a %One -> b) %Many -> Maybe a %One -> Maybe b
map x Nothing = Nothing
map x (Just z) = Just (x z)

maybeToList :: Maybe a %One -> [a]
maybeToList Nothing = []
maybeToList (Just y) = [y]

fromMaybe :: Maybe a %One -> a %Many -> a
fromMaybe Nothing y = y
fromMaybe (Just z) y = z
```

4.7 CONCLUSION

SUBTRACTIVE RESOURCE MANAGEMENT Ultimately, we opted to focus on the additive resource management scheme for Chapter 4 due to the evidence regarding the efficiency of its implementation for a fully graded typing calculus (i.e. that the subtractive approach required a far greater amount of time solving constraints).

In the subtractive scheme, the VAR rule generates a constraint which determines if the use of a variable is permissible based on the rest of the partially synthesised program:

$$\frac{\exists s.\ r \sqsupseteq s + 1}{\Gamma, x :_r A \vdash A \Rightarrow x \mid \Gamma, x :_s A}\ \textsc{Var}$$

i.e., r must overapproximate one use of x plus the future use of x given by the existential s, as in the synthesis rule for the subtractive synthesis calculus in Chapter 3 on page 39.

Our comparative evaluation of the additive and subtractive schemes in Section 3.7 showed that these, and other associated constraints from the subtractive approach, are larger, typically more complex, and are discharged more frequently than their counterparts in the additive system (every time a variable usage is being considered in the above case). Our evaluation concluded that the only situation where subtractive decisively outperformed additive was on purely linear programs. This, coupled with the fact that the subtractive approach has limitations in the presence of polymorphic grades, influenced our decision to adopt the additive scheme, especially as we considered much more complex programs than in the calculi of Chapter 3, e.g., targeting recursion.

In this chapter we presented a synthesis calculus for a feature-rich type system based on the fully graded λ-calculus of section 2.3.2, which complements our graded linear system of Chapter 3 to allow us to synthesise programs for the major approaches to graded type systems. This approach to synthesis enabled us to synthesise Haskell programs using graded types via the Linear Types language extension. We found through our experimental evaluation that synthesising programs in this calculus requires the exploration of fewer synthesis paths when using the information provided by grades to prune the search space of candidate programs. In terms of pure speed, un-graded type-directed program synthesis outperformed our implementation, largely due to the overhead of discharging constraints to an SMT solver.

This discrepancy in speed necessitates that our synthesis tool seek a more efficient means of interfacing with the solver. However, the theoretical advantage of grade-based pruning remains clear: the search space was the same or reduced in all but one of our benchmarking examples in Table 4.3.

In Section 3.8 of Chapter 3 we considered the problem of synthesising programs that distribute a graded modality over a data type. There, our inability to perform *deep* pattern matching in our core calculus rendered us unable to synthesise programs for these *distributive laws*, an issue which is resolved in this chapter.

In the following chapter, we present an alternative approach to generating programs which exhibit this distributive behaviour using a generic programming methodology. The approach we present in Chapter 5 is not type-directed program synthesis of the kind we have seen in this chapter, i.e. it is not based on enumerative search. This approach complements the synthesis calculi presented here and in Chapter 3, providing users with a means to automatically generate programs purely from a type for a common class of graded programs, as well as some other useful structural combinators.

5

AUTOMATICALLY DERIVING GRADED COMBINATORS

Thus far we have considered program synthesis from the perspective of enumerative search, using our types to guide us and pruning the space of programs where possible. This approach yielded a synthesis tool which was highly expressive, allowing the synthesis of a program term for each syntactic form in our core calculus. In this chapter we present an alternative approach, which targets a specific class of graded programs: graded *distributive* and structural combinators. We view this approach as a useful complement to the more powerful type-directed synthesis. Much of the content of this chapter is derived from Hughes et al. [2020], our Linearity/TLLA 2020 paper.

When programming with graded modal types, we have observed there is often a need to 'distribute' a graded modality over a type, and vice versa, in order to compose programs. That is, we may find ourselves in possession of a $\Box_r(\mathsf{F}\alpha)$ value (for some parametric data type F) which needs to be passed to a pre-existing function (of our own codebase or a library) which requires a $\mathsf{F}(\Box_r\alpha)$ value, or perhaps vice versa. A *distributive law* (in the categorical sense, due to Street [1972]) provides a conversion from one to the other. In this chapter, we present a procedure to automatically synthesise these distributive operators, applying a generic programming methodology [Hinze, 2000] to compute these operations given the base type (e.g., $\mathsf{F}\alpha$ in the above description). This serves to ease the use of graded modal types in practice, removing boilerplate code by automatically generating these 'interfacing functions' on-demand, for user-defined data types as well as built-in types.

Throughout, we refer to distributive laws of the form $\Box_r(\mathsf{F}\alpha) \to \mathsf{F}(\Box_r\alpha)$ as *push* operations (as they 'push' the graded modality inside the type constructor F), and dually $\mathsf{F}(\Box_r\alpha) \to \Box_r(\mathsf{F}\alpha)$ as *pull* operations (as they 'pull' the graded modality outside F). We have already encountered some examples of push and pull operations in Chapter 3; our benchmarks showcased pull for product and sum types (i.e. $\mathsf{F} = \otimes$ and $\oplus$, respectively), and push for linear function types ($\mathsf{F} = \multimap$).

As a standalone methodology for generating a common class of graded programs, this "deriving mechanism" serves as a complement

to the synthesis calculi of Chapters 3 and 4. In many cases the solution programs to these distributive problems are straightforwardly derivable from the type alone, making the costly enumerative search of type-directed synthesis unnecessary. One lens through which this can be viewed is as a library of patterns for solving a class of common synthesis problems when dealing with graded types.

Thus, we present a tool for an automatic procedure which calculates distributive laws from graded types and present a formal analysis of their properties. This approach is realised in Granule, embedded into the compiler.

ROADMAP We begin with a motivating example in Section 5.1 before defining our extended calculus in Section 5.2 (which extends the calculus of Figure 2.3 in much the same way that 4 extends 2.4) providing an idealised, simply-typed subset of *LinearBase* Granule with which we develop the core deriving mechanism. Section 5.3 gives the procedures for deriving *push* and *pull* operators for the calculus. Section 5.4 describes the details of how these procedures are realised in the Granule language. We then provide examples of how other structural combinators in Granule may be derived using this tool in Section 5.5. Finally, Section 6.3 discusses related work, while Section 5.6 highlights future work.

We start with a motivating example typifying the kind of software engineering impedance problem that distributive laws solve. We do so in Granule code since it is the main vehicle for the developments here.

5.1 MOTIVATING EXAMPLE

Consider the situation of projecting the first element of a pair. In Granule, this first-projection is defined and typed as the following polymorphic function (whose syntax is reminiscent of Haskell or ML):

```
fst : ∀ { a b : Type, s : Semiring } . (a, b [0 : s]) → a
fst (x, [y]) = x
```

Linearity is the default, so this represents a linear function applied to linear values. However, the second component of the pair is graded, allowing weakening to be applied via unboxing in the body to discard `y` of type `b`. In graded linear λ-calculus of Section 2.3.1, we denote '`b [0]`' as the type $\Box_0 b$.

The type for `fst` is however somewhat restrictive: what if we are trying to use such a function with a value (call it `myPair`) whose type is not of the form `(a, b [0])` but rather `(a, b) [r]` for some grading term `r` which permits weakening? Such a situation readily arises when we are composing functional code, say between libraries or between a library and user code. In this situation, `fst myPair` is ill-typed. Instead, we could define a different first projection function for use with `myPair`:

```
fst' : ∀ { a b : Type, s : Semiring, r : s }
     . {0 ⩽ r} ⇒ (a, b) [r] → a
fst' [(x, y)] = x
```

This implementation uses various language features of Granule to make it as general as possible. Firstly, the function is polymorphic in the grade `r` and in the semiring `s` of which `r` is an element. Next, a refinement constraint `0 ⩽ r` specifies that by the pre-ordering $\leqslant$ associated with the semiring `s`, that `0` is approximated by `r` (essentially, that `r` permits weakening). The rest of the type and implementation looks more familiar for computing a first projection, but now the graded modality is over the entire pair.

From a software engineering perspective, it is cumbersome to create alternate versions of generic combinators every time we are in a slightly different situation with regards the position of a graded modality. Fortunately, this is an example to which a general *distributive law* can be deployed. In this case, we could define the following distributive law of graded modalities over products, call it `pushPair`:

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

This 'pushes' the graded modality `r` into the pair (via pattern matching on the modality and the pair inside it, and then reintroducing the modality on the right hand side via `[x]` and `[y]`), distributing the graded modality to each component. Given this combinator, we can now apply `fst (pushPair myPair)` to yield a value of type `a [r]`, on which we can apply the Granule standard library function `extract`:

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

to get the original `a` value we desired:

```
extract (fst (pushPair myPair)) : a
```

The `pushPair` function could be provided by the standard library, and thus we have not had to write any specialised combinators ourselves: we have applied supplied combinators to solve the problem.

Now imagine we have introduced some custom data type `List` on which we have a *map* function:

```
data List a = Cons a (List a) | Nil

map : ∀ { a b : Type } . (a → b) [0..∞] → List a → List b
map [f] Nil = Nil;
map [f] (Cons x xs) = Cons (f x) (map [f] xs)
```

Note that, via a graded modality, the type of `map` specifies that the parameter function, of type `a → b` is non-linear, used between 0 and ∞ times. Imagine now we have a value `myPairList : (List (a, b)) [r]`

and we want to map first projection over it. But `fst` expects `(a, b [0])` and even with `pushPair` we require `(a, b) [r]`. *We need another distributive law*, this time of the graded modality over the `List` data type. Since `List` was user-defined, we now must roll our own `pushList` operation with type `List ((a, b) [r]) → List (a [r], b [r])`, and so we are back to making specialised combinators for our data types.

The crux of this chapter is that such distributive laws can be automatically calculated given the definition of a type. With our Granule implementation of this approach (Section 5.4), we can then solve this combination problem via the following composition of combinators:

```
map (extract . fst . push @(,)) (push @List myPairList) :
    List a
```

where the `push` operations are written with their base type via `@` (a type application) and whose definitions and types are automatically generated during type checking. Thus the `push` operation is a *data-type generic function* [Hinze, 2000]. This generic function is defined inductively over the structure of types, thus a programmer can introduce a new user-defined algebraic data type and have the implementation of the generic distributive law derived automatically. This reduces both the initial and future effort (e.g., if an ADT definition changes or new ADTs are introduced). Furthermore, this technique is much less expensive than a proof search technique as no SMT solving on grade equations is required.

Dual to the above, there are situations where a programmer may wish to *pull* a graded modality out of a structure. This is possible with a dual distributive law, which could be written by hand as:

```
pullPair : ∀ { a b : Type, s : Semiring, m n : s }
         . (a [n], b [m]) → (a, b) [n ⊓ m]
pullPair ([x], [y]) = [(x, y)]
```

Note that the resulting grade is defined by the greatest-lower bound (meet) of `n` and `m`, if it exists as defined by a pre-order for semiring `s` (that is, ⊓ is not a total operation). This allows some flexibility in the use of the *pull* operation when grades differ in different components but have a greatest-lower bound which can be 'pulled out'. Our approach also allows such operations to be generically derived.

5.2 EXTENDING THE GRADED LINEAR-λ-CALCULUS

We define here a calculus which extends the graded linear λ-calculus of 2.3.1 with data constructors, pattern matching, and recursive data types. This language constitutes a simplified monomorphic subset of Granule. We include notions of data constructors and their elimination via **case** expressions as a way to unify the handling of regular type constructors.

$$\frac{\Gamma_1, x : A \vdash t_1 : A \qquad \Gamma_2, x : A \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{letrec}\ x\ =\ t_1\ \textbf{in}\ t_2 : B}\ \textsc{LetRec}$$

$$\frac{(C : B_1 \multimap ... \multimap B_n \multimap A) \in D}{\emptyset\ \vdash\ C : B_1 \multimap ... \multimap B_n \multimap A}\ \textsc{Con}$$

$$\frac{\Gamma \vdash t : A \quad \cdot \vdash p_i : A \triangleright \Delta_i \quad \Gamma', \Delta_i \vdash t_i : B}{\Gamma + \Gamma' \vdash \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; ...; p_n \mapsto t_n : B}\ \textsc{Case}$$

Figure 5.1: Typing rules for the extended graded linear λ-calculus

The full syntax of terms and types is given by:

$$\begin{aligned}
t &::= x \mid t_1\, t_2 \mid \lambda x.t \mid [t] \mid C\, t_0 \ldots t_n \\
&\quad \mid \textbf{case}\ t\ \textbf{of}\ p_1 \mapsto t_1; ...; p_n \mapsto t_n \mid \textbf{letrec}\ x\ =\ t_1\ \textbf{in}\ t_2 && \text{(terms)} \\
p &::= x \mid _ \mid [p] \mid C\, p_0 \ldots p_n && \text{(patterns)} \\
A, B &::= A \multimap B \mid \alpha \mid A \otimes B \mid A \oplus B \mid \mathsf{Unit} \mid \Box_r A \mid \mu X.A \mid X && \text{(types)} \\
C &::= () \mid \mathsf{inl} \mid \mathsf{inr} \mid (,) && \text{(data constructors)}
\end{aligned}$$

Terms comprise the graded linear λ-calculus of Section 2.3.1, extended with data constructors, case statements for pattern matching over a scrutinee term, and standard recursive let bindings. Patterns may either be variable, wildcard, box, or constructor patterns. A notable difference from Section 2.3.1 is that unboxing here takes place via case pattern matching, instead of a specialised let expression, i.e. **let** $[x] = y$ **in** t becomes **case** y **of** $[x] \mapsto t$

The syntax of types is fairly straightforward. We make type variables explicit in our syntax through the variable α to allow distributive laws to be derived on parametric types, and X represents a set of recursion variables which are not exposed to the user. For the most part, typing follows the calculus defined in section 3.1. Figure 5.1 gives the additional rules. We briefly explain the extensions introduced for this chapter.

The LetRec rule provides recursive bindings in the standard way.

Data constructors with zero or more arguments are introduced via the Con rule. Here, the constructors that concern us are units, products, and coproducts (sums), given by D, a global set of data constructors with their types, defined:

$$\begin{aligned}
D = &\{() : \mathsf{Unit}\} \\
&\cup \{(,) : A \multimap B \multimap A \otimes B \mid \forall A, B\} \\
&\cup \{\mathsf{inl} : A \multimap A \oplus B \mid \forall A, B\} \\
&\cup \{\mathsf{inr} : B \multimap A \oplus B \mid \forall A, B\}
\end{aligned}$$

$$\frac{}{\cdot \vdash x : A \triangleright x : A}\ \textsc{Pvar} \qquad \frac{\cdot \vdash p_i : B_i \triangleright \Gamma_i}{\cdot \vdash C p_1 .. p_n : A \triangleright \Gamma_1, .., \Gamma_n}\ \textsc{Pcon}$$

$$\frac{r \vdash p : A \triangleright \Gamma}{\cdot \vdash [p] : \Box_r A \triangleright \Gamma}\ \textsc{Pbox}$$

$$\frac{r \vdash p_i : B_i \triangleright \Gamma_i \qquad |A| > 1 \Rightarrow 1 \sqsubseteq r}{r \vdash C p_1 .. p_n : A \triangleright \Gamma_1, .., \Gamma_n}\ [\textsc{Pcon}]$$

$$\frac{}{r \vdash x : A \triangleright x :_r A}\ [\textsc{Pvar}] \qquad \frac{0 \sqsubseteq r}{r \vdash \_ : A \triangleright \varnothing}\ [\textsc{Pwild}]$$

Figure 5.2: Pattern typing rules for the extended graded linear λ-calculus

Constructors are eliminated by pattern matching via the CASE rule. Patterns p are typed by judgments of the form $?r \vdash p : A \triangleright \Delta$ meaning that a pattern p has type A and produces a context of typed binders Δ (used, e.g., in the typing of the case branches). The information to the left of the turnstile denotes optional grade information arising from being in an unboxing pattern and is syntactically defined as either:

$$?r ::= - \mid r \qquad \text{(enclosing grade)}$$

where $-$ means the present pattern is not nested inside an unboxing pattern and r that the present pattern is nested inside an unboxing pattern for a graded modality with grade r.

The rules of pattern typing are given in Figure 5.2. The rule (PBox) provides graded modal elimination (an 'unboxing' pattern), propagating grade information into the typing of the sub-pattern. Thus **case** t **of** $[p] \to t'$ can be used to eliminate a graded modal value. Variable patterns are typed via two rules depending on whether the variable occurs inside an unbox pattern ([Pvar]) or not (Pvar), with the [Pvar] rule producing a binding with the grade of the enclosing box's grade r. As with variable patterns, constructor patterns are split between rules for patterns which either occur inside an unboxing pattern or not. In the former case, the grade information is propagated to the sub-pattern(s), with the additional constraint that if there is more than one data constructor for the type A (written $|A| > 1$), then the grade r must approximate 1 (written $1 \sqsubseteq r$) as pattern matching

incurs a usage to inspect the constructor. The operation $|A|$ counts the number of data constructors for a type:

$$\begin{aligned}
|\mathsf{Unit}| &= 1 \\
|A \multimap B| &= 1 \\
|\Box_r A| &= |A| \\
|A \oplus B| &= |A| + |B| \\
|A \otimes B| &= |A||B| \\
|\mu X.A| &= |A[\mu X.A / X]|
\end{aligned}$$

and $|X|$ is undefined (or effectively 0) since we do not allow unguarded recursion variables in types. A type A must therefore involve a sum type for $|A| > 1$.

Since a wildcard pattern _ discards a value, this is only allowed inside an unboxing pattern where the enclosing grade permits weakening, captured via $0 \sqsubseteq r$ in rule [PWILD].

One can note that this calculus is very similar to that of Chapter 4, extending the core calculus of Figure 2.4 with ADTS, recursion, and pattern matching. For practicality, we also introduce an explicit **letrec** construct in this calculus, which was not present in Chapter 4.

5.3 AUTOMATICALLY DERIVING *push* AND *pull*

Now that we have established the language, we describe the algorithmic calculation of distributive laws. Universal quantification over type variables (α) takes place implicitly prior to running the deriving mechanism, thus type variables may be treated as logical atoms.

5.3.1 *Notation*

Let $\mathsf{F} : \mathsf{Type}^n \to \mathsf{Type}$ be an n-ary type constructor (i.e. a constructor which takes n type arguments), whose free type variables provide the n parameter types. We write $\mathsf{F}\overline{\alpha_i}$ for the application of F to type variables α_i for all $1 \leq i \leq n$.

5.3.2 *Push*

We automatically calculate *push* for F applied to n type variables $\overline{\alpha_i}$ as the operation:

$$[\![\mathsf{F}\overline{\alpha_i}]\!]_{\mathsf{push}} : \Box_r \mathsf{F}\overline{\alpha_i} \multimap \mathsf{F}(\overline{\Box_r \alpha_i})$$

where we require $1 \sqsubseteq r$ *if* $|\mathsf{F}\overline{\alpha_i}| > 1$ due to the [PCON] rule (e.g., if F contains a sum).

$$
\begin{aligned}
[\![\mathsf{Unit}]\!]^{\Sigma}_{\mathsf{push}}\ z &= \mathbf{case}\ z\ \mathbf{of}\ [()] \to () \\
[\![\alpha]\!]^{\Sigma}_{\mathsf{push}}\ z &= z \\
[\![X]\!]^{\Sigma}_{\mathsf{push}}\ z &= \Sigma(X)\ z \\
[\![A \oplus B]\!]^{\Sigma}_{\mathsf{push}}\ z &= \mathbf{case}\ z\ \mathbf{of}\ \ [\mathsf{inl}\ x] \to \mathsf{inl}\ [\![A]\!]^{\Sigma}_{\mathsf{push}}[x]; \\
&\qquad\qquad\qquad\qquad\ \ [\mathsf{inr}\ y] \to \mathsf{inr}\ [\![B]\!]^{\Sigma}_{\mathsf{push}}[y] \\
[\![A \otimes B]\!]^{\Sigma}_{\mathsf{push}}\ z &= \mathbf{case}\ z\ \mathbf{of}\ [(x,y)] \to ([\![A]\!]^{\Sigma}_{\mathsf{push}}[x], [\![B]\!]^{\Sigma}_{\mathsf{push}}[y]) \\
[\![A \multimap B]\!]^{\Sigma}_{\mathsf{push}}\ z &= \lambda y.\mathbf{case}\ z\ \mathbf{of}\ [f] \to \\
&\qquad\qquad\qquad\qquad \mathbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{pull}}\ y\ \mathbf{of}\ [u] \to [\![B]\!]^{\Sigma}_{\mathsf{push}}[(f\ u)] \\
[\![\mu X.A]\!]^{\Sigma}_{\mathsf{push}}\ z &= \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \mu X.\Box_r A \multimap (\mu X.A)[\overrightarrow{\Box_r \alpha_i / \alpha_i}]}_{\mathsf{push}}\ \mathbf{in}\ f\ z
\end{aligned}
$$

Figure 5.3: Interpretation rules for $[\![A]\!]_{\mathsf{push}}$

For types A closed with respect to recursion variables, let $[\![A]\!]_{\mathsf{push}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{push}}\ z$ given by an intermediate interpretation $[\![A]\!]^{\Sigma}_{\mathsf{push}}$ where Σ is a context of *push* combinators for the recursive type variables. This interpretation is defined by Figure 5.3. In the case of *push* on a value of type Unit, we pattern match on the value, eliminating the graded modality via the unboxing pattern match and returning the unit value. For type variables α, *push* is simply the identity of the value, while for recursion variables X we lookup the variable's binding in Σ and apply it to the value z. For sum and product types, *push* works by pattern matching on the type's constructor(s) and then inductively applying *push* to the promoted arguments, re-applying them to the constructor(s). Unlike *pull* below, the *push* operation can be derived for function types, with a contravariant use of *pull*. For recursive types, we inductively apply *push* to the value with a fresh recursion variable bound in Σ, representing a recursive application of push. There is no derivation of a distributive law for types which are themselves graded modalities, as this would depend on the particular graded modalities being distributed (see Gaboardi et al. [2016]).

Section B.2.3 in Appendix B gives the proof that $[\![A]\!]_{\mathsf{push}}$ is type sound, i.e., its derivations are well-typed:

Proposition 1 (Type soundness of $[\![\mathsf{F}\ \overline{\alpha_i}]\!]^{\Sigma}_{\mathsf{push}}$). $[\![\mathsf{F}\overline{\alpha_i}]\!]^{\Sigma}_{\mathsf{push}} : \Box_r \mathsf{F}\overline{\alpha_i} \to \mathsf{F}(\overline{\Box_r \alpha_i})$

$$
\begin{aligned}
[\![\mathsf{Unit}]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \mathbf{case}\ z\ \mathbf{of}\ () \to [()] \\
[\![\alpha]\!]^{\Sigma}_{\mathsf{pull}}\ z &= z \\
[\![X]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \Sigma(X)\ z \\
[\![A \oplus B]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \mathbf{case}\ z\ \mathbf{of}\ \ \mathsf{inl}\ x \to \mathbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{pull}}\ x\ \mathbf{of}\ [u] \to [\mathsf{inl}\ u]; \\
&\qquad\qquad\qquad\qquad\quad \mathsf{inr}\ y \to \mathbf{case}\ [\![B]\!]^{\Sigma}_{\mathsf{pull}}\ y\ \mathbf{of}\ [v] \to [\mathsf{inr}\ v] \\
[\![A \otimes B]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \mathbf{case}\ z\ \mathbf{of}\ (x, y) \to \\
&\qquad\qquad \mathbf{case}\ ([\![A]\!]^{\Sigma}_{\mathsf{pull}}\ x, [\![B]\!]^{\Sigma}_{\mathsf{pull}}\ y)\ \mathbf{of}\ ([u], [v]) \to [(u, v)] \\
[\![\mu X.A]\!]^{\Sigma}_{\mathsf{pull}}\ z &= \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \mu X.A[\overrightarrow{\Box_{r_i}\alpha_i / \alpha_i}] \multimap \Box_{\prod_1^n r_i}(\mu X.A)}_{\mathsf{pull}}\ \mathbf{in}\ f\ z
\end{aligned}
$$

Figure 5.4: Interpretation rules for $[\![A]\!]_{\mathsf{pull}}$

5.3.3 *Pull*

We automatically calculate *pull* for F applied to n type variables $\overline{\alpha_i}$ as the operation:

$$[\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{pull}} : \mathsf{F}\ (\overline{\Box_{r_i}\alpha_i}) \multimap \Box_{\prod_{i=1}^n r_i}(\mathsf{F}\ \overline{\alpha_i})$$

Type constructor F here is applied to n arguments each of the form $\Box_{r_i}\alpha_i$, i.e., each with a different grading of which the greatest-lower bound $\prod_{i=1}^n r_i$ (derived from the semiring's pre-order, as per Definition 2.3.1) is the resulting grade (see `pullPair` from Section 5.1).

For types A closed with respect to recursion variables, let $[\![A]\!]_{\mathsf{pull}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{pull}}\ z$ given by an intermediate interpretation $[\![A]\!]^{\Sigma}_{\mathsf{pull}}$ where Σ is a context of *pull* combinators for the recursive type variables. This interpretation is defined by Figure 5.4.

Just like *push*, we cannot apply *pull* to graded modalities themselves. Unlike *push*, we cannot apply *pull* to function types. That is, we cannot derive a distributive law of the form $(\Box_r A \multimap \Box_r B) \multimap \Box_r(A \multimap B)$. This is because introducing the concluding $\Box_r$ would require the incoming function $(\Box_r A \multimap \Box_r B)$ to itself be inside $\Box_r$ due to the promotion rule (PR), which does not match the type scheme for *pull*.

The rest of the derivation above is similar but dual to that of *push*.

Section B.2.3 in Appendix B gives the proof that $[\![A]\!]_{\mathsf{pull}}$ is type sound, i.e., its derivations are well-typed:

Proposition 2 (Type soundness of $[\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{pull}}$)**.** $\mathsf{F}\ (\overline{\Box_{r_i}\alpha_i}) \to \Box_{\prod_{i=1}^n r_i}(\mathsf{F}\ \overline{\alpha_i})$

Example 5.3.1. To illustrate the above procedures, the derivation of $\lambda z.[\![(\alpha \otimes \alpha) \multimap \beta]\!]_{\mathsf{push}}\ z : \Box_r((\alpha \otimes \alpha) \multimap \beta) \multimap ((\Box_r\alpha \otimes \Box_r\alpha) \multimap \Box_r\beta)$ is:

$$
\begin{aligned}
&\lambda z.[\![(\alpha \otimes \alpha) \multimap \beta]\!]^{\emptyset}_{\mathsf{push}}\ z \\
&= \lambda z.\lambda y.\mathbf{case}\ z\ \mathbf{of}\ [f] \to \mathbf{case}\ [\![\alpha \otimes \alpha]\!]^{\emptyset}_{\mathsf{pull}}\ y\ \mathbf{of}\ [u] \to [\![\beta]\!]^{\emptyset}_{\mathsf{push}}[(f\ u)] \\
&= \lambda z.\lambda y.\mathbf{case}\ z\ \mathbf{of}\ [f] \to \\
&\qquad \mathbf{case}\ (\mathbf{case}\ y\ \mathbf{of}\ (x', y') \to \\
&\qquad\qquad \mathbf{case}\ ([\![\alpha]\!]^{\emptyset}_{\mathsf{pull}}\ x', [\![\alpha]\!]^{\emptyset}_{\mathsf{pull}}\ y')\ \mathbf{of}\ ([u], [v]) \to [(u, v)])\ \mathbf{of} \\
&\qquad\quad [u] \to [\![\beta]\!]^{\emptyset}_{\mathsf{push}}[(f\ u)] \\
&= \lambda z.\lambda y.\mathbf{case}\ z\ \mathbf{of}\ [f] \to \\
&\qquad \mathbf{case}\ (\mathbf{case}\ y\ \mathbf{of}\ (x', y') \to \\
&\qquad\qquad \mathbf{case}\ (x', y')\ \mathbf{of}\ ([u], [v]) \to [(u, v)])\ \mathbf{of}\ [u] \to [(f\ u)]
\end{aligned}
$$

Remark 3. One might ponder whether linear logic's exponential $!A$ [Girard, 1987] is modelled by the graded necessity modality over $\mathbb{N}_\infty$ intervals, i.e., with $!A \triangleq \Box_{0..\infty}A$. This is a reasonable assumption, but $\Box_{0..\infty}A$ has a slightly different meaning to $!A$, exposed here: whilst $[\![A \otimes B]\!]_{\mathsf{push}} : \Box_{0..\infty}(A \otimes B) \multimap (\Box_{0..\infty}A \otimes \Box_{0..\infty}B)$ is derivable in our language, linear logic does not permit $!(A \otimes B) \multimap (!A \otimes !B)$. Models of ! provide only a monoidal functor structure which gives *pull* for $\otimes$, but not *push* [Benton et al., 1992]. This structure can be recovered in Granule through the introduction of a partial type-level operation which selectively disallows *push* for $\otimes$ in semirings which model the ! modality of linear logic. An example of such a semiring is the {Zero, One, ω} semiring we encountered in Example 2.3.2 on page 20, which is intended to specifically model linear logic's ! modality.[1]

The algorithmic definitions of 'push' and 'pull' can be leveraged in a programming context to automatically yield these combinators for practical purposes. We discuss how this is leveraged inside the Granule compiler in Section 5.4. Before that, we study the algebraic behaviour of the derived distributive laws.

5.3.4 *Properties*

We note that these distributive laws are mutually inverse:

Proposition 5.3.1 (Pull is right inverse to push)**.** *For all n-arity types* F *which do not contain function types, then for type variables* $(\alpha_i)_{i \in 1 \leq i \leq n}$ *and for all grades* $r \in \mathcal{R}$ *where* $1 \sqsubseteq r$ *if* $|\mathsf{F}\overline{\alpha_i}| > 1$*, then:*

$$[\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{pull}}([\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{push}}) = id\ : \Box_r\mathsf{F}\overline{\alpha_i} \multimap \Box_r\mathsf{F}\overline{\alpha_i}$$

1 The work in Hughes et al. [2021] arose as a result of this work.

Proposition 5.3.2 (Pull is left inverse to push). *For all n-arity types* F *which do not contain function types, then for type variables* $(\alpha_i)_{i \in 1 \leq i \leq n}$ *and for all grades* $r \in \mathcal{R}$ *where* $1 \sqsubseteq r$ *if* $|\mathsf{F}\overline{\alpha_i}| > 1$*, then:*

$$[\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{push}}([\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{pull}}) = id\ : \mathsf{F}(\Box_r \overline{\alpha_i}) \multimap \mathsf{F}(\Box_r \overline{\alpha_i})$$

Section B.2.4 of Appendix B gives the proofs, leveraging a typed equational theory for our lanugage. This equational theory is defined in Section B.2.1 of Appendix B.

Additional properties of these distributive laws can be found in Hughes et al. [2020]. Prima facie, the above *push* and *pull* operations are simply distributive laws between two (parametric) type constructors F and $\Box_r$, the latter being the graded modality. However, both F and $\Box_r$ have additional structure. If the mathematical terminology of 'distributive laws' is warranted, then such additional structure should be preserved by *push* and *pull* (e.g., as in how a distributive law between a monad and a comonad must preserve the behaviour of the monad and comonad operations after applying the distributive law [Power and Watanabe, 2002]). We choose to omit discussion of these properties here as they are less directly relevant to the purpose of this thesis.

5.4 IMPLEMENTATION IN GRANULE

The Granule type checker implements the algorithmic derivation of *push* and *pull* distributive laws as covered in the previous section. Whilst the syntax of our language types had unit, sum, and product types as primitives, in Granule these are provided by a more general notion of type constructor which can be extended by user-defined, generalized algebraic data types (GADTs). The procedure outlined in Section 5.3 is therefore generalised slightly so that it can be applied to any data type: the case for $A \oplus B$ is generalised to types with an arbitrary number of data constructors.

Our deriving mechanism is exposed to programmers via explicit (visible) type application (akin to that provided in GHC Haskell [Eisenberg et al., 2016]) on reserved names `push` and `pull`. Written `push @T` or `pull @T`, this signals to the compiler that we wish to derive the corresponding distributive laws at the type `T` (where `T` is an n-ary type constructor). For example, for the `List : Type → Type` data type from the standard library, we can write the expression `push @List` which the type checker recognises as a function of type:

```
push @List : ∀ { a : Type, s : Semiring, r : s }
           . {1 ⩽ r} ⇒ (List a) [r] → List (a [r])
```

Note this function is not only polymorphic in the grade, but polymorphic in the semiring itself. Granule identifies different graded

modalities by their semirings, and thus this operation is polymorphic in the graded modality. When the type checker encounters such a type application, it triggers the derivation procedure of Section 5.3, which also calculates the type. The result is then stored in the state of the frontend to be passed to the interpreter (or compiler) after type checking. The derived operations are memoized so that they need not be re-calculated if a particular distributive law is required more than once. Otherwise, the implementation largely follows Section 5.3 without surprises, apart from some additional machinery for specialising the types of data constructors coming from (generalized) ADTs.

5.4.1 *Examples*

Earlier, we motivated the crux of the work in this chapter with a concrete example, which we can replay here in Granule, using its type application technique for triggering the automatic derivation of the distributive laws. Previously, we defined `pushPair` by hand which can now be replaced with:

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

Note that in Granule `(,)` is an infix type constructor for product types as well as terms. We can then replace the previous `fst'` with:

```
fst' : ∀ { a b : Type, r : Semiring }
     . {0 ⩽ r} ⇒ (a, b) [r] → a
fst' = let [x'] = fst (push @(,) x) in x'
```

The point however in the example is that we need not even define this intermediate combinator, but can instead write the following wherever we need to compute the first projection of `myPair : (a, b) [r]`:

```
extract (fst (push @(,) myPair)) : a
```

We already saw that we can then generalise this by applying this first projection inside of the list directly using `push @List`:

```
map (extract . fst . push @(,)) (push @List myPairList) :
    List a
```

where `myPairList : (List (a, b)) [r]`.

In a slightly more elaborate example, we can use the `pull` combinator for pairs to implement a function that duplicates a pair (given that both elements can be consumed twice):

```
copyPair : ∀ { a, b : Type }
         . (a [0..2], b [2..4]) → ((a, b), (a, b))
-- where, copy : a [2] → (a, a)
copyPair x = copy (pull @(,) x)
```

Note `pull` computes the greatest-lower bound of intervals `0..2` and `2..4` which is `2..2`, i.e., we can provide a pair of `a` and `b` values which can each be used exactly twice: exactly what is required for `copy`.

As another example, interacting with Granule's indexed types (GADTs), consider a simple programming task of taking the head of a sized-list (vector) and duplicating it into a pair. The `head` operation is typed:

```
head : ∀ { a : Type, n : Nat }
     . (Vec (n + 1) a) [0..1] → a
```

which has a graded modal input with grade `0..1` meaning the input vector is used 0 or 1 times: the head element is used once (linearly) for the return but the tail is discarded.

This head element can then be copied via a graded modality, e.g., a value of type `(Vec (n + 1) (a [2])) [0..1]` permits:

```
copyHead' : ∀ { a : Type, n : Nat :}
          . (Vec (n + 1) (a [2])) [0..1] → (a, a)
-- [y] unboxes (a [2]) to y:a usable twice
copyHead' xs = let [y] = head xs in (y, y)
```

Here we "unbox" the graded modal value of type `a [2]` to get a non-linear variable `y` which we can use precisely twice. However, what if we are in a programming context where we have a value `Vec (n + 1) a` with no graded modality on the type `a`? We can employ two idioms here: (i) take a value of type `(Vec (n + 1) a) [0..2]` and split its modality in two: `(Vec (n + 1) a) [2] [0..1]`, and then (ii) use *push* on the inner graded modality `[2]` to get `(Vec (n + 1) (a [2])) [0..1]`.

Using `push @Vec` we can thus write the following to duplicate the head element of a vector:

```
copyHead : ∀ { a : Type, n : Nat }
         . (Vec (n + 1) a) [0..2] → (a, a)
copyHead = copy . head . push@(→) [push @Vec] . disject
```

which employs the combinator `disject` from the standard library and two derived distributive laws, of type:

```
disject   : ∀ { a : Type, s : Semiring, n m : s }
          . a [m * n] → (a [n]) [m]
push @Vec : ∀ { a : Type, n : Nat, s : Semiring, r : s }
          . (Vec n a) [r] → Vec n (a [r])
push@(→) : ∀ { a b : Type, s : Semiring, r : s }
          . (a  → b) [r] → a [r] → b [r]
```

5.5 DERIVING OTHER USEFUL STRUCTURAL COMBINATORS

So far we have motivated the use of distributive laws, and demonstrated that they are useful in practice when programming in languages with linear and graded modal types. The same methodology we have been discussing can also be used to derive other useful generic combinators for programming with linear and graded modal types. In this section, we consider two structural combinators, `drop`

$$
\begin{aligned}
[\![C^{\mathsf{w}}]\!]^{\Sigma}_{\mathsf{drop}} z &= \mathsf{drop}^{C^{\mathsf{w}}}\ z \\
[\![\mathsf{Unit}]\!]^{\Sigma}_{\mathsf{drop}} z &= \mathbf{case}\ z\ \mathbf{of}\ () \to () \\
[\![X]\!]^{\Sigma}_{\mathsf{drop}} z &= \Sigma(X) z \\
[\![A \oplus B]\!]^{\Sigma}_{\mathsf{drop}} z &= \mathbf{case}\ z\ \mathbf{of}\ \ \mathsf{inl}\ x \to [\![A]\!]_{\mathsf{drop}}(x);\ \ \mathsf{inr}\ y \to [\![B]\!]_{\mathsf{drop}}(y) \\
[\![A \otimes B]\!]^{\Sigma}_{\mathsf{drop}} z &= \mathbf{case}\ z\ \mathbf{of}\ (x, y) \to \\
&\qquad \mathbf{case}\ [\![A]\!]_{\mathsf{drop}}(x)\ \mathbf{of}\ () \to \\
&\qquad\quad \mathbf{case}\ [\![B]\!]_{\mathsf{drop}}(y)\ \mathbf{of}\ () \to () \\
[\![\mu X.A]\!]^{\Sigma}_{\mathsf{drop}} z &= \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : A \multimap 1}_{\mathsf{drop}}\ \mathbf{in}\ f\ z
\end{aligned}
$$

Figure 5.5: Interpretation rules for $[\![A]\!]_{\mathsf{drop}}$

and `copyShape`, in Granule as well as related type classes for dropping, copying, and moving resources in Linear Haskell.

5.5.1 *A Combinator for Weakening ("drop")*

First, we consider a combinator for "dropping," or consuming, values in Granule:

$$[\![\mathsf{F}\alpha]\!]_{\mathsf{drop}} : \mathsf{F}\alpha \multimap \mathsf{FUnit}$$

The built-in type constants of Granule can be split into those which permit structural weakening C^{w} such as `Int`, `Char`, and those which do not C^{l} such as `Handle` (file handles) and `Chan` (concurrent channels).

Those that permit weakening contain non-abstract values that can in theory be systematically inspected in order to consume them. Granule provides a built-in implementation of `drop` for C^{w} types, which is then used by the derivation procedure of Figure 5.5 to derive weakening on compound types.

Note we cannot use this procedure in a polymorphic context (over type variables α) since type polymorphism ranges over all types, including those which cannot be dropped like C^{l}.

5.5.2 *A Combinator for Copying "shape"*

The "shape" of values for a parametric data types F can be determined by a function *shape* : $\mathsf{F}A \to \mathsf{FUnit}$, usually derived when F is a functor by lifting a function $A \to \mathsf{Unit}$ (dropping elements) [Jay and Cockett, 1994]. This provides a way of capturing the size, shape, and form of a data structure. Often when programming with data structures which must be used linearly, we may wish to reason about properties of the data structure (such as the length or "shape" of the structure) but we

may not be able to drop the contained values. Instead, we wish to extract the shape but without consuming the original data structure itself. For example, say we have some list of values and wish to know its length. Our `length` function may take a linear list as an argument and consumes it, giving us back an integer. But what if we are not yet ready to give up our list?

This can be accomplished with a function which copies the data structure exactly, returning this duplicate along with a data structure of the same shape, but with the terminal nodes replaced with values of the unit type Unit (the 'spine'). For example, consider a pair of integers: `(1, 2)`. Then applying `copyShape` to this pair would yield `(((), ()), (1, 2))`. The original input pair is duplicated and returned on the right of the pair, while the left value contains a pair with the same structure as the input, but with values replaced with `()`. This is useful, as it allows us to use the left value of the resulting pair to reason about the structure of the input (e.g., its depth / size), while preserving the original input. This is particularly useful for deriving size and length combinators for collection-like data structures. As with "drop", we can derive such a function automatically:

$$[\![\mathsf{F}\alpha]\!]_{\mathsf{copyShape}} : \mathsf{F}\alpha \multimap \mathsf{FUnit} \otimes \mathsf{F}\alpha$$

defined by $[\![A]\!]_{\mathsf{copyShape}} = \lambda z.[\![A]\!]^{\emptyset}_{\mathsf{copyShape}} z$ by an intermediate interpretation $[\![A]\!]^{\Sigma}_{\mathsf{copyShape}}$, given by Figure 5.6. The implementation recursively follows the structure of the type, replicating the constructors, reaching the crucial case where a polymorphically type $z : \alpha$ is mapped to $((), z)$ in the third equation.

5.5.3 *Implementation in Granule*

Granule implements both these derived combinators in a similar way to *push/pull* providing `copyShape` and `drop` which can be derived for a type `T` via type application, e.g. `drop @T : T → ()` if it can be derived. Otherwise, the type checker produces an error, explaining why `drop` is not derivable at type `T`. As an example of using `copyShape` in Granule, consider the case where we want to find the length of a list, without consuming the list itself. Without `copyShape`, we would have to write our `length` function so that it returns the input list after computing its size, resulting in the rather cumbersome implementation:

```
data List = Cons a | Nil

data N = S N | Z

length : ∀ { a : Type } . List a → (N, List a)
length Nil = (Z, Nil);
length (Cons x xs) =
  let (n, xs') = length xs in (S n, Cons x xs')
```

$$
\begin{aligned}
[\![C^w]\!]^{\Sigma}_{\mathsf{copyShape}} z &= ((),\ z) \\
[\![\mathsf{Unit}]\!]^{\Sigma}_{\mathsf{copyShape}} z &= \mathbf{case}\ z\ \mathbf{of}\ () \to ((),\ ()) \\
[\![\alpha]\!]^{\Sigma}_{\mathsf{copyShape}} z &= ((), z) \\
[\![X]\!]^{\Sigma}_{\mathsf{copyShape}} z &= \Sigma(X) z \\
[\![A \oplus B]\!]^{\Sigma}_{\mathsf{copyShape}} z &= \mathbf{case}\ z\ \mathbf{of} \\
&\qquad \mathsf{inl}\ x \to \mathbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{copyShape}}(x)\ \mathbf{of}\ (s,\ x') \to \\
&\qquad\qquad (\mathsf{inl}\ s,\ \mathsf{inl}\ x') \\
&\qquad \mathsf{inr}\ y \to \mathbf{case}\ [\![B]\!]^{\Sigma}_{\mathsf{copyShape}}(y)\ \mathbf{of}\ (s,\ y') \to \\
&\qquad\qquad (\mathsf{inr}\ s,\ \mathsf{inr}\ y') \\
[\![A \otimes B]\!]^{\Sigma}_{\mathsf{copyShape}} z &= \mathbf{case}\ z\ \mathbf{of}\ (x, y) \to \\
&\qquad \mathbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{copyShape}}(x)\ \mathbf{of}\ (s,\ x') \to \\
&\qquad\qquad \mathbf{case}\ [\![B]\!]^{\Sigma}_{\mathsf{copyShape}}(y)\ \mathbf{of} \\
&\qquad\qquad\qquad (s',\ y') \to ((s,\ s'),\ (x',\ y')) \\
[\![\mu X.A]\!]^{\Sigma}_{\mathsf{copyShape}} z &= \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : A \multimap 1 \otimes A}_{\mathsf{copyShape}}\ \mathbf{in}\ f\ z
\end{aligned}
$$

Figure 5.6: Interpretation rules for $[\![A]\!]_{\mathsf{copyShape}}$

Using `copyShape`, we can rewrite this definition of `length` into a simpler form which doesn't have to worry about how it consumes its input:

```
length : ∀ { a : Type } List a → N
length Nil = Z
length (Cons _ xs) = S (length xs)
```

and then to call `length` on a list, we instead pass in the shape of the list rather than the list itself:

```
let (shape, myList') = copyShape@List myList in length shape
```

5.6 CONCLUSION

The work described here addresses the practical aspects of applying these techniques in real-world programming. Our hope is that this aids the development of the next generation of programming languages with rich type systems for high-assurance programming.

In the next chapter, we collect and describe the related work for various aspects of this thesis. Following this, we conclude our synthesis journey by describing some future directions that this work might take.

6

RELATED WORK

This chapter collects various branches of related work from previous chapters. We begin in Section 6.1 with a brief overview of other approaches to the problem of resource non-determinism (or the "resource management problem") in the context of linear logic proof search, an issue which was discussed at length in Chapter 3. Having established this background, Section 6.2 offers a comparison between the approach of this thesis and another recent work which deals with the notion of resources in synthesis: the ReSyn tool of Knoth et al. [2019]. Finally, we conclude this chapter with Section 6.3, an overview of the work that relates to Chapter 5 and our approach to generic programming for graded types.

6.1 COMPARISON OF RESOURCE MANAGEMENT APPROACHES

This section discusses alternative approaches to the issue of resource non-determinism in synthesis, which we discussed in Section 3.2.

Before the work of Hodas and Miller [1994], the problem of resource non-determinism was first identified by Harland and Pym [2000]. Their solution delays splitting of contexts at a multiplicative connective. They proposed a solution where proof search is formulated in terms of constraints on propositions. The logic programming language Lygon [lyg] implements this approach.

Our approach to synthesis implements a *backward* style of proof search: starting from the goal, recursively search for solutions to sub-goals. In contrast to this, *forward* reasoning approaches attempt to reach the goal by building sub-goals from previously proved sub-goals until the overall goal is proved. Chaudhuri and Pfenning [2005a,b] consider forward approaches to proof search in linear logic using the *inverse method* [Degtyarev and Voronkov, 2001] where the resource non-determinism that is typical to backward approaches is absent.

6.2 COMPARISON WITH RESYN

Knoth et al. [2019]'s work introduces a programming language and synthesis tool named ReSyn which uses resource constraints obtained

through automated amortised resource analysis (AARA) [Çiçek et al., 2017, Hoffmann et al., 2012, Wang et al., 2017] and Liquid refinement types [Rondon et al., 2008] to guide the synthesis process, producing a program which adheres to these constraints. Although both this approach and our own both relate to the static tracking of resources, the type system of ReSyn and Granule differ in their notion of resources.

In ReSyn, a syntactic construct called 'tick' represents the consumption of a resource. which can be used to track several properties such as asymptotic time complexity, memory usage, and execution time. For example, 'tick' may be applied to each recursive call in a function, allowing the programmer to provide a static bound on the number of recursive calls that a program makes.

This AARA approach is combined with Liquid types to support verification of non-trivial functional properties of a program. Types are annotated with a natural number (termed *potential*) which provides an upper bound on resource usage. Knoth et al. [2019] draw a comparison between this notion of potential and Bounded Linear Logic [Girard et al., 1992]. In Granule, one might express potential through the interval over the extended natural numbers semiring. Liquid types also allow ReSyn to express resource bounds which are dependent on a program's values, using conditional linear arithmetic (CLIA) to represent potential instead of constant values. The type checker then ensures that a program has enough potential to fulfil the requirements provided by the type

Whilst cost annotation in ReSyn provides its notion of resources, the relationship between this approach and the general semiring-graded account of resources considered is not clear. In fact, we believe it is largely unknown how to relate a 'tick-based' cost model with a semiring graded necessity (although some work in this direction exists by Çiçek et al. [2016]).

However, despite these differences in type system, there is still a comparison to be made between how ReSyn and our own work handle program synthesis. Synthesis in ReSyn must still address the issue of resource management when constructing programs which adhere to some potential. Similarly to this work, a synthesis context in ReSyn contains a set of free variables available for use in synthesis with some 'potential' annotation. When synthesising syntactic constructs with multiple sub-terms the ReSyn is required to distribute the available resources between the different branches of synthesis. In ReSyn, when a context Γ requires sharing between Γ_1 and Γ_2 such that $\Gamma_1 + \Gamma_2 = \Gamma$, ReSyn needs to guess what potential annotations are needed such that synthesis of both sub-terms has the necessary potential. In contrast to the algorithmic resource schemes we implement in Section 3.2, ReSyn uses a constraint-based approach, which shares a greater familiarity with the approach of Harland and Pym [2000]. We elide the details of

their constraint generation and solving here, and encourage interested readers to study their paper.

6.3 GENERIC DERIVING METHODOLOGY AND GRADED DISTRIBUTIVE LAWS

In this section we consider some of the wider related work as they relate to the ideas presented in Chapter 5.

6.3.1 *Generic Programming Methodology*

The deriving mechanism for Granule is based on the methodology of generic functional programming [Hinze, 2000], where functions may be defined generically for all possible data types in the language; generic functions are defined inductively on the structure of the types. This technique has notably been used before in Haskell, where there has been a strong interest in deriving type class instances automatically. Particularly relevant to this work is the work on generic deriving [Magalhães et al., 2010], which allows Haskell programmers to automatically derive arbitrary class instances using standard datatype-generic programming techniques as described above.

6.3.2 *Non-graded Distributive Laws*

Distributive laws are standard components in abstract mathematics. Distributive laws between categorical structures used for modelling modalities (like monads and comonads) are well explored. For example, Brookes and Stone [1993] defined a categorical semantics using monads combined with comonads via a distributive law capturing both intensional and effectful aspects of a program. Power and Watanabe [2002] study in detail different ways of combining comonads and monads via distributive laws. Such distributive laws have been applied in the programming languages literature, e.g., for modelling streams of partial elements [Uustalu and Vene, November 2006].

6.3.3 *Graded Distributive Laws*

Gaboardi et al. [2016] define families of graded distributive laws for graded monads and comonads. They include the ability to interact the grades, e.g., with operations such as $\Box_{\iota(r,f)} \Diamond_f A \to \Diamond_{\kappa(r,f)} \Box_r A$ between a graded comonad $\Box_r$ and graded monad $\Diamond_f$ where ι and κ capture information about the distributive law in the grades. In comparison, our distributive laws here are more prosaic since they involve only a graded comonad (semiring graded necessity) distributed over a func-

tor and vice versa. That said, the scheme of Gaboardi et al. suggests that there might be interesting graded distributive laws between $\Box_r$ and the indexed types, for example, $\Box_r(\mathsf{Vec}\, n\, A) \to \mathsf{Vec}\,(r * n)\,(\Box_1 A)$ which internally replicates a vector. However, it is less clear how useful such combinators would be in general or how systematic their construction would be. In contrast, the distributive laws explained here appear frequently and have a straightforward uniform calculation.

We noted in Section 5.3 that neither of our distributive laws can be derived over graded modalities themselves, i.e., we cannot derive $push : \Box_r \Box_s A \to \Box_s \Box_r A$. Such an operation would itself be a distributive law between two graded modalities, which may have further semantic and analysis consequences beyond the normal derivations here for regular types. Exploring this is future work, for which the previous work on graded distributive laws can provide a useful scheme for considering the possibilities here. Furthermore, Granule has both graded comonads and graded monads so there is scope for exploring possible graded distributive laws between these in the future following Gaboardi et al. [2016].

6.3.4 *Typed Analysis of Consumption in Pattern Matching*

This chapter's study of distributive laws provides an opportunity to consider design decisions for the *typed analysis of pattern matching* since the operations of Section 5.3 are derived by pattern matching in concert with grading. We compare here the choices made surrounding the typing of pattern matching in four works (1) Granule and its core calculus [Orchard et al., 2019] (2) the graded modal calculus Λ^p of Abel and Bernardy [2020] (3) the dependent graded system GRAD of Choudhury et al. [2021] and (4) Linear Haskell [Bernardy et al., 2018].

GRANULE Pattern matching against a graded modality $\Box_r A$ (with pattern $[p]$) in Granule is provided by the PBOX rule (Figure 5.2) which triggers typing pattern p 'under' a grade r at type A. This was denoted via the optional grade information $r \vdash p : A \rhd \Gamma$ which then pushes grading down onto the variables bound within p. Furthermore, it is only under such a pattern that wildcard patterns are allowed ([PWILD]), requiring $0 \sqsubseteq r$, i.e., r can approximate 0 (where 0 denotes weakening). None of the other systems considered here have such a facility for weakening via pattern matching.

For a type A with more than one constructor ($|A| > 1$), pattern matching its constructors underneath an r-graded box requires $1 \sqsubseteq r$. For example, eliminating sums inside an r-graded box $\Box_r(A \oplus B)$ requires $1 \sqsubseteq r$ as distinguishing inl or inr constitutes a *consumption* which reveals information (i.e., pattern matching on the 'tag' of the

data constructors). By contrast, a type with only one constructor cannot convey any information by its constructor and so matching on it is not counted as a consumption: eliminating $\Box_r(A \otimes B)$ places no requirements on r. The idea that unary data types do not incur consumption (since no information is conveyed by its constructor) is a refinement here to the original Granule paper as described by Orchard et al. [2019], which for [PCON] had only the premise $1 \sqsubseteq r$ rather than $|A| > 1 \implies 1 \sqsubseteq r$ here (although the implementation already reflected this idea).

THE Λ^p CALCULUS Abel and Bernardy's unified modal system Λ^p is akin to Granule, but with pervasive grading (rather than base linearity) akin to the coeffect calculus [Petricek et al., 2014] and Linear Haskell [Bernardy et al., 2018]. Similarly to the situation in Granule, Λ^p also places a grading requirement when pattern matching on a sum type, given by the following typing rule in their syntax [Abel and Bernardy, 2020, Fig 1, p.4]:

$$\frac{\gamma\Gamma \vdash t : A_1 + A_2 \qquad \delta\Gamma, x_i :^q A_i \vdash u_i : C \qquad q \leq 1}{(q\gamma + \delta)\Gamma \vdash \mathsf{case}^q\ t\ \mathsf{of}\ \{\mathsf{inj}_1 x_1 \mapsto u_1; \mathsf{inj}_2 x_2 \mapsto u_2\} : C}\ +\text{-ELIM}$$

The key aspects here are that variables x_i bound in the case are used with grade q as denoted by the graded assumption $x_i :^q A_i$ in the context of typing u_i and then that $q \leq 1$ which is exactly our constraint that $1 \sqsubseteq r$ (their ordering just runs in the opposite direction to ours). For the elimination of pair and unit types in Λ^p there is no such constraint, matching our idea that arity affects usage, captured in Granule by $|A| > 1 \implies 1 \sqsubseteq r$. Their typed-analysis of patterns is motivated by their parametricity theorems.

GRAD The dependent graded type system GRAD of Choudhury et al. also considers the question of how to give the typing of pattern matching on sum types, with a rule in their system [Choudhury et al., 2021, p.8] which closely resembles the $+$-ELIM rule for Λ^p:

$$\frac{\Delta;\Gamma_1 \vdash q : A_1 \oplus A_2 \qquad \Delta;\Gamma_2 \vdash b_1 : A_1 \xrightarrow{q} B \qquad \Delta;\Gamma_2 \vdash b_2 : A_2 \xrightarrow{q} B \qquad 1 \leq q}{\Delta; q \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_q\ a\ \mathbf{of}\ b_1; b_2 : B}\ \text{STCASE}$$

The direction of the preordering in GRAD is the same as that in Granule but, modulo this ordering and some slight restructuring, the case rule captures the same idea as Λ^p: "both branches of the base analysis *must* use the scrutinee at least once, as indicated by the $1 \leq q$ constraint." [Choudhury et al., 2021, p.8]. Choudhury et al., also provide a heap-based semantics which serves to connect the meaning of grades with a concrete operational model of usage, which then motivates the grading on sum elimination here. In the simply-typed

version of GRAD, matching on the components of a product requires that each component is consumed linearly.

LINEAR HASKELL The paper on Linear Haskell by Bernardy et al. [2018] has a **case** expression for eliminating arbitrary data constructors, with grading similar to the rules seen above. Initially, this rule is for the setting of a semiring over $\mathcal{R} = \{1, \omega\}$ and has no requirements on the grading to represent the notion of inspection, consumption, or usage due to matching on (multiple) constructors. This is reflected in the current implementation where we can define the following sum elimination:

```
match :: (Either a b) %r -> (a %1 -> c) -> (b %1 -> c) -> c
match (Left x) f _  = f x
match (Right x) _ g = g x
```

However, later when considering the generalisation to other semirings they state that "*typing rules are mostly unchanged with the caveat that* case_π *must exclude* $\pi = 0$" [Bernardy et al., 2018] where π is the grade of the **case** guard. This appears a more coarse-grained restriction than the other three systems, excluding even the possibility of Granule's weakening wildcard pattern which requires $0 \leq \pi$. Currently, such a pattern must be marked as `'Many` in Linear Haskell (i.e., it cannot explain that first projection on a pair does not use the pair's second component). Furthermore, the condition $\pi \neq 0$ does not require that π actually represents a consumption, unlike the approaches of the other three systems. The argument put forward by Abel and Bernardy for their restriction to mark a consumption ($q \leq 1$) for the sake of parametricity is a compelling one, and the concrete model of Choudhury et al. gives further confidence that this restriction captures well an operational model. Thus, it seems there is a chance for fertilisation between the works mentioned here and Linear Haskell's vital work, towards a future where grading is a key tool in the typed-functional programmer's toolbox.

7

CONCLUSION

In this dissertation, we have provided a framework for designing program synthesis tools based on linear and graded type systems, overcoming the challenges imposed by treating program values as resources, and leveraging the type systems' properties to build two efficient synthesis tools, targeting two common flavours of graded type system: the linear-base style and the graded-base style.

In our thesis statement in Chapter 1, we claimed that this work would demonstrate how linear and graded types could be integrated into a synthesis tool. The synthesis calculi we have developed evidence this claim, and we have also demonstrated their efficiency with respect to the search space of programs in our evaluation in Chapter 4.

In summary, in our work we have succeeded in:

- Building several core calculi for two varieties of graded type systems, encompassing the major approaches in the literature, with both styles differing in expressivity and imposing unique requirements.
- Implementing these synthesis calculi as tools for Granule, as fully integrated components of the Granule toolchain.
- Evaluating each of these systems in a variety of criteria, our main result being that synthesis with graded types is an effective way of reducing the program synthesis search space.
- Showing that programs require fewer examples than in a purely example-driven synthesis setting, such as MYTH.
- Providing an alternative approach to generating graded combinators, based on a generic programming methodology.
- Adapting our synthesis technique to Haskell, showing the viability of our framework as a foundation for building synthesis tools for other graded type systems in the real world.
- Proved the soundness of each of these systems, reasoning about the behavioural properties of our synthesis calculi.

Essentially, we have shown that program synthesis from linear and graded types is a feasible and effective approach to reducing the search space of a program synthesis task. The expressivity of linear and graded types in describing the static semantics of a program offers many benefits to programmers, so harnessing this expressivity in synthesis is a something that we find will be useful to programs in this context, allowing the synthesis of programs often without the need for additional specification such as examples.

There remain many avenues for future exploration, which we intend to pursue, and hope that others will also find use in this approach when designing synthesis tools for their own type systems.

7.1 FUTURE DIRECTIONS

Our goal was to demonstrate the viability of a resource-aware type-directed program synthesis tool which assists the programmer in writing programs with resource-sensitive types. We intend to pursue further improvements to our tool which serve this end, including reducing the overhead of SMT solving, integrating examples into the search algorithm itself in the style of Myth [Osera and Zdancewic, 2015] and Leon [Kneuss et al., 2013], as well as considering possible semiring-dependent optimisations that may be applicable.

7.1.1 *SMT Solving*

Related to the above, there is scope for improving the interaction between the synthesis tool and the SMT solver, to make synthesis more efficient. Both evaluations showed that the bulk of synthesis time is spent solving constraints in the SMT solver. Making this interaction more streamlined is therefore particularly appealing.

Alternatively, one could imagine implementing custom solvers for the more commonly used semirings. From our lists of benchmarking examples in Tables 3.3 and 4.2, we can see the natural number, and intervals over the natural numbers semirings appear frequently, more than any other semiring. An SMT solver which focuses purely on solving in these semirings, sacrificing the generality provided by solvers such as Z3 [de Moura and Bjørner, 2008], may be a worthwhile avenue of exploration. This approach would also avoid the overhead of serialising constraints to SMT-LIB format and starting up Z3.

7.1.2 *Generalised Algebraic Data Types*

A logical next step is to incorporate GADTs (Generalised ADTs), i.e., indexed types, into the synthesis algorithm. Granule provides support for user-defined GADTs, and the interaction between grades and type

indices is a key contributor to Granule's expressive power [Orchard et al., 2019]. Consider our list type benchmarks for example. In most cases, when we want to synthesise a recursive function definition which takes a list as input, we have to give the list a 0..∞ interval grade to account for potentially unlimited usage. Take for example a program that replicates a value some number of times to create a list typed:

```
rep : ∀ { a : Type } . Int → a % 0..∞ → List a
spec
    rep % 0..∞
rep = ?
```

Given a standard indexed type of natural numbers, defined:

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

and sized-indexed vectors:

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

a refined `rep` function can be given a much tighter specification, connecting the usage of the input function to the length of the vector, from which we could synthesise the program:

```
vrep : ∀ { n : Nat, a : Type } . N n → a % n → Vec n a
spec
    vrep % n
vrep Z c = Nil;
vrep (S n) c = Cons c (vrep n c)
```

The latter type not only provides us with a greater opportunity to prune grade-violating programs, its type is also much more descriptive of the user's intent. Adapting our approach to GADTs is future work, and mostly consists of extending the typing for our synthesis rule for **case** statements to handle GADT specialisation.

GADTs are sometimes referred to as *lightweight* dependent types. Dependent type systems allow arbitrary program properties to be expressed and verified during type checking, with types being indexed by arbitrary program terms. Several works have endeavoured to integrate fully-dependent types into a quantitative setting [Choudhury et al., 2021, Abel et al., 2023, McBride, 2016, Krishnaswami et al., 2015]. One such example, Moon et al. [2021]'s GERTY is based on Granule and has a prototype implementation, making it a particularly appealing target for experimenting with the synthesis in a fully-dependent graded setting, using the foundations laid by the work in this thesis. Another is the Idris 2 programming language [Brady, 2021], which implements dependent types in a quantitative setting, and is an example of such resourceful systems being used in a practical system.

7.1.3 *Ownership-Based Type Systems*

Linearity is closely related to the notions of *uniqueness* [Barendsen and Smetsers, 1993, Smetsers et al., 1994], as well as *ownership* and *borrowing* [Mycroft and Voigt, 2013]. The programming language Rust uses a system based on these ideas [Matsakis and Klock, 2014, Jung et al., 2017, 2019], providing memory safety guarantees through the "borrow checker", which ensures that only one owner of a value can write to its memory location at any time. Multiple reads are permitted, however, via *borrowing*. Marshall et al. [2022] give a type theoretic view of Rust's borrow checker, which Rust treats as a separate entity from the type checker. This view opens the gateway for type-directed program synthesis, where techniques we provide for synthesis in this work can be generalised to incorporate Rust-like approaches to resourceful types.

7.1.4 *Large Language Models*

With the rise in Large Language Models (LLMs) showing their power at program synthesis tasks [Austin et al., 2021, Jain et al., 2021], the deductive approach still has something to contribute: it provides correct-by-construction synthesis based on specifications, rather than predicted programs which may violate more fine-grained type constraints (e.g., as provided by grades). Future approaches may combine both LLM approaches with deductive approaches, where the logical engine of the deductive approach can guide prediction, e.g. by being used for hyperparameter tuning. Exploring this is further work and a general opportunity and challenge for the synthesis community.

7.2 FINAL REMARKS

Type-directed program synthesis has long been an attractive field in computer science partially due to the potential it offers: the ability to write programs that are correct by construction, with significant help from the computer.

We feel that type-directed program synthesis complements linear and graded types very well. As richer types further constrain the number of possible inhabitants, theoretically there is less work to be done by the computer to identify the program which behaves according to the user's intent. A potential future extension to this work is to use this resourceful information to help inform users why their desired program was not able to be synthesised.

Programming with linear and graded types can require significantly more forethought than standard functional programming, and programs which seem correct at first glance to a user might actually

be resource-violating. Our hope is that the tools developed in this work may be useful in reducing this cognitive overhead, and that the ideas we have developed may be useful to those in both the program synthesis, and the quantitative types communities.

BIBLIOGRAPHY

Logic programming with linear logic. URL http://www.cs.rmit.edu.au/lygon/. Accessed 19th June 2020.

Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proc. ACM Program. Lang.*, 4(ICFP):90:1–90:28, 2020. doi: 10.1145/3408972. URL https://doi.org/10.1145/3408972.

Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. A graded modal dependent type theory with a universe and erasure, formalized. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023. doi: 10.1145/3607862. URL https://doi.org/10.1145/3607862.

Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1):3–57, 1993. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(93)90181-R. URL https://www.sciencedirect.com/science/article/pii/030439759390181R.

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 934–950, 2013. doi: 10.1007/978-3-642-39799-8_67. URL https://doi.org/10.1007/978-3-642-39799-8_67.

Guillaume Allais. Typing with Leftovers-A mechanization of Intuitionistic Multiplicative-Additive Linear Logic. In *23rd International Conference on Types for Proofs and Programs (TYPES 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 06 1992. ISSN 0955-792X. doi: 10.1093/logcom/2.3.297.

Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65, 2018. doi: 10.1145/3209108.3209189. URL https://doi.org/10.1145/3209108.3209189.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In Rudrapatna K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 41–51, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-540-48211-6.

Nick Benton, Gavin Bierman, Valeria De Paiva, and Martin Hyland. Linear lambda-calculus and categorical models revisited. In *International Workshop on Computer Science Logic*, pages 61–84. Springer, 1992. doi: 10.1007/3-540-56992-8_6.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, 2018. doi: 10.1145/3158093. URL https://doi.org/10.1145/3158093.

Edwin C. Brady. Idris 2: Quantitative type theory in practice. *CoRR*, abs/2104.00480, 2021. URL https://arxiv.org/abs/2104.00480.

Stephen Brookes and Kathryn V Stone. Monads and comonads in intensional semantics. Technical report, Pittsburgh, PA, USA, 1993. URL https://dl.acm.org/doi/10.5555/865105.

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer, 2014. doi: 10.1007/978-3-642-54833-8_19.

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 316–329, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346603. doi: 10.1145/3009837.3009858. URL https://doi.org/10.1145/3009837.3009858.

Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1):133 – 163, 2000. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(99)00173-5.

Kaustuv Chaudhuri and Frank Pfenning. A Focusing Inverse Method Theorem Prover for First-Order Linear Logic. In *Proceedings of the 20th International Conference on Automated Deduction*, CADE' 20, page 69–83, Berlin, Heidelberg, 2005a. Springer-Verlag. ISBN 3540280057. doi: 10.1007/11532231_6.

Kaustuv Chaudhuri and Frank Pfenning. Focusing the Inverse Method for Linear Logic. In *Proceedings of the 19th International Conference on Computer Science Logic*, CSL'05, page 200–215, Berlin, Heidelberg, 2005b. Springer-Verlag. ISBN 3540282319. doi: 10.1007/11538363_15.

Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. A graded dependent type system with a usage-aware semantics. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021. doi: 10.1145/3434331. URL https://doi.org/10.1145/3434331.

Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. A type theory for incremental computational complexity with control flow changes. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 132–145. ACM, 2016. doi: 10.1145/2951913.2951950. URL https://doi.org/10.1145/2951913.2951950.

Harry Clarke, Vilem-Benjamin Liepelt, and Dominic Orchard. Scrap your reprinter. 2017.

Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient smt solver. volume 4963, pages 337–340, 04 2008.

Anatoli Degtyarev and Andrei Voronkov. Chapter 4 - the inverse method. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 179 – 272. North-Holland, Amsterdam, 2001. ISBN 978-0-444-50813-3. doi: https://doi.org/10.1016/B978-044450813-3/50006-0.

Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. Visible type application. In *European Symposium on Programming*, pages 229–254. Springer, 2016. doi: 10.1007/978-3-662-49498-1_10.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 229–239, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737977. URL https://doi.org/10.1145/2737924.2737977.

Jonas Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. Leveraging Rust Types for Program Synthesis. *To appear in the Proceedings of PLDI*, 2023.

Jonathan Frankle, Peter-Michael Osera, David Walker, and S. Zdancewic. Example-directed synthesis: a type-theoretic in-

terpretation. *ACM SIGPLAN Notices*, 51:802–815, 01 2016a. doi: 10.1145/2914770.2837629.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. *ACM SIGPLAN Notices*, 51(1):802–815, 2016b.

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 357–370, 2013. doi: 10.1145/2429069.2429113.

Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. Combining effects and coeffects via grading. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 476–489. ACM, 2016. doi: 10.1145/2951913.2951939.

Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014*, volume 8410 of *Lecture Notes in Computer Science*, pages 331–350. Springer, 2014. doi: 10.1007/978-3-642-54833-8_18.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1): 1 – 101, 1987. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(87)90045-4.

Jean-Yves Girard, Andre Scedrov, and Philip J Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical computer science*, 97(1):1–66, 1992. doi: 10.1016/0304-3975(92)90386-T.

Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, page 219–239, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. volume 46, pages 317–330, 01 2011. doi: 10.1145/1926385.1926423.

James Harland and David J. Pym. Resource-distribution via boolean constraints. *CoRR*, cs.LO/0012018, 2000. URL https://arxiv.org/abs/cs/0012018.

Ralf Hinze. A new approach to generic functional programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on*

Principles of programming languages, pages 119–132, 2000. doi: 10.1145/325694.325709.

J.S. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327 – 365, 1994. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.1994.1036.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ml. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 781–786, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31424-7.

Jack Hughes and Dominic Orchard. Resourceful program synthesis from graded linear types. In *Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings*, pages 151–170, 2020. doi: 10.1007/978-3-030-68446-4_8. URL https://doi.org/10.1007/978-3-030-68446-4_8.

Jack Hughes and Dominic Orchard. Program synthesis from graded types. 2024. Conditionally accepted to ESOP 2024.

Jack Hughes, Michael Vollmer, and Dominic Orchard. Deriving distributive laws for graded linear types. In Ugo Dal Lago and Valeria de Paiva, editors, *Proceedings Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity&TLLA@IJCAR-FSCD 2020, Online, 29-30 June 2020*, volume 353 of *EPTCS*, pages 109–131, 2020. doi: 10.4204/EPTCS.353.6. URL https://doi.org/10.4204/EPTCS.353.6.

Jack Hughes, Daniel 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.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis, 2021.

C Barry Jay and J Robin B Cockett. Shapely types and shape polymorphism. In *European Symposium on Programming*, pages 302–316. Springer, 1994. doi: 10.1007/3-540-57880-3_20.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi: 10.1145/3158154. URL https://doi.org/10.1145/3158154.

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for rust. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.

Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 633–646. ACM, 2014. doi: 10.1145/2535838.2535846.

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *SIGPLAN Not.*, 40(9):192–203, September 2005. ISSN 0362-1340. doi: 10.1145/1090189.1086390.

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. OOPSLA '13, page 407–426, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323741. doi: 10.1145/2509136.2509555. URL https://doi.org/10.1145/2509136.2509555.

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-Guided Program Synthesis. *CoRR*, abs/1904.07415, 2019. URL http://arxiv.org/abs/1904.07415.

Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 17–30, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2676969. URL https://doi.org/10.1145/2676726.2676969.

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. *SIGPLAN Not.*, 45(6):316–329, jun 2010. ISSN 0362-1340. doi: 10.1145/1809028.1806632. URL https://doi.org/10.1145/1809028.1806632.

Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. *Logical Methods in Computer Science*, Volume 8, Issue 4, October 2012. ISSN 1860-5974. doi: 10.2168/lmcs-8(4:11)2012. URL http://dx.doi.org/10.2168/LMCS-8(4:11)2012.

Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410 (46):4747–4768, 2009.

Vilem-Benjamin Liepelt, Dominic Orchard, and Daniel Marshall. A tale of two graded calculi: The marriage of coeffects and graded comonads. Unpublished, 2024.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A Generic Deriving Mechanism for Haskell. *SIGPLAN Not.*, 45 (11):37–48, September 2010. ISSN 0362-1340. doi: 10.1145/2088456.1863529. URL https://doi.org/10.1145/2088456.1863529.

Zohar Manna and Richard Waldinger. Synthesis: Dreams → programs. *Software Engineering, IEEE Transactions on*, SE-5:294– 328, 08 1979. doi: 10.1109/TSE.1979.234198.

Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

Daniel Marshall, Michael Vollmer, and Dominic Orchard. Linearity and uniqueness: An entente cordiale. In Ilya Sergey, editor, *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*, volume 13240 of *Lecture Notes in Computer Science*, pages 346–375. Springer, 2022. doi: 10.1007/978-3-030-99336-8_13. URL https://doi.org/10.1007/978-3-030-99336-8_13.

Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34(3):103–104, oct 2014. ISSN 1094-3641. doi: 10.1145/2692956.2663188. URL https://doi.org/10.1145/2692956.2663188.

Conor McBride. *I Got Plenty o' Nuttin'*, pages 207–233. Springer International Publishing, Cham, 2016. ISBN 978-3-319-30936-1. doi: 10.1007/978-3-319-30936-1_12.

Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

Benjamin Moon, Harley Eades III, and Dominic Orchard. Graded Modal Dependent Type Theory. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 462–490. Springer, 2021. doi: 10.1007/978-3-030-72019-3_17. URL https://doi.org/10.1007/978-3-030-72019-3_17.

Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20:629–679, 1994. ISSN 0743-1066. doi: https://doi.org/10.1016/0743-1066(94)90035-3. URL https://www.sciencedirect.com/science/article/pii/0743106694900353. Special Issue: Ten Years of Logic Programming.

Alan Mycroft and Janina Voigt. *Notions of Aliasing and Ownership*, pages 59–83. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36946-9. doi: 10.1007/978-3-642-36946-9_4. URL https://doi.org/10.1007/978-3-642-36946-9_4.

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *PACMPL*, 3 (ICFP):110:1–110:30, 2019. doi: 10.1145/3341714.

Dominic A. Orchard, Tomas Petricek, and Alan Mycroft. The semantic marriage of monads and effects. *CoRR*, abs/1401.5391, 2014. URL http://arxiv.org/abs/1401.5391.

Peter-Michael Osera. Constraint-based type-directed program synthesis, 2019.

Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *SIGPLAN Not.*, 50(6):619–630, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2738007.

Peter-Michael Santos Osera. Program synthesis with types, 2015.

Tomas Petricek. *Context-aware programming languages*. PhD thesis, University of Cambridge, 3 2017.

Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: Unified static analysis of context-dependence. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, pages 385–397, 2013. doi: 10.1007/978-3-642-39212-2_35. URL https://doi.org/10.1007/978-3-642-39212-2_35.

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*, pages 123–135. ACM, 2014. doi: 10.1145/2692915.2628160.

Frank Pfenning. Lecture notes in linear logic, January 2002.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 522–538, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342612. doi: 10.1145/2908080.2908093. URL https://doi.org/10.1145/2908080.2908093.

John Power and Hiroshi Watanabe. Combining a monad and a comonad. *Theoretical Computer Science*, 280(1-2):137–162, 2002. ISSN 0304-3975. doi: 10.1016/S0304-3975(01)00024-X.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, jun 2008. ISSN 0362-1340. doi: 10.1145/1379022.1375602. URL https://doi.org/10.1145/1379022.1375602.

Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In Hans Jürgen Schneider and Hartmut Ehrig, editors, *Graph Transformations in Computer Science*, pages 358–379, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. ISBN 978-3-540-48333-5.

A.L. Smirnov. Graded monads and rings of polynomials. *Journal of Mathematical Sciences*, 151(3):3032–3051, 2008. ISSN 1072-3374. doi: 10.1007/s10958-008-9013-7. URL http://dx.doi.org/10.1007/s10958-008-9013-7.

Calvin Smith and Aws Albarghouthi. Synthesizing differentially private programs. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi: 10.1145/3341698.

Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149–168, 1972. doi: 10.1016/0022-4049(72)90019-9.

Phillip D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1):161–175, jan 1977. ISSN 0004-5411. doi: 10.1145/321992.322002. URL https://doi.org/10.1145/321992.322002.

Tarmo Uustalu and Varmo Vene. The Essence of Dataflow Programming. *Lecture Notes in Computer Science*, 4164:135–167, November 2006. doi: 10.1007/11894100_5.

Philip Wadler. Linear types can change the world! In Manfred Broy and Cliff B. Jones, editors, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561. North-Holland, 1990.

Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, jan 2003. ISSN 1529-3785. doi: 10.1145/601775.601776. URL https://doi.org/10.1145/601775.601776.

Peng Wang, Di Wang, and Adam Chlipala. Timl: a functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133903. URL https://doi.org/10.1145/3133903.

Uma Zalakain and Ornela Dardha. Pi with leftovers: a mechanisation in Agda. *arXiv preprint arXiv:2005.05902*, 2020.

Part II

APPENDIX

A

BENCHMARKING SUITES

A.1 SYNTHESISED PROGRAMS FOR THE GRADED LINEAR SYNTHESIS CALCULI

This section includes the synthesised programs for the benchmarking problems in Section 3.7.

A.1.1 *Hilbert*

1. ⊗-Intro:

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

2. ⊗-Elim:

```
and1 : ∀ { a b : Type } .  (a, b [0]) → a
and1 (y, [u]) = y

and2 : ∀ { a b : Type } . (a [0], b) → b
and2 ([u], z) = z
```

3. ⊕-Intro:

```
data Either a b = Left a | Right b

or1 : ∀ a b . a → Either a b
or1 x = Left x

or2 : ∀ a b . b → Either a b
or2 x = Right x
```

4. ⊕-Elim:

```
data Either a b = Left a | Right b

or3 : ∀ { a b c : Type }
    . (a → c) [0..1]
    → (b → c) [0..1]
    → (Either a b)
```

```
    → c
or3 [u] [v] (Right x6) = v x6;
or3 [u] [v] (Left x5) = u x5
```

5. SKI:

```
s : ∀ { a b c : Type }
  . (a → (b → c))
  → (a → b)
  → a [2]
  → c
s x y [u] = (x u) (y u)

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

i : ∀ { a : Type } . a → a
i x = x
```

A.1.2 *Comp*

1. 0/1:

```
comp-01 : ∀ { a b c : Type }
        . (a [] → b []) []
        → (b [] → c []) []
        → (a [] → c) []
comp-01 [z] [u] =
    [λv →
        let [w] = v in
        let [q] = z [w] in
        let [x9] = u [q] in
        let [x11] = z [w] in x9]
```

2. CBN:

```
comp-cbn : ∀ { a b c : Type }
         . (a [] → b) []
         → (b [] → c) [] → a [] → c
comp-cbn [u] [w] [v] = w [u [v]]
```

3. CBV:

```
comp-cbv : ∀ { a b c : Type }  .
         (a [] → b []) []
         → (b [] → c []) []
         → (a [] → c []) []
comp-cbv [z] [u] =
    [λv →
        (λ [w] →
            [(λ [q] → q) (u [(λ [x9] → x9) (z [w])])]
    ) v]
```

4. coK-$\mathcal{R}$:

```
compGen : ∀ {k : Coeffect, n m : k, a b c : Type}
        . (a [m] → b) [n]
        → (b [n] → c)
        → a [n ∗ m]
        → c
compGen [u] y [v] = y [u [v]]
```

5. coK-$\mathbb{N}$:

```
compNat : ∀ {n m : Nat, a b c : Type}
     . (a [m] → b) [n]
     → (b [n] → c)
     → a [n ∗ m]
     → c
compNat [u] y [v] = y [u [v]]
```

6. lin:

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

A.1.3 *Dist*

1. $\otimes$-!:

```
pull : ∀ { a b : Type } . (a [], b []) → (a, b) []
pull ([u], [v]) = [(u, v)]
```

2. $\otimes$-$\mathbb{N}$:

```
pull : ∀ {a b : Type, n : Nat}
     . (a [n], b [n])
     → (a, b) [n]
pull ([u], [v]) = [(u, v)]
```

3. $\otimes$-$\mathcal{R}$:

```
pull : ∀ {a b : Type, k : Coeffect, c : k}
     . (a [c], b [c])
     → (a, b) [c]
pull ([u], [v]) = [(u, v)]
```

4. $\oplus$-!:

```
data Either a b = Left a | Right b

pull : ∀ { a b : Type }
```

```
     . Either (a []) (b [])
     → (Either a b) []
pull (Right [v]) = [Right v];
pull (Left [z]) = [Left z]
```

5. ⊕-ℕ:

```
data Either a b = Left a | Right b

pull : ∀ {a b : Type, n : Nat}
     . Either (a [n]) (b [n])
     → (Either a b) [n]
pull (Right [v]) = [Right v];
pull (Left [z]) = [Left z]
```

6. ⊕-$\mathcal{R}$:

```
data Either a b = Left a | Right b

pull : ∀ { a b : Type, k : Coeffect, c : k }
     . Either (a [c]) (b [c])
     → (Either a b) [c]
pull (Right [v]) = [Right v];
pull (Left [z]) = [Left z]
```

7. —∘-!:

```
push : ∀ { a b : Type }
     . (a → b) []
     → a []
     → b []
push [z] [u] = [z u]
```

8. —∘-ℕ:

```
push : ∀ {a b : Type, c : Nat}
     . (a → b) [c]
     → a [c]
     → b [c]
push [z] [u] = [z u]
```

9. —∘-$\mathcal{R}$:

```
push : ∀ { a b : Type, k : Coeffect, c : k }
     . (a → b) [c]
     → a [c]
     → b [c]
push [z] [u] = [z u]
```

A.1.4 *Vec*

1. vec5:

```
vec5 : ∀ { a b : Type }
     . (a → b) [5]
     → ((((a, a), a), a), a)
     → ((((b, b), b), b), b)
vec5 [z] ((((x9, x10), x8), x6), x4) =
    ((((z x9, z x8), z x6), z x4), z x10)
```

2. vec10:

```
vec10 : ∀ { a b : Type }
      . (a → b) [10]
      → (((((((((a, a), a), a), a), a), a), a), a), a)
      → (((((((((b, b), b), b), b), b), b), b), b), b)
vec10 [z] (((((((((x19, x20), x18), x16), x14), x12),
    x10), x8), p), v) =
    (((((((((z x20, z v), z p), z x8), z x10), z x12), z
     x14), z x16), z x18), z x19)
```

3. vec15:

```
vec15 : ∀ { a b : Type }
      . (a → b) [15]
      → ((((((((((((((a, a), a), a), a), a), a), a), a)
    , a), a), a), a), a), a)
      → ((((((((((((((b, b), b), b), b), b), b), b), b)
    , b), b), b), b), b), b)
vec15 [z] ((((((((((((((x29, x30), x28), x26), x24), x22
    ), x20), x18), x16), x14), x12), x10), x8), p), v) =
    ((((((((((((((z x30, z v), z p), z x8), z x10), z
    x12), z x14), z x16), z x18), z x20), z x22), z x24)
    , z x26), z x28), z x29)
```

4. vec20:

```
vec20 : ∀ { a b : Type }
      . (a → b) [20]
      → ((((((((((((((((((((a, a), a), a), a), a), a), a
    ), a), a), a), a), a), a), a), a), a), a), a), a), a)
      → ((((((((((((((((((((b, b), b), b), b), b), b), b
    ), b), b), b), b), b), b), b), b), b), b), b), b), b)
vec20 [z] ((((((((((((((((((((x39, x40), x38), x36), x34)
    , x32), x30), x28), x26), x24), x22), x20), x18),
    x16), x14), x12), x10), x8), p), v) =
    ((((((((((((((((((((z x40, z v), z p), z x8), z x10),
     z x12), z x14), z x16), z x18), z x20), z x22), z
    x24), z x26), z x28), z x30), z x32), z x34), z x36)
    , z x38), z x39)
```

A.1.5 *Misc*

1. split⊕:

```
data Either a b where Left a | Right b

splitPlus : ∀ { a b c : Type }
          . b [2..3]
          → Either a c
          → Either (a, b [2..2]) (c, b [3..3])
splitPlus [z] (Right x4) = Right (x4, [z]);
splitPlus [z] (Left x3) = Left (x3, [z])
```

2. split⊗:

```
splitPair : ∀ { a : Type }
          . (a → a → a) [0..2]
          → a [10..10]
          → (a [2..2], a [6..6])
splitPair [z] [u] = ([(z u) u], [u])
```

3. share:

```
share : ∀ { a : Type }
      . (a → a → a) [0..2]
      → a [10..10]
      → (a [2..2], a [6..6])
share [z] [u] = ([(z u) u], [u])
```

4. Exm. 3.1.2:

```
dontLeak : ∀ { a b : Type }
         . (a [Public], a [Private])
         → ((a, ()) [Public] → b)
         → b
dontLeak ([w], [v]) y = y [(w, ())]
```

A.2 SYNTHESISED PROGRAMS FOR THE FULLY GRADED SYNTHESIS CALCULUS

This section includes the synthesised programs, their synthesis contexts, and examples used for the benchmarking problems in Section 4.5. The context of each synthesised program is listed by the program's `spec`. If the program has no `spec`, then synthesis is occurring in an empty context.

For each program we list the examples required to synthesise the correct result in the Graded case, and the additional examples required to synthesise the same program without grades (i.e. in the Cartesian setting). See Section 4.5 for further details.

A.2.1 *List*

1. append:

```
language GradedBase

data List a = Nil | Cons a (List a)

append : ∀ { a : Type }
       . (List a) %1
       → a %1
       → List a
append x y = (Cons y) x
```

with no Graded examples and Cartesian example(s):

```
append (Cons 1 Nil) 2 = (Cons 2 (Cons 1 Nil));
```

2. concat:

```
language GradedBase

data List a = Cons a (List a) | Nil

concat : ∀ { a : Type }
       . (List a) %1..∞
       → (List a) %1..∞
       →  List a
spec
    concat %0..∞
concat Nil y = y;
concat (Cons z u) y = (Cons z) ((concat u) y)
```

with no Graded examples and Cartesian examples(s):

```
concat (Cons 0 Nil) Nil = (Cons 0 Nil);
concat (Cons 0 Nil) (Cons 1 Nil) = (Cons 0 (Cons 1 Nil))
    ;
concat (Cons 0 (Cons 1 (Cons 2 Nil))) (Cons 3 (Cons 4
    Nil)) = Cons 0 (Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil))
    ));
```

3. empty:

```
language GradedBase

data List a = Nil | Cons a (List a)

empty : ∀ { a b : Type } . () → List a
empty () = Nil
```

4. snoc:

```
language GradedBase

data List a = Nil | Cons a (List a)

data N = S N | Z
```

```
snoc : ∀ { a : Type }
     . (List a) %1..∞
     → a %1..∞
     → List a
spec
    snoc %0..∞
snoc x y =
    (case x of
      Nil → (Cons y) x;
      Cons z u → (Cons z) ((snoc u) y))
```

with Graded example(s):

```
snoc (Cons Z Nil) (S Z) = (Cons Z (Cons (S Z) Nil));
```

5. drop:

```
language GradedBase

data List a = Cons a (List a) | Nil

data N = S N | Z

drop : ∀ { a : Type } . N %0..∞ → List (a) %0..∞ →
    List a
spec
    drop % 0..∞
drop x y =
    (case y of
        Nil →
    (case x of
        Z → y;
        S z → (drop z) y);
        Cons p q →
    (case x of
        Z → q;
        S x8 → q))
```

with Graded example(s):

```
drop (S Z) (Cons (S Z) (Cons Z Nil)) = Cons Z Nil;
```

6. flatten:

```
language GradedBase

data List a = Cons a (List a) | Nil

append : ∀ { a : Type }
       . (List a) %1..∞
       → (List a) %1..∞
       →  List a
```

```
append Nil y = y;
append (Cons z u) y = (Cons z) ((append u) y)

flatten : ∀ { a : Type } . (List (List a)) %0..∞ → (
    List a)
spec
    flatten % 0..∞ , append %0..∞
flatten Nil = (append Nil) Nil;
flatten (Cons u v) = (append u) (flatten v)
```

with Graded example(s):

```
flatten (Cons (Cons 1 Nil) (Cons (Cons 1 Nil) Nil)) =
    Cons 1 (Cons 1 Nil);
```

7. bind:

```
language GradedBase

data List a = Nil | Cons a (List a)

data N = S N | Z

data Bool = True | False;

append : ∀ { a : Type }
       . (List a) %1..∞
       → (List a) %1..∞
       →  List a
append Nil y = y;
append (Cons z u) y = (Cons z) ((append u) y)

concat : ∀ { a : Type } . (List (List a)) %1..∞ → (
    List a)
concat Nil = Nil;
concat (Cons u v) = (append u) (concat v)

map : ∀ { a b : Type }
    . (a %1..∞ → b) %0..∞
    → List a %1..∞
    → List b
map x Nil = Nil;
map x (Cons z u) = (Cons (x z)) ((map x) u)

isEven : N %1..∞ → List N
isEven Z = Nil;
isEven (S Z) = Cons (S Z) Nil;
isEven (S (S z)) = concat (Cons (isEven z) Nil)

bind : ∀ { a b : Type }
     . List a %1..∞
     → (a %1..∞ → List b) %0..∞
     → List b
```

```
spec
    map %1..∞, concat %1..∞
bind x y = concat ((map (λw → w)) x)
```

with no Graded examples and Cartesian example(s):

```
bind (Cons Z Nil) isEven = Nil;
bind (Cons (S Z) Nil) isEven = (Cons (S Z) Nil);
```

8. return:

```
language GradedBase

data List a =  Cons a (List a) | Nil

return : ∀ { a b : Type } . a → List a
return x = (Cons x) Nil
```

with no Graded examples and Cartesian example(s):

```
return 1 = Cons 1 Nil;
```

9. inc:

```
language GradedBase

data List a = Cons a (List a) | Nil

data N = S N | Z

map : (List N) %1..∞
    → (N %1..∞ → N) %0..∞
    → (List N)
map Nil f = Nil;
map (Cons x xs) f = (Cons (f x) (map xs f))

inc : ∀ a . (List N) %1..∞ → (List N)
spec
    map %1..∞
inc x = (map x) (λu → S u)
```

with Graded example(s):

```
inc (Cons (S Z) Nil) = (Cons (S (S Z)) Nil);
```

10. head:

```
language GradedBase

data List a = Cons a (List a) | Nil

head : ∀ { a : Type } . (List a) %0..1 → a %0..1 → a
head Nil y = y;
head (Cons z u) y = z
```

with Graded example(s):

```
head (Cons 1 (Cons 2 Nil)) 2 = 1;
```

11. tail:

```
language GradedBase

data List a = Cons a (List a) | Nil

tail : ∀ { a : Type } . List a %0..1 → List a
tail Nil = Nil;
tail (Cons y z) = z
```

with Graded example(s):

```
tail (Cons 1 (Cons 2 Nil)) = Cons 2 Nil;
```

12. last:

```
language GradedBase

data List a = Cons a (List a) | Nil

data Maybe a = Just a | Nothing

last : ∀ { a : Type } . (List a) %0..∞ → Maybe a
spec
    last %0..∞
last Nil = Nothing;
last (Cons y z) =
    (case z of
        Nil → last z;
        Cons v w → Just v)
```

with Graded example(s):

```
last (Cons 1 (Cons 2 Nil)) = Just 2;
```

and Cartesian example(s):

```
last (Cons 1 Nil) = Just 1;
```

13. length:

```
language GradedBase

data List a = Cons a (List a) | Nil

data N = S N | Z

length : ∀ { a : Type } . List a %0..∞ → N
spec
    length %0..∞
length Nil = Z;
length (Cons y z) = S (length z)
```

with Graded example(s):

```
length (Cons 1 (Cons 1 Nil)) = S (S Z);
```

14. map:

```
language GradedBase

data List a = Nil | Cons a (List a)

data Bool = True | False;

neg : Bool %1..∞ → Bool
neg True = False;
neg False = True

map : ∀ { a b : Type }
    . (a % 1..∞ → b) %0..∞
    → List a %1..∞
    → List b
spec
    map % 0..∞
map x Nil = Nil;
map x (Cons z u) = (Cons (x z)) ((map x) u)
```

with no Graded examples and Cartesian example(s):

```
map neg (Cons True Nil) = Cons False Nil;
```

15. replicate5:

```
language GradedBase

data List a =  Cons a (List a) | Nil

replicate5 : ∀ { a : Type } . a %5 → List a
replicate5 x = (Cons x) ((Cons x) ((Cons x) ((Cons x) ((
    Cons x) Nil))))
```

with no Graded examples and Cartesian example(s):

```
replicate5 1 = Cons 1 (Cons 1 (Cons 1 (Cons 1 (Cons 1
    Nil))));
```

16. replicate10:

```
language GradedBase

data List a =  Cons a (List a) | Nil

replicate10 : ∀ { a : Type } . a %10 → List a
replicate10 x = (Cons x) ((Cons x) ((Cons x) ((Cons x)
    ((Cons x) ((Cons x) ((Cons x) ((Cons x) ((Cons x) ((
    Cons x) Nil)))))))))
```

with no Graded examples and Cartesian example(s);

```
replicate10 1 = (Cons 1 (Cons 1 (Cons 1 (Cons 1 (Cons 1
    (Cons 1 (Cons 1 (Cons 1 (Cons 1 (Cons 1 Nil)))))))))
    );
```

17. replicateN:

```
language GradedBase

data List a =  Cons a (List a) | Nil

data N = S N | Z

replicateN : ∀ { a : Type }
           . N %0..∞
           → a %0..∞
           → List a
spec
    replicateN % 0..∞
replicateN Z y = Nil;
replicateN (S z) y = (Cons y) ((replicateN z) y)
```

with Graded example(s):

```
replicateN (S (S Z)) 2 = (Cons 2 (Cons 2 Nil));
```

18. stutter:

```
language GradedBase

data List a = Cons a (List a) | Nil

stutter : ∀ { a : Type } . List (a [2]) %1..∞ → List a
spec
    stutter % 0..∞
stutter Nil = Nil;
stutter (Cons [u] z) = (Cons u) ((Cons u) (stutter z))
```

19. sum:

```
language GradedBase

data N = S N | Z

data List a = Cons a (List a) | Nil

fold : List N %0..∞
     → (N % 1..∞
     → N % 1..∞ → N) %0..∞
     → N %0..∞ → N
fold Nil f acc = acc;
fold (Cons x xs) f acc =  fold xs f (f acc x)
```

```
add : N %1..∞ → N %1..∞ → N
add Z n2 = n2;
add (S n1) n2 = S (add n1 n2)

sum : List N %0..∞ → N
spec
    fold %0..∞, add %0..∞
sum x = ((fold x) (λp → λq → (add p) q)) ((add Z) Z)
```

with Graded example(s):

```
sum (Cons (S Z) (Cons (S Z) Nil)) = (S (S Z));
```

and Cartesian example(s):

```
sum (Cons (S Z) (Cons (S Z) (Cons (S Z) Nil))) = (S (S (
    S Z)));
```

A.2.2 *Stream*

1. build:

```
language CBN
language GradedBase

data Stream a = Next a (Stream a)

ones : () %1..1 → Stream Int
ones () = Next 1 (ones ())

head : ∀ { a : Type } . Stream a %0..1 → a
head (Next x xs) = x

build : ∀ { a : Type } . a %1..1 → (Stream a) %1..1 →
     Stream a
build x y = (Next x) y
```

with no Graded examples and Cartesian example(s):

```
head (build 2 (ones ())) = 2;
```

2. map:

```
language CBN
language GradedBase

data Stream a = Next a (Stream a)

data Bool = True | False

trues : () %1..∞ → Stream Bool
trues () = Next True (trues ())
```

```
neg : Bool %1..∞ → Bool
neg True = False;
neg False = True

head : ∀ { a : Type } . Stream a %0..∞ → a
head (Next x xs) = x

map : ∀ { a b : Type }
    . Stream a %1..∞
    → (a %1..∞ → b) %1..∞
    → Stream b
spec
    map % 1..∞
map (Next z u) y = (Next (y z)) ((map_stream u) y)
```

with no Graded examples and Cartesian example(s):

```
head (map_stream (trues ()) neg) = False;
```

3. take1:

```
language CBN
language GradedBase

data Stream a = Next a (Stream a)

data Bool = True | False

ones : () %1..1 → Stream Int
ones () = Next 1 (ones ())

head : ∀ { a : Type } . (Stream a) %0..1 → a
head (Next x _) = x

take1 : ∀ { a : Type } . Stream a %0..1 → a
take1 (Next y z) = y
```

with no Graded examples and Cartesian example(s):

```
take1 (Next 2 (ones ())) = 2;
```

4. take2:

```
language CBN
language GradedBase

data Stream a = Next a (Stream a)
data Bool = True | False

ones : () %1..1 → Stream Int
ones () = Next 1 (ones ())

head : ∀ { a : Type } . (Stream a) %0..1 → a
```

```
head (Next x _) = x

take2 : ∀ { a : Type } . Stream a %0..1 → (a, a)
take2 (Next y (Next u v)) = (u, y)
```

with no Graded examples and Cartesian example(s):

```
take2 (Next 2 (ones ())) = (2, 1);
```

5. take3:

```
language CBN
language GradedBase

data Stream a = Next a (Stream a)

data Bool = True | False

ones : () %1..1 → Stream Int
ones () = Next 1 (ones ())

head : ∀ { a : Type } . (Stream a) %0..1 → a
head (Next x _) = x

take3 : ∀ { a : Type } . Stream a %0..1 → (a, (a, a))
take3 (Next y (Next u (Next w p))) = (y, (w, u))
```

with no Graded examples and Cartesian example(s):

```
take3 (Next 3 (Next 2 (ones ()))) = (3, (2, 1));
```

A.2.3 *Bool*

1. neg:

```
language GradedBase

data Bool = True | False

neg : Bool %1 → Bool
neg True = False;
neg False = True
```

with Graded example(s):

```
neg True = False;
neg False = True;
```

2. and:

```
language GradedBase

data Bool = True | False
```

```
and : Bool %1 → Bool %1 → Bool
and True True = True;
and False True = False;
and True False = False;
and False False = False
```

with Graded example(s):

```
and True True = True;
and False True = False;
and True False = False;
and False False = False;
```

3. impl:

```
language GradedBase

data Bool = True | False

impl : Bool %1 → Bool %1 → Bool
impl True True = True;
impl True False = False;
impl False True = True;
impl False False = True
```

with Graded example(s):

```
impl True True = True;
impl True False = False;
impl False True = True;
impl False False = True;
```

4. or:

```
language GradedBase

data Bool = True | False

or : Bool %1 → Bool %1 → Bool
or True True = True;
or False True = True;
or True False = True;
or False False = False
```

with Graded example(s):

```
or True True = True;
or False True = True;
or True False = True;
or False False = False
```

5. xor:

```
language GradedBase

data Bool = True | False

xor : Bool %1 → Bool %1 → Bool
xor True True = False;
xor False True = True;
xor True False = True;
xor False False = False
```

with Graded example(s):

```
xor True True = False;
xor False True = True;
xor True False = True;
xor False False = False
```

A.2.4 *Maybe*

1. bind:

```
language GradedBase

data Maybe a = Just a | Nothing

data N = S N | Z

data Bool = True | False;

isEven : N %1..∞ → Maybe N
isEven Z = Nothing;
isEven (S Z) = Just (S Z);
isEven (S (S z)) =
    case isEven z of
        Nothing → Nothing;
        Just (S Z) → Just (S Z)

bind : ∀ { a b : Type }
     . Maybe a %1..1
     → (a %1..1 → Maybe b) %0..1
     → Maybe b
bind Nothing y = Nothing;
bind (Just z) y = y z
```

with no Graded example(s) and Cartesian example(s):

```
bind (Just (S Z)) isEven = Just (S Z);
```

2. fromMaybe:

```
language GradedBase
```

```
data Maybe a = Nothing | Just a

fromMaybe : ∀ { a : Type } . Maybe a %(1..1) → a
    %(0..1) → a
fromMaybe Nothing y = y;
fromMaybe (Just z) y = z
```

with no Graded example(s) and Cartesian example(s):

```
fromMaybe  Nothing 1 = 1;
fromMaybe (Just 1) 2 = 1;
```

3. return:

```
language GradedBase

data Maybe a = Nothing | Just a

return : ∀ { a : Type } . a → Maybe a
return x = Just x
```

4. isJust:

```
language GradedBase

data Maybe a = Just a | Nothing

data Bool = True | False

isJust : ∀ { a : Type } . (Maybe a) %0..1 → Bool
isJust Nothing = False;
isJust (Just y) = True
```

with Graded example(s):

```
isJust (Just 1) = True;
isJust Nothing = False;
```

5. isNothing:

```
language GradedBase

data Maybe a = Just a | Nothing

data Bool = True | False

isNothing : ∀ { a : Type } . (Maybe a) %0..1 → Bool
isNothing Nothing = True;
isNothing (Just y) = False
```

with Graded example(s):

```
isNothing (Just 1) = False;
isNothing Nothing = True;
```

6. map:

```
language GradedBase

data Maybe a = Nothing | Just a

data N = S N | Z
data Bool = True | False

isOne : N %0..1 → Bool
isOne (S Z) = True;
isOne _ = False

map : ∀ { a b : Type }
    . (a → b) %(0..1)
    → (Maybe a) %(1..1)
    → Maybe b
map x Nothing = Nothing;
map x (Just z) = Just (x z)
```

with no Graded examples and Cartesian example(s):

```
map isOne (Just (S Z)) = Just True;
```

7. mplus:

```
language GradedBase

data Maybe a = Nothing | Just a

mplus : ∀ { a b : Type }
      . Maybe a %(0..1)
      → Maybe b %(0..1)
      → Maybe (a, b)
mplus Nothing Nothing = Nothing;
mplus (Just z) Nothing = Nothing;
mplus Nothing (Just v) = Nothing;
mplus (Just w) (Just v) = Just (w, v)
```

with Graded example(s):

```
mplus (Just 1) (Just 2) = Just (1, 2);
```

A.2.5 *Nat*

1. isEven:

```
language GradedBase

data N = S N | Z

data Bool = True | False
```

```
isEven : N %1..∞ → Bool
spec
    isEven % 0..∞
isEven Z = True;
isEven (S Z) = False;
isEven (S (S z)) = isEven z
```

with Graded example(s):

```
isEven (S (S Z)) = True;
isEven (S (S (S Z))) = False;
```

2. pred:

```
language GradedBase

data N = S N | Z

pred : N %1 → N
pred Z = Z;
pred (S y) = y
```

with Graded example(s):

```
pred (S (S Z)) = (S Z);
```

3. succ:

```
language GradedBase

data N = S N | Z

succ : N %1 → N
succ x = S x
```

with Graded example(s):

```
succ Z = (S Z);
```

4. sum:

```
language GradedBase

data N = S N | Z

sum : N %1..∞ → N %1..∞ → N
spec
    sum % 0..∞
sum Z y = y;
sum (S z) y = S ((sum z) y)
```

with Graded example(s):

```
sum (S Z) (S Z) = (S (S Z));
```

and Cartesian example(s):

```
sum (S Z) Z = (S Z);
sum (S (S (S (S Z)))) (S (S Z)) = (S (S (S (S (S (S Z)))
    )));
```

A.2.6 *Tree*

1. map:

```
language GradedBase

data Tree a = Leaf | Node (Tree a) a (Tree a)

data Bool = True | False

neg : Bool %1..∞ → Bool
neg True = False;
neg False = True

map : ∀ { a b : Type }
    . Tree a %1..∞
    → (a %1..∞ → b) %0..∞
    → Tree b
spec
    map %0..∞
map Leaf y = Leaf;
map (Node z u v) y = ((Node ((map z) y)) (y u)) ((map v)
     y)
```

with no Graded examples and Cartesian example(s):

```
map (Node Leaf True Leaf) neg = (Node Leaf False Leaf);
```

2. sutter:

```
language GradedBase

data Tree a = Leaf | Node (Tree a) a (Tree a)

stutter : ∀ { a b : Type } . Tree (a [2]) %1..∞ → Tree
     (a, a)
spec
    stutter %0..∞
stutter Leaf = Leaf;
stutter (Node y [v] u) = ((Node (stutter y)) (v, v)) (
    stutter u)
```

with no Graded examples and Cartesian example(s):

```
stutter (Node Leaf [1] Leaf) = (Node Leaf (1, 1) Leaf);
```

3. sum:

```
language GradedBase

data N = S N | Z
:W
data Tree a = Leaf | Node (Tree a) a (Tree a)

add : N %1..∞ → N %1..∞ → N
add Z y = y;
add (S z) y = S ((add z) y)

fold : Tree N %0..∞
     → (N %1..∞ → N %1..∞ → N) %0..∞
     → N %0..∞
     → N
fold Leaf f acc = acc;
fold (Node l x r) f acc =
    fold l f (f x (fold r f acc))

sum : ∀ { a : Type } . Tree N %0..∞ → N
spec
    fold %1..∞, add %1..∞
sum x = (add Z) (((fold x) (λp → λq → (add p) q)) Z)
```

with Graded example(s):

```
sum Leaf = Z;
sum (Node Leaf (S (S Z)) Leaf) = (S (S Z));
sum (Node (Node Leaf (S Z) (Node Leaf (S Z) Leaf)) (S (S
     Z)) Leaf) = (S (S (S (S Z))));
```

A.2.7 *Misc*

1. compose:

```
language GradedBase

comp : ∀ {k : Coeffect, n m : k, a b c : Type}
     . (a %m → b) %n
     → (b %n → c) %(1 : k)
     → a %(n * m)
     → c
comp x y z = y (x z)
```

2. copy:

```
language GradedBase

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

3. push:

```
language GradedBase

push : ∀ { a b : Type, k : Coeffect, c : k }
     . (a → b) %c
     → a %c
     → b [c]
push x y = [x y]
```

B

PROOFS

B.1 PROOFS FOR THE LINEAR-BASE CALCULI

This section gives the proofs of Lemma 3.3.1 and Lemma 3.4.1, along with soundness results for the additive pruning variant.

We first state and prove some intermediate results about context manipulations which are needed for the main lemmas.

Definition B.1.1 (Context approximation)**.** For contexts Γ_1, Γ_2 then:

$$\frac{}{\varnothing \sqsubseteq \varnothing} \qquad \frac{\Gamma_1 \sqsubseteq \Gamma_2}{\Gamma_1, x : A \sqsubseteq \Gamma_2, x : A}$$

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \qquad r \sqsubseteq s}{\Gamma_1, x :_r A \sqsubseteq \Gamma_2, x :_s A} \qquad \frac{\Gamma_1 \sqsubseteq \Gamma_2 \qquad 0 \sqsubseteq s}{\Gamma_1 \sqsubseteq \Gamma_2, x :_s A}$$

This is actioned in type checking by iterative application of Approx.

Lemma B.1.1 ($\Gamma + (\Gamma' - \Gamma'') \sqsubseteq (\Gamma + \Gamma') - \Gamma''$)**.**

Proof. Induction over the structure of both Γ' and Γ''. The possible forms of Γ' and Γ'' are considered in turn:

1. $\Gamma' = \varnothing$ and $\Gamma'' = \varnothing$
   We have:

   $$(\Gamma + \varnothing) - \varnothing = \Gamma + (\varnothing - \varnothing)$$

   From definitions 2.3.3 and 3.3.1, we know that on the left hand side:

   $$\begin{aligned}(\Gamma + \varnothing) - \varnothing &= \Gamma + \varnothing \\ &= \Gamma\end{aligned}$$

   and on the right-hand side:

   $$\begin{aligned}\Gamma + (\varnothing - \varnothing) &= \Gamma + \varnothing \\ &= \Gamma\end{aligned}$$

   making both the left and right hand sides equivalent:

   $$\Gamma = \Gamma$$

2. $\Gamma' = \Gamma', x : A$ and $\Gamma'' = \emptyset$
   We have

   $$(\Gamma + \Gamma', x : A) - \emptyset = \Gamma + (\Gamma, x : A - \emptyset)$$

   From definitions 2.3.3 and 3.3.1, we know that on the left hand side we have:

   $$\begin{aligned}(\Gamma + \Gamma', x : A) - \emptyset &= (\Gamma, \Gamma'), x : A - \emptyset \\ &= (\Gamma, \Gamma'), x : A\end{aligned}$$

   and on the right hand side:

   $$\begin{aligned}\Gamma + (\Gamma, x : A - \emptyset) &= \Gamma + \Gamma', x : A \\ &= (\Gamma, \Gamma', x : A)\end{aligned}$$

   making both the left and right hand sides equal:

   $$(\Gamma, \Gamma'), x : A = (\Gamma, \Gamma'), x : A$$

3. $\Gamma' = \Gamma', x : A$ and $\Gamma'' = \Gamma'', x : A$
   We have

   $$(\Gamma + \Gamma', x : A) - \Gamma'', x : A = \Gamma + (\Gamma', x : A - \Gamma'', x : A)$$

   From definitions 2.3.3 and 3.3.1, we know that on the left hand side we have:

   $$\begin{aligned}(\Gamma + \Gamma', x : A) - \Gamma'', x : A &= (\Gamma, \Gamma'), x : A - \Gamma'', x : A \\ &= \Gamma, \Gamma' - \Gamma''\end{aligned}$$

   and on the right hand side:

   $$\begin{aligned}\Gamma + (\Gamma', x : A - \Gamma'', x : A) &= \Gamma + (\Gamma' - \Gamma'') \\ &= \Gamma, \Gamma' - \Gamma''\end{aligned}$$

   making both the left and right hand sides equivalent:

   $$\Gamma, \Gamma' - \Gamma'' = \Gamma, \Gamma' - \Gamma''$$

4. $\Gamma' = \Gamma', x :_r A$ and $\Gamma'' = \emptyset$
   We have

   $$(\Gamma + \Gamma', x :_r A) - \emptyset = \Gamma + (x :_r A - \emptyset)$$

   From definitions 2.3.3 and 3.3.1, we know that on the left hand side we have:

   $$\begin{aligned}(\Gamma + \Gamma', x :_r A) - \emptyset &= (\Gamma + \Gamma', x :_r A) \\ &= (\Gamma, \Gamma'), x :_r A\end{aligned}$$

and on the right hand side:

$$\Gamma + (\Gamma', x :_r A - \emptyset) = \Gamma + (\Gamma', x :_r A) \quad = (\Gamma, \Gamma'), x :_r A$$

making both the left and right hand sides equivalent:

$$(\Gamma, \Gamma'), x :_r A = (\Gamma, \Gamma'), x :_r A$$

5. $\Gamma' = \Gamma', x :_r A$ and $\Gamma'' = \Gamma'', x :_s A$

Thus we have (for the LHS of the inequality term):

$$\Gamma + (\Gamma', x :_r A - \Gamma'', x :_s A)$$

which by context subtraction yields:

$$\Gamma + (\Gamma', x :_r A - \Gamma'', x :_s A) = \Gamma + (\Gamma' - \Gamma''), x :_{q'} A$$

where:

$$\exists q'.r \sqsupseteq q' + s \quad \forall \hat{q'}.r \sqsupseteq \hat{q'} + s \implies q' \sqsupseteq \hat{q'} \qquad (2)$$

And for the LHS of the inequality, from definitions 2.3.3 and 3.3.1 we have:

$$\begin{aligned}(\Gamma + \Gamma', x :_r A) - \Gamma'', x :_s A &= (\Gamma + \Gamma'), x :_r A - \Gamma'', x :_s A \\ &= ((\Gamma + \Gamma') - \Gamma''), x :_r A - x :_s A \\ &= ((\Gamma + \Gamma') - \Gamma''), x :_q A\end{aligned}$$

where:

$$\exists q.r \sqsupseteq q + s \quad \forall \hat{q}.r \sqsupseteq \hat{q} + s \implies q \sqsupseteq \hat{q} \qquad (1)$$

Applying $\exists q.r \sqsupseteq q + s$ to maximality (2) (at $\hat{q'} = q$) then yields that $q \sqsubseteq q'$.

Therefore, applying induction, we derive:

$$\frac{(\Gamma + (\Gamma' - \Gamma'')) \sqsubseteq ((\Gamma + \Gamma') - \Gamma'') \qquad q \sqsubseteq q'}{(\Gamma + (\Gamma' - \Gamma'')), x :_q A \sqsubseteq ((\Gamma + \Gamma') - \Gamma''), x :_{q'} A}$$

satisfying the lemma statement.

□

Lemma B.1.2 $((\Gamma - \Gamma') + \Gamma' \sqsubseteq \Gamma)$**.**

Proof. The proof follows by induction over the structure of Γ'. The possible forms of Γ' are considered in turn:

1. $\Gamma' = \emptyset$
   We have:

   $$(\Gamma - \emptyset) + \emptyset = \Gamma$$

   From definition 3.3.1, we know that:

   $$\Gamma - \emptyset = \Gamma$$

   and from definition 2.3.3, we know:

   $$\Gamma + \emptyset = \Gamma$$

   giving us:

   $$\Gamma = \Gamma$$

2. $\Gamma' = \Gamma'', x : A$
   and let $\Gamma = \Gamma', x : A$.

   $$(\Gamma', x : A - \Gamma'', x : A) + \Gamma'', x : A = \Gamma$$

   From definition 2.3.3, we know that:

   $$\begin{aligned}(\Gamma', x : A - \Gamma'', x : A) + \Gamma'', x : A &= ((\Gamma' - \Gamma'') + \Gamma''), x : A \\ \textit{induction} &= \Gamma', x : A \\ &= \Gamma\end{aligned}$$

   thus satisfying the lemma statement by equality.

3. $\Gamma' = \Gamma'', x :_r A$
   and let $\Gamma = \Gamma', x :_s A$.
   We have:

   $$(\Gamma', x :_s A - \Gamma'', x :_r A) + \Gamma'', x :_r A$$

   From definition 3.3.1, we know that:

   $$\begin{aligned}&(\Gamma', x :_s A - \Gamma'', x :_r A) + \Gamma'', x :_r A \\ &= (\Gamma' - \Gamma''), x :_q A + \Gamma'', x :_r A \\ &= ((\Gamma' - \Gamma'') + \Gamma''), x :_{q+r} A\end{aligned}$$

   where $s \sqsupseteq q + r$ and $\forall q'.s \sqsupseteq q' + r \implies q \sqsupseteq q'$.
   Then by induction we derive the ordering:

   $$\frac{((\Gamma' - \Gamma'') + \Gamma'') \sqsubseteq \Gamma' \qquad q + r \sqsubseteq s}{((\Gamma' - \Gamma'') + \Gamma''), x :_{q+r} A \sqsubseteq \Gamma', x :_s A}$$

   which satifies the lemma statement.

□

Lemma B.1.3 (Context negation)**.** For all contexts Γ:

$$\emptyset \sqsubseteq \Gamma - \Gamma$$

Proof. By induction on the structure of Γ:

- $\Gamma = \emptyset$ Trivial.
- $\Gamma = \Gamma', x : A$ then $(\Gamma', x : A) - (\Gamma', x : A) = \Gamma' - \Gamma'$ so proceed by induction.
- $\Gamma = \Gamma', x :_r A$ then $\exists q.\ (\Gamma', x :_r A) - (\Gamma', x :_r A) = (\Gamma - \Gamma'), x :_q A$ such that $r \sqsupseteq q + r$ and $\forall q'. r \sqsupseteq q' + r \implies q \sqsupseteq q'$.

  Instantiating maximality with $q' = 0$ and reflexivity then we have $0 \sqsubseteq q$. From this, and the inductive hypothesis, we can construct:

$$\frac{\emptyset \sqsubseteq (\Gamma - \Gamma') \quad 0 \sqsubseteq q}{\emptyset \sqsubseteq (\Gamma - \Gamma'), x :_q A}$$

□

Lemma B.1.4. For all contexts Γ_1, Γ_2, where $[\Gamma_2]$ (i.e., Γ_2 is all graded) then:

$$\Gamma_2 \sqsubseteq \Gamma_1 - (\Gamma_1 - \Gamma_2)$$

Proof. By induction on the structure of Γ_2.

- $\Gamma_2 = \sqsubseteq$

  Then $\Gamma_1 - (\Gamma_1 - \emptyset) = \Gamma_1 - \Gamma_1$.

  By Lemma B.1.3, then $\emptyset \sqsubseteq (\Gamma_1 - \Gamma_1)$ satisfying this case.

- $\Gamma_2 = \Gamma_2', x :_s A$

  By the premises $\Gamma_1 \sqsubseteq \Gamma_2$ then we can assume $x \in \Gamma_1$ and thus (by context rearrangement) $\Gamma_1', x :_r A$.

  Thus we consider $(\Gamma_1', x :_r A) - ((\Gamma_1', x :_r A) - (\Gamma_2', x :_s A))$.

$$\begin{aligned} & (\Gamma_1', x :_r A) - ((\Gamma_1', x :_r A) - (\Gamma_2', x :_s A)) \\ = \ & (\Gamma_1', x :_r A) - ((\Gamma_1' - \Gamma_2'), x :_q A) \\ = \ & (\Gamma_1' - (\Gamma_1' - \Gamma_2')), x :_{q'} A \end{aligned}$$

  where (1) $\exists q.\ r \sqsupseteq q + s$ with (2) $(\forall \hat{q}. r \sqsupseteq \hat{q} + s \implies q \sqsupseteq \hat{q})$
  and (3) $\exists q'.\ r \sqsupseteq q' + q$ with (4) $(\forall \hat{q}'. r \sqsupseteq \hat{q}' + s \implies q' \sqsupseteq \hat{q}')$.

Apply (1) to (4) by letting $\hat{q'} = s$ and by commutativity of $+$ then we get that $q' \sqsupseteq s$.

By induction we have that

$$\Gamma_1' \sqsubseteq \Gamma_1' - (\Gamma_1' - \Gamma_2') \tag{ih}$$

Thus we get that:

$$\frac{s \sqsubseteq q' \quad \Gamma_1' \sqsubseteq \Gamma_1' - (\Gamma_1' - \Gamma_2')}{\Gamma_1', x :_s A \sqsubseteq (\Gamma_1' - (\Gamma_1' - \Gamma_2')), x :_{q'} A}$$

- $\Gamma_2 = \Gamma_2', x : A$ Trivial as it violates the grading condition of the premise.

□

B.1.1 *Soundness of the Subtractive Graded Linear Typing Calculus*

Lemma 3.3.1 (Subtractive synthesis soundness)**.** For all Γ and A then:

$$\Gamma \vdash A \Rightarrow^- t \mid \Delta \quad \Longrightarrow \quad \Gamma - \Delta \vdash t : A$$

i.e. t has type A under context $\Gamma - \Delta$, that contains just those linear and graded variables with grades reflecting their use in t.

Proof. Structural induction over the synthesis rules. Each of the possible synthesis rules are considered in turn.

1. Case LinVar$^-$
   In the case of linear variable synthesis, we have the derivation:

   $$\frac{}{\Gamma, x : A \vdash A \Rightarrow^- x \mid \Gamma}\ \textsc{LinVar}^-$$

   By the definition of context subtraction, $(\Gamma, x : A) - \Gamma = x : A$, thus we can construct the following typing derivation, matching the conclusion:

   $$\frac{}{x : A \vdash x : A}\ \textsc{Var}$$

2. Case GrVar$^-$
   Matching the form of the lemma, we have the derivation:

   $$\frac{\exists s.\, r \sqsupseteq s + 1}{\Gamma, x :_r A \vdash A \Rightarrow^- x \mid \Gamma, x :_s A}\ \textsc{GrVar}^-$$

   By the definition of context subtraction, $(\Gamma, x :_r A) - (\Gamma, x :_s A) = x :_q A$ where (1) $\exists q.\, r \sqsupseteq q + s$ and $\forall q'.r \sqsupseteq q' + s \implies q \sqsupseteq q'$.

Applying maximality (1) with $q = 1$ then we have that $1 \sqsubseteq q$ (*)

Thus, from this we can construct the typing derivation, matching the conclusion:

$$\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}\ \textsc{Var}}{x :_1 A \vdash x : A \qquad 1 \sqsubseteq q\ (*)}\ \textsc{Der}}{x :_q A \vdash x : A}\ \textsc{Approx}$$

3. Case $\multimap^-_R$
We thus have the derivation:

$$\dfrac{\Gamma, x : A \vdash B \Rightarrow^- t \mid \Delta \qquad x \notin |\Delta|}{\Gamma \vdash A \multimap B \Rightarrow^- \lambda x.t \mid \Delta}\ \multimap^-_R$$

By induction we then have that:

$$(\Gamma, x : A) - \Delta \vdash t : B$$

Since $x \notin |\Delta|$ then by the definition of context subtraction we have that $(\Gamma, x : A) - \Delta = (\Gamma - \Delta), x : A$. From this, we can construct the following derivation, matching the conclusion:

$$\dfrac{(\Gamma - \Delta), x : A \vdash t : B}{\Gamma - \Delta \vdash \lambda x.t : A \multimap B}\ \textsc{Abs}$$

4. Case $\multimap^-_L$
Matching the form of the lemma, the application derivation is:

$$\dfrac{\Gamma, x_2 : B \vdash C \Rightarrow^- t_1 \mid \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1 \vdash A \Rightarrow^- t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^- [(x_1\, t_2)/x_2]t_1 \mid \Delta_2}\ \multimap^-_L$$

By induction, we have that:

$$(\Gamma, x_2 : B) - \Delta_1 \vdash t_1 : C \tag{ih1}$$
$$\Delta_1 - \Delta_2 \vdash t_2 : A \tag{ih2}$$

By the definition of context subtraction and since $x_2 \notin |\Delta_1|$ then (ih1) is equal to:

$$(\Gamma - \Delta_1), x_2 : B \vdash t_1 : C \tag{ih1'}$$

We can thus construct the following typing derivation, making use of of the admissibility of linear substitution (Lemma 4.1.1):

$$\dfrac{\dfrac{(\Gamma - \Delta_1), x_2 : B \multimap C \vdash t_1 : C}{\Gamma - \Delta_1 \vdash \lambda x_2.t_1 : B \multimap C}\ \textsc{abs} \qquad \dfrac{\dfrac{}{x_1 : A \multimap B \vdash x_1 : A \multimap B}\ \textsc{var} \qquad \Delta_1 - \Delta_2 \vdash t_2 : A}{(\Delta_1 - \Delta_2), x_1 : A \multimap B \vdash x_1\, t_2 : B}\ \textsc{app}}{(\Gamma - \Delta_1) + (\Delta_1 - \Delta_2), x_1 : A \multimap B \vdash [(x_1\, t_2)/x_2]t_1 : C}\ \textsc{app}$$

From Lemma B.1.1, we have that

$$((\Gamma - \Delta_1) + (\Delta_1 - \Delta_2)), x_1 : A \multimap B \sqsubseteq (((\Gamma - \Delta_1) + \Delta_1) - \Delta_2), x_1 : A \multimap B$$

and from Lemma B.1.2, that:

$$(((\Gamma - \Delta_1) + \Delta_1) - \Delta_2), x_1 : A \multimap B \sqsubseteq (\Gamma - \Delta_2), x_1 : A \multimap B$$

which, since x_1 is not in Δ_2 (as x_1 is not in Γ) $(\Gamma - \Delta_2), x_1 : A \multimap B = (\Gamma, x_1 : A \multimap B) - \Delta_2$. Applying these inequalities with APPROX then yields the lemma's conclusion $(\Gamma, x_1 : A \multimap B) - \Delta_2 \vdash [(x_1\, t_2)/x_2]t_1 : C$.

5. Case $\Box^-_R$
The synthesis rule for boxing can be constructed as:

$$\frac{\Gamma \vdash A \Rightarrow^- t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^- [t] \mid \Gamma - r \cdot (\Gamma - \Delta)}\ \Box^-_R$$

By induction on the premise we get:

$$\Gamma - \Delta \vdash t : A$$

Since we apply scalar multipication ih the conclusion of the rule to $\Gamma - \Delta$ then we know that all of $\Gamma - \Delta$ must be graded assumptions.

From this, we can construct the typing derivation:

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

Via Lemma B.1.4, we then have that $(r \cdot \Gamma - \Delta) \sqsubseteq (\Gamma - (\Gamma - (r \cdot (\Gamma - \Delta))))$ thus, we can derived:

$$\frac{\dfrac{[\Gamma - \Delta] \vdash t : A}{r \cdot [\Gamma - \Delta] \vdash [t] : \Box_r A \quad \text{Lem. B.1.4}}\ \textsc{Pr}}{\Gamma - (\Gamma - (r \cdot (\Gamma - \Delta))) \vdash [t] : \Box_r A}\ \textsc{Approx}$$

Satisfying the goal of the lemma.

6. Case $\Box^-_L$
The synthesis rule for unboxing has the form:

$$\frac{\Gamma, x_2 :_r A \vdash B \Rightarrow^- t \mid \Delta, x_2 :_s A \qquad 0 \sqsubseteq s}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^- \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid \Delta}\ \Box^-_L$$

By induction on the premise we have that:

$$(\Gamma, x_2 :_r A) - (\Delta, x_2 :_s A) \vdash t : B$$

By the definition of context subtraction we get that $\exists q$ and:

$$(\Gamma, x_2 :_r A) - (\Delta, x_2 :_s A) = (\Gamma - \Delta), x_2 :_q A$$

such that $r = q + s$

We also have that $0 \sqsubseteq s$.

By monotonicity with $q \sqsubseteq q$ (reflexivity) and $0 \sqsubseteq s$ then $q \sqsubseteq q + s$.

By context subtraction we have $r = q + s$ therefore $q \sqsubseteq r$ (*).

From this, we can construct the typing derivation:

$$\dfrac{\dfrac{\dfrac{}{x_1 : \Box_r A \vdash x_1 : \Box_r A}\ \textsc{Var}}{(\Gamma - \Delta), x_2 :_q A \vdash t : B \quad (*)}\ \textsc{approx}}{(\Gamma - \Delta), x_1 : \Box_r A \vdash \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t : B}\ \textsc{Let}$$

Which matches the goal.

7. Case $\otimes_R^-$

The synthesis rule for pair introduction has the form:

$$\frac{\Gamma \vdash A \Rightarrow^- t_1 \mid \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow^- t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^- (t_1, t_2) \mid \Delta_2}\ \otimes_R^-$$

By induction we get:

$$\Gamma - \Delta_1 \vdash t_1 : A \tag{ih1}$$
$$\Delta_1 - \Delta_2 \vdash t_2 : B \tag{ih2}$$

From this, we can construct the typing derivation:

$$\frac{\Gamma - \Delta_1 \vdash t_1 : A \qquad \Delta_1 - \Delta_2 \vdash t_2 : B}{(\Gamma - \Delta_1) + (\Delta_1 - \Delta_2) \vdash (t_1, t_2) : A \otimes B}\ \textsc{Pair}$$

From Lemma B.1.1, we have that:

$$(\Gamma - \Delta_1) + (\Delta_1 - \Delta_2) \sqsubseteq ((\Gamma - \Delta_1) + \Delta_1) - \Delta_2$$

and from Lemma B.1.2, that:

$$((\Gamma - \Delta_1) + \Delta_1) - \Delta_2 \sqsubseteq \Gamma - \Delta_2$$

From which we then apply Approx to the above derivation, yielding the goal $\Gamma - \Delta_2 \vdash (t_1, t_2) : A \otimes B$.

8. Case $\otimes_L^-$
   The synthesis rule for pair elimination has the form:

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^- t_2 \mid \Delta \qquad x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^- \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta}\ \otimes_L^-$$

   By induction we get:

$$(\Gamma, x_1 : A, x_2 : B) - \Delta \vdash t_2 : C$$

   since $x_1 \notin |\Delta| \wedge x_2 \notin |\Delta|$ then $(\Gamma, x_1 : A, x_2 : B) - \Delta = (\Gamma - \Delta), x_1 : A, x_2 : B$.

   From this, we can construct the following typing derivation, matching the conclusion:

$$\frac{\dfrac{}{x_3 : A \otimes B \vdash x_3 : A \otimes B}\ \textsc{Var} \qquad (\Gamma - \Delta), x_1 : A, x_2 : B \vdash t_2 : C}{(\Gamma - \Delta), x_3 : A \otimes B \vdash \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 : C}\ \textsc{Case}$$

   which matches the conclusion since $(\Gamma - \Delta), x_3 : A \otimes B = (\Gamma, x_3 : A \otimes B) - \Delta$ since $x_3 \notin |\Delta|$ by its disjointness from Γ.

9. Case $\oplus 1_R^-$ and $\oplus 2_R^-$
   The synthesis rules for sum introduction are straightforward. For $\oplus 1_R^-$ we have the rule:

$$\frac{\Gamma \vdash A \Rightarrow^- t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^- \mathbf{inl}\,t \mid \Delta}\ \oplus 1_R^-$$

   By induction we have:

$$\Gamma - \Delta \vdash t : A \qquad \text{(ih1)}$$

   from which we can construct the typing derivation, matching the conclusion:

$$\frac{\Gamma - \Delta \vdash t : A}{\Gamma - \Delta \vdash \mathbf{inl}\,t : A \oplus B}\ \oplus 1_R^-$$

   Matching the goal. And likewise for $\oplus 2_R^-$.

10. Case $\oplus_L^-$ The synthesis rule for sum elimination has the form:

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow^- t_1 \mid \Delta_1 \quad \Gamma, x_3 : B \vdash C \Rightarrow^- t_2 \mid \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2|}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^- \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcap \Delta_2}\ \oplus_L^-$$

   By induction:

$$(\Gamma, x_2 : A) - \Delta_1 \vdash t_1 : C \qquad \text{(ih)}$$
$$(\Gamma, x_3 : B) - \Delta_2 \vdash t_2 : C \qquad \text{(ih)}$$

From this we can construct the typing derivation, matching the conclusion:

$$\frac{}{x_1 : A \oplus B \vdash t_1 : A \oplus B}\ \textsc{Var} \qquad \frac{(\Gamma - \Delta_1), x_2 : A \vdash t_2 : C \qquad (\Gamma - \Delta_2), x_3 : B \vdash t_3 : C}{(\Gamma, x_1 : A \oplus B) - (\Delta_1 \sqcap \Delta_2) \vdash \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 : C}\ \textsc{Case}$$

11. Case Unit^-_R

$$\frac{}{\Gamma \vdash \mathsf{Unit} \Rightarrow^- () \mid \Gamma}\ \mathsf{Unit}^-_R$$

By Lemma B.1.3 we have that $\emptyset \sqsubseteq \Gamma - \Gamma$ then we have:

$$\frac{\dfrac{}{\emptyset \vdash () : \mathsf{Unit}}\ \mathsf{Unit}}{\Gamma - \Gamma \vdash () : \mathsf{Unit}}\ \textsc{Approx}$$

Matching the goal

12. Case Unit^-_L

$$\frac{\Gamma \vdash C \Rightarrow^- t \mid \Delta}{\Gamma, x : \mathsf{Unit} \vdash C \Rightarrow^- \mathbf{let}\,() = x\,\mathbf{in}\,t \mid \Delta}\ \mathsf{Unit}^-_L$$

By induction we have:

$$\Gamma - \Delta \vdash t : C \qquad \text{(ih)}$$

Then we make the derivation:

$$\frac{\dfrac{}{x : \mathsf{Unit} \vdash x : \mathsf{Unit}}\ \textsc{Var} \qquad \Gamma - \Delta \vdash t : C}{(\Gamma - \Delta), x : \mathsf{Unit} \vdash \mathbf{let}\,() = x\,\mathbf{in}\,t : C}\ \textsc{LetUnit}$$

where the context is equal to $(\Gamma, x : \mathsf{Unit}) - \Delta$.

13. Case $\textsc{der}^-$

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^- t \mid \Delta, x :_{s'} A \qquad y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s + 1}{\Gamma, x :_r A \vdash B \Rightarrow^- [x/y]t \mid \Delta, x :_{s'} A}\ \textsc{der}^-$$

By induction:

$$(\Gamma, x :_s A, y : A) - (\Delta, x :_{s'} A) \vdash t : B \qquad \text{(ih)}$$

By the definition of context subtraction we have (since also $y \notin |\Delta|$)

$$\begin{aligned}&(\Gamma, x :_s A, y : A) - (\Delta, x :_{s'} A)\\ =\ &(\Gamma - \Delta), x :_q A, y : A\end{aligned}$$

where $\exists q. s \sqsupseteq q + s'$ (1) and $\forall \hat{q}. s \sqsupseteq \hat{q} + s' \implies q \sqsupseteq \hat{q}$ (2)

The goal context is computed by:

$$\begin{aligned}&(\Gamma, x :_r A) - (\Delta, x :_{s'} A)\\ =\ &(\Gamma - \Delta), x :_{q'} A\end{aligned}$$

where $r \sqsupseteq q' + s'$ (3) and $\forall \hat{q}'. r \sqsupseteq \hat{q}' + s' \implies q' \sqsupseteq \hat{q}'$ (4)

From the premise of DER$^-$we have $r \sqsupseteq (s + 1)$.

$$\begin{aligned}\text{congruence of + and (1)} &\implies s + 1 \sqsupseteq q + s' + 1 && (5)\\ \text{transitivity with DER}^-\text{premise and (5)} &\implies r \sqsupseteq q + s' + 1 && (6)\\ \text{+ assoc./comm. on (6)} &\implies r \sqsupseteq q + 1 + s' && (7)\\ \text{apply (8) to (4) with } \hat{q}' = q + 1 &\implies q' \sqsupseteq q + 1 && (8)\end{aligned}$$

Using this last result we derive:

$$\dfrac{\dfrac{\dfrac{(\Gamma - \Delta), x :_q A, y : A \vdash t : B}{(\Gamma - \Delta), x :_q A, y :_1 A \vdash t : B}\ \textsc{Der}}{(\Gamma - \Delta), x :_{q+1} A \vdash [x/y]t : B}\ \textsc{contraction} \quad (8)}{(\Gamma - \Delta), x :_{q'} A \vdash [x/y]t : B}\ \textsc{approx}$$

Which matches the goal.

□

B.1.2 *Soundness of the Additive Graded Linear Typing Calculus*

Lemma 3.4.1 (Additive synthesis soundness)**.** Given a particular pre-ordered semiring $\mathcal{R}$ parametrising the calculi, then, for all contexts Γ and Δ, types A and terms t:

$$\Gamma \vdash A \Rightarrow^+ t \mid \Delta \quad \implies \quad \Delta \vdash t : A$$

Proof. 1. Case LINVAR$^+$

In the case of linear variable synthesis, we have the derivation:

$$\dfrac{}{\Gamma, x : A \vdash A \Rightarrow^+ x \mid x : A}\ \textsc{LinVar}^+$$

Therefore we can construct the following typing derivation, matching the conclusion:

$$\frac{}{x : A \vdash x : A}\ \textsc{var}$$

2. Case GrVar$^+$
Matching the form of the lemma, we have the derivation:

$$\frac{}{\Gamma, x :_r A \vdash A \Rightarrow^+ x \mid x :_1 A}\ \textsc{GrVar}^+$$

From this we can construct the typing derivation, matching the conclusion:

$$\frac{\dfrac{}{x : A \vdash x : A}\ \textsc{Var}}{x :_1 A \vdash x : A}\ \textsc{Der}$$

3. Case $\multimap^+_R$
We thus have the derivation:

$$\frac{\Gamma, x : A \vdash B \Rightarrow^+ t \mid \Delta, x : A}{\Gamma \vdash A \multimap B \Rightarrow^+ \lambda x.t \mid \Delta}\ \multimap^+_R$$

By induction on the premise we then have:

$$\Delta, x : A \vdash t : B$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\Delta, x : A \vdash t : B}{\Delta \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

4. Case $\multimap^+_L$
Matching the form of the lemma, the application derivation can be constructed as:

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1 \mid \Delta_1, x_2 : B \qquad \Gamma \vdash A \Rightarrow^+ t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^+ [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap^+_L$$

By induction on the premises we then have the following typing judgments:

$$\Delta_1, x_2 : B \vdash t_1 : C$$
$$\Delta_2 \vdash t_2 : A$$

We can thus construct the following typing derivation, making use of the admissibility of linear substitution (Lemma 4.1.1):

$$\dfrac{\dfrac{}{x_1 : A \multimap B \vdash x_1 : A \multimap B}\,\textsc{var} \qquad \Delta_2 \vdash t_2 : A}{\Delta_2, x_1 : A \multimap B \vdash x_1\, t_2 : B}\,\textsc{app} \qquad \Delta_1, x_2 : B \vdash t_1 : C}{(\Delta_1 + \Delta_2), x_1 : A \multimap B \vdash [(x_1\, t_2)/x_2]t_1 : C}\,(\text{L. } 4.1.1)$$

5. Case $\Box^+_R$
   The synthesis rule for boxing can be constructed as:

$$\dfrac{\Gamma \vdash A \Rightarrow^+ t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^+ [t] \mid r \cdot \Delta}\,\Box^+_R$$

   By induction we then have:

$$\Delta \vdash t : A$$

   In the conclusion of the above derivation we know that $r \cdot \Delta$ is defined, therefore it must be that all of Δ are graded assumptions, i.e., we have that $[\Delta]$ holds. We can thus construct the following typing derivation, matching the conclusion:

$$\dfrac{[\Delta] \vdash t : A}{r \cdot [\Delta] \vdash [t] : \Box_r A}\,\textsc{Pr}$$

6. Case $\textsc{der}^+$
   From the dereliction rule we have:

$$\dfrac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^+ t \mid \Delta, y : A}{\Gamma, x :_s A \vdash B \Rightarrow^+ [x/y]t \mid \Delta + x :_1 A}\,\textsc{der}^+$$

   By induction we get:

$$\Delta, y : A \vdash t : B \qquad \text{(ih)}$$

   Case on $x \in \Delta$

   - $x \in \Delta$, i.e., $\Delta = \Delta', x :_{s'} A$.

     Then by admissibility of contraction we can derive:

$$\dfrac{\dfrac{\Delta', x :_{s'} A, y : A \vdash t : B}{\Delta', x :_{s'} A, y :_1 A \vdash t : B}\,\textsc{Der}}{(\Delta', x :_{s'} A) + x :_1 A \vdash [x/y]t : B}$$

     Satisfying the lemma statment.

- $x \notin \Delta$. Then again from the admissiblity of contraction, we derive the typing:

$$\dfrac{\dfrac{\Delta, y : A \vdash t : B}{\Delta, y :_1 A \vdash t : B}\,\textsc{Der}}{\Delta + x :_1 A \vdash [x/y]t : B}$$

  which is well defined as $x \notin \Delta$ and gives the lemma conclusion.

7. Case $\Box_L^+$
   The synthesis rule for unboxing has the form:

$$\dfrac{\Gamma, x_2 :_r A \vdash B \Rightarrow^+ t \mid \Delta \qquad \textit{if } x_2 :_s A \in \Delta \textit{ then } s \sqsubseteq r \textit{ else } 0 \sqsubseteq r}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow^+ \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid (\Delta \backslash x_2), x_1 : \Box_r A}\,\Box_L^+$$

   By induction we have that:

$$\Delta \vdash t : B \tag{ih}$$

   Case on $x_2 :_s A \in \Delta$

   - $x_2 :_s A \in \Delta$, i.e., $s \sqsubseteq r$.
     From this, we can construct the typing derivation, matching the conclusion:

$$\dfrac{\dfrac{}{x_1 : \Box_r A \vdash x_1 : \Box_r A}\,\textsc{var} \qquad \Delta, x_2 :_r A \vdash t : B}{\Delta, x_1 : \Box_r A \vdash \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t : B}\,\textsc{let}\Box$$

   - $x_2 :_s A \notin \Delta$, i.e., $0 \sqsubseteq r$.
     From this, we can construct the typing derivation, matching the conclusion:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x_1 : \Box_r A \vdash x_1 : \Box_r A}\,\textsc{var}}{\Delta \vdash t : B}\,\textsc{Weak} \qquad 0 \sqsubseteq r}{\Delta, x_2 :_r A \vdash t : B}\,\textsc{Approx}}{\Delta, x_1 : \Box_r A \vdash \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t : B}\,\textsc{let}\Box$$

8. Case $\otimes_R^+$

   The synthesis rule for pair introduction has the form:

$$\dfrac{\Gamma \vdash A \Rightarrow^+ t_1 \mid \Delta_1 \qquad \Gamma \vdash B \Rightarrow^+ t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^+ (t_1, t_2) \mid \Delta_1 + \Delta_2}\,\otimes_R^+$$

By induction on the premises we have that:

$$\Delta_1 \vdash t_1 : A \tag{ih1}$$
$$\Delta_2 \vdash t_2 : B \tag{ih2}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\Delta_1 \vdash t_1 : A \qquad \Delta_2 \vdash t_2 : B}{\Delta_1 + \Delta_2 \vdash (t_1, t_2) : A \otimes B}\ \textsc{pair}$$

9. Case $\otimes_L^+$
The synthesis rule for pair elimination has the form:

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow^+ t_2 \mid \Delta, x_1 : A, x_2 : B}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow^+ \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta, x_3 : A \otimes B}\ \otimes_L^+$$

By induction on the premises we have that:

$$\Delta_1 \vdash t_1 : A \tag{ih1}$$
$$\Delta_2 \vdash t_2 : B \tag{ih2}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\dfrac{}{x_3 : A \otimes B \vdash x_3 : A \otimes B}\ \textsc{Var} \qquad \Delta, x_1 : A, x_2 : B \vdash t_2 : C}{\Delta, x_3 : A \otimes B \vdash \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 : C}\ \textsc{LetPair}$$

10. Case $\oplus 1_R^+$ and $\oplus 2_R^+$
The synthesis rules for sum introduction are straightforward. For $\oplus 1_R^+$ we have the rule:

$$\frac{\Gamma \vdash A \Rightarrow^+ t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inl}\,t \mid \Delta}\ \oplus 1_R^+$$

By induction on the premises we have that:

$$\Delta \vdash t : A \tag{ih}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\Delta \vdash t : A}{\Delta \vdash \mathbf{inl}\,t : A \oplus B}\ \textsc{Inl}$$

Likewise, for the $\oplus 2_R^+$ we have the synthesis rule:

$$\frac{\Gamma \vdash B \Rightarrow^+ t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inr}\,t \mid \Delta}\ \oplus 2_R^+$$

By induction on the premises we have that:

$$\Delta \vdash t : B \tag{ih}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\Delta \vdash t : B}{\Delta \vdash \mathbf{inl}\, t : A \oplus B}\ \textsc{Inr}$$

11. Case $\oplus_L^+$
The synthesis rule for sum elimination has the form:

$$\frac{\begin{array}{c}\Gamma, x_2 : A \vdash C \Rightarrow^+ t_1 \mid \Delta_1, x_2 : A \\ \Gamma, x_3 : B \vdash C \Rightarrow^+ t_2 \mid \Delta_2, x_3 : B\end{array}}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow^- \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcup \Delta_2, x_1 : A \oplus B}\ \oplus_L^+$$

By induction on the premises we have that:

$$\Delta_1, x_2 : A \vdash t_1 : C \tag{ih1}$$
$$\Delta_2, x_3 : B \vdash t_2 : C \tag{ih2}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\dfrac{}{x_1 : A \oplus B \vdash x_1 : A \oplus B}\ \textsc{Var} \qquad \Delta_1, x_2 : A \vdash t_1 : C \qquad \Delta_2, x_3 : B \vdash t_2 : C}{(\Delta_1 \sqcup \Delta_2), x_1 : A \oplus B \vdash \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 : C}\ \textsc{Case}$$

12. Case Unit_R^+
The synthesis rule for unit introduction has the form:

$$\frac{}{\Gamma \vdash \mathsf{Unit} \Rightarrow^+ () \mid \varnothing}\ \mathsf{Unit}_R^+$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{}{\varnothing \vdash () : \mathsf{Unit}}\ \mathsf{Unit}$$

13. Case Unit_L^+
The synthesis rule for unit elimination has the form:

$$\frac{\Gamma \vdash C \Rightarrow^+ t \mid \Delta}{\Gamma, x : \mathsf{Unit} \vdash C \Rightarrow^+ \mathbf{let}\,() = x\,\mathbf{in}\, t \mid \Delta, x : \mathsf{Unit}}\ \mathsf{Unit}_L^+$$

By induction on the premises we have that:

$$\Delta \vdash t : C \tag{ih}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\dfrac{\dfrac{}{x : \mathsf{Unit} \vdash x : \mathsf{Unit}}\ \textsc{Var} \qquad \Delta \vdash t : C}{\Delta, x : \mathsf{Unit} \vdash \mathbf{let}\,() = x\,\mathbf{in}\,t : C}\ \textsc{LetUnit}$$

□

B.1.3 *Soundness of the Additive Pruning Graded Linear Typing Calculus*

Lemma 3.4.2 (Additive pruning synthesis soundness)**.** For all Γ and A:

$$\Gamma \vdash A \Rightarrow^{\pm} t \mid \Delta \quad \Longrightarrow \quad \Delta \vdash t : A$$

Proof. The cases for the rules in the additive pruning synthesis calculus are equivalent to lemma (3.4.1), except for the cases of the $\multimap^{\pm}_{L}$ and $\otimes^{\pm}_{R}$ rules which we consider here:

1. Case $\multimap^{\pm}_{L}$
   Matching the form of the lemma, the application derivation can be constructed as:

   $$\dfrac{\Gamma, x_2 : B \vdash C \Rightarrow^{\pm} t_1 \mid \Delta_1, x_2 : B \qquad \Gamma - \Delta_1 \vdash \Rightarrow^{\pm} t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{\pm} [(x_1\ t_2)/x_2] \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap^{\pm}_{L}$$

   By induction on the premises we then have the following typing judgments:

   $$\begin{aligned} \Delta_1, x_2 : B &\vdash t_1 : C \\ \Delta_2 &\vdash t_2 : A \end{aligned}$$

   We can thus construct the following typing derivation, making use of the admissibility of linear substitution (Lemma 4.1.1):

   $$\dfrac{\dfrac{\dfrac{}{x_1 : A \multimap B \vdash x_1 : A \multimap B}\ \textsc{var} \qquad \Delta_2 \vdash t_2 : A}{\Delta_2, x_1 : A \multimap B \vdash x_1\,t_2 : B}\ \textsc{app} \qquad \Delta_1, x_2 : B \vdash t_1 : C}{(\Delta_1 + \Delta_2), x_1 : A \multimap B \vdash [(x_1\,t_2)/x_2]t_1 : C}\ (\text{L. } 4.1.1)$$

2. Case $\otimes^{\pm}_{R}$

   The synthesis rule for the pruning alternative for pair introduction has the form:

$$\frac{\Gamma \vdash A \Rightarrow^{\pm} t_1 \mid \Delta_1 \qquad \Gamma - \Delta_1 \vdash B \Rightarrow^{\pm} t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{\pm} (t_1, t_2) \mid \Delta_1 + \Delta_2} \otimes_R^{\pm}$$

By induction on the premises we have that:

$$\Delta_1 \vdash t_1 : A \tag{ih1}$$
$$\Delta_2 \vdash t_2 : B \tag{ih2}$$

From this, we can construct the typing derivation, matching the conclusion:

$$\frac{\Delta_1 \vdash t_1 : A \qquad \Delta_2 \vdash t_2 : B}{\Delta_1 + \Delta_2 \vdash (t_1, t_2) : A \otimes B} \textsc{pair}$$

□

B.1.4 *Soundness of Focusing for the Subtractive Linear Graded Synthesis Calculus*

Lemma 3.5.1 (Soundness of focusing for subtractive synthesis)**.** For all contexts Γ, Ω and types A, B then:

1. Right Async:	$\Gamma; \Omega \vdash A \Uparrow \Rightarrow^{-} t \mid \Delta$	$\iff$	$\Gamma, \Omega \vdash A \Rightarrow^{-} t \mid \Delta$
2. Left Async:	$\Gamma; \Omega \Uparrow \vdash B \Rightarrow^{-} t \mid \Delta$	$\iff$	$\Gamma, \Omega \vdash B \Rightarrow^{-} t \mid \Delta$
3. Right Sync:	$\Gamma; \emptyset \vdash A \Downarrow \Rightarrow^{-} t \mid \Delta$	$\iff$	$\Gamma \vdash A \Rightarrow^{-} t \mid \Delta$
4. Left Sync:	$\Gamma; x : A \Downarrow \vdash B \Rightarrow^{-} t \mid \Delta$	$\iff$	$\Gamma, x : A \vdash B \Rightarrow^{-} t \mid \Delta$
5. Focus Right:	$\Gamma; \emptyset \Uparrow \vdash B \Rightarrow^{-} t \mid \Delta$	$\iff$	$\Gamma \vdash B \Rightarrow^{-} t \mid \Delta$
6. Focus Left:	$\Gamma, x : A; \emptyset \Uparrow \vdash C \Rightarrow^{-} t \mid \Delta$	$\iff$	$\Gamma, x : A \vdash B \Rightarrow^{-} t \mid \Delta$

i.e. t has type A under context Δ, which contains assumptions with grades reflecting their use in t.

Proof. 1. Case 1. Right Async:

a) Case $\multimap_R^-$
In the case of the right asynchronous rule for abstraction introduction, the synthesis rule has the form:

$$\frac{\Gamma; \Omega, x : A \vdash C \Uparrow \Rightarrow^{-} t \mid \Delta \quad x \notin |\Delta|}{\Gamma; \Omega \vdash A \multimap B \Uparrow \Rightarrow^{-} \lambda x.t \mid \Delta} \multimap_R^-$$

By induction on the first premise, we have that:

$$(\Gamma, \Omega), x : A \vdash A \Rightarrow^{-} t \mid \Delta \tag{ih}$$

from case 1 of the lemma. From which, we can construct the following instantiation of the $\multimap^-_R$ synthesis rule in the non-focusing calculus:

$$\frac{(\Gamma, \Omega), x : A \vdash B \Rightarrow^- t \mid \Delta \quad x \notin |\Delta|}{\Gamma, \Omega \vdash A \multimap B \Rightarrow^- \lambda x.t \mid \Delta} \multimap^-_R$$

b) Case $\Uparrow^-_R$
In the case of the right asynchronous rule for transition to a left asynchronous judgement, the synthesis rule has the form:

$$\frac{\Gamma; \Omega \Uparrow \vdash C \Rightarrow^- t \mid \Delta \qquad C \text{ not RIGHT ASYNC}}{\Gamma; \Omega \vdash C \Uparrow \Rightarrow^- t \mid \Delta} \Uparrow^-_R$$

By induction on the first premise, we have that:

$$\Gamma, \Omega \vdash C \Rightarrow^- t \mid \Delta$$

from case 2 of the lemma.

2. Case 2. Left Async:

a) Case $\otimes^-_L$
In the case of the left asynchronous rule for pair elimination, the synthesis rule has the form:

$$\frac{\begin{array}{c}\Gamma; \Omega, x_1 : A, x_2 : B \Uparrow \vdash C \Rightarrow^- t_2 \mid \Delta \\ x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|\end{array}}{\Gamma; \Omega, x_3 : A \otimes B \Uparrow \vdash C \Rightarrow^- \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta} \otimes^-_L$$

By induction on the first premise, we have that:

$$(\Gamma, \Omega), x_1 : A, x_2 : B \vdash C \Rightarrow^- t \mid \Delta \qquad \text{(ih)}$$

from From which, we can construct the following instantiation of the $\otimes^-_R$ synthesis rule in the non-focusing calculus:

$$\frac{\begin{array}{c}(\Gamma, \Omega), x_1 : A, x_2 : B \vdash C \Rightarrow^- t \mid \Delta \\ x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|\end{array}}{\Gamma, (\Omega, x_3 : A \otimes B) \vdash C \Rightarrow^- \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t \mid \Delta_2} \otimes^-_L$$

b) Case $\oplus^-_L$
In the case of the left asynchronous rule for sum elimination, the synthesis rule has the form:

$$\frac{\Gamma; \Omega, x_2 : A \Uparrow \vdash C \Rightarrow^- t_1 \mid \Delta_1 \quad \Gamma; \Omega, x_3 : B \Uparrow \vdash C \Rightarrow^- t_2 \mid \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2|}{\Gamma; \Omega, x_1 : A \oplus B \Uparrow \vdash C \Rightarrow^- \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcap \Delta_2} \oplus^-_L$$

By induction on the first and second premises, we have that:

$$(\Gamma, \Omega), x_2 : A \vdash C \Rightarrow^- t_1 \mid \Delta_1 \tag{ih1}$$

$$(\Gamma, \Omega), x_3 : B \vdash C \Rightarrow^- t_2 \mid \Delta_2 \tag{ih2}$$

from case 2 of the lemma. From which, we can construct the following instantiation of the $\oplus_L^-$ synthesis rule in the non-focusing calculus:

$$\frac{(\Gamma, \Omega), x_2 : A \vdash C \Rightarrow^- t_1 \mid \Delta_1 \quad (\Gamma, \Omega), x_3 : B \vdash C \Rightarrow^- t_2 \mid \Delta_2 \quad x_2 \notin |\Delta_1| \quad x_3 \notin |\Delta_2|}{\Gamma, (\Omega, x_1 : A \oplus B) \vdash C \Rightarrow^- \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \Delta_1 \sqcap \Delta_2}\ \oplus_L^-$$

c) Case Unit_L^-
In the case of the left asynchronous rule for unit elimination, the synthesis rule has the form:

$$\frac{\Gamma; \varnothing \vdash C \Rightarrow^- t \mid \Delta}{\Gamma; x : \mathsf{Unit} \vdash C \Rightarrow^- \mathbf{let}\,() = x\,\mathbf{in}\,t \mid \Delta}\ \mathsf{Unit}_L^-$$

By induction on the premise, we have that:

$$\Gamma \vdash C \Rightarrow^- t \mid \Delta \tag{ih}$$

from case 2 of the lemma. From which, we can construct the following instantiation of the Unit_L^- synthesis rule in the non-focusing calculus matching the conclusion:

$$\frac{\Gamma \vdash C \Rightarrow^- t \mid \Delta}{\Gamma, x : \mathsf{Unit} \vdash C \Rightarrow^- \mathbf{let}\,() = x\,\mathbf{in}\,t \mid \Delta}\ \mathsf{Unit}_L^-$$

d) Case $\Box_L^-$
In the case of the left asynchronous rule for graded modality elimination, the synthesis rule has the form:

$$\frac{\Gamma; \Omega, x_2 :_r A \Uparrow \vdash B \Rightarrow^- t \mid \Delta, x_2 :_s A \qquad 0 \sqsubseteq s}{\Gamma; \Omega, x_1 : \Box_r A \Uparrow \vdash B \Rightarrow^- \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid \Delta}\ \Box_L^-$$

By induction on the first premise, we have that:

$$(\Gamma, \Omega), x_2 :_r A \vdash B \Rightarrow^- t \mid \Delta, x_2 :_s A \tag{ih}$$

from case 2 of the lemma. From which, we can construct the following instantiation of the $\Box_L^-$synthesis rule in the non-focusing calculus:

$$\frac{(\Gamma, \Omega), x_2 :_r A \vdash B \Rightarrow^- t \mid \Delta, x_2 :_s A \qquad 0 \sqsubseteq s}{\Gamma, (\Omega, x_1 : \Box_r A) \vdash B \Rightarrow^- \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid \Delta}\ \Box_L^-$$

e) Case $\Uparrow_L^-$
In the case of the left asynchronous rule for transitioning an assumption from the focusing context Ω to the non-focusing context Γ, the synthesis rule has the form:

$$\frac{\Gamma, x : A; \Omega \Uparrow \vdash C \Rightarrow^- t \mid \Delta \qquad A \text{ not } \textsc{Left Async}}{\Gamma; \Omega, x : A \Uparrow \vdash C \Rightarrow^- t \mid \Delta} \Uparrow_L^-$$

By induction on the first premise, we have that:

$$\Gamma, x : A, \Omega \vdash C \Rightarrow^- t \mid \Delta \tag{ih}$$

from case 2 of the lemma.

3. Case 3. Right Sync:

a) Case $\otimes_R^-$
In the case of the right synchronous rule for pair introduction, the synthesis rule has the form:

$$\frac{\Gamma; \cdot \vdash A \Downarrow \Rightarrow^- t_1 \mid \Delta_1 \qquad \Delta_1; \cdot \vdash B \Downarrow \Rightarrow^- t_2 \mid \Delta_2}{\Gamma; \cdot \vdash A \otimes B \Downarrow \Rightarrow^- (t_1, t_2) \mid \Delta_2} \otimes_R^-$$

By induction on the first and second premises, we have that:

$$\Gamma \vdash A \Rightarrow^- t_1 \mid \Delta_1 \tag{ih1}$$

$$\Delta_1 \vdash B \Rightarrow^- t_2 \mid \Delta_2 \tag{ih2}$$

from case 3 of the lemma. From which, we can construct the following instantiation of the $\otimes_R^-$ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma \vdash A \Rightarrow^- t_1 \mid \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow^- t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^- (t_1, t_2) \mid \Delta_2} \otimes_R^-$$

b) Case $\oplus 1_R^-$ and $\oplus 2_R^-$
In the case of the right synchronous rules for sum introduction, the synthesis rules has the form:

$$\frac{\Gamma; \cdot \vdash A \Downarrow \Rightarrow^- t \mid \Delta}{\Gamma; \cdot \vdash A \oplus B \Downarrow \Rightarrow^- \mathbf{inl}\, t \mid \Delta} \oplus 1_L^+$$

$$\frac{\Gamma; \cdot \vdash B \Downarrow \Rightarrow^- t \mid \Delta}{\Gamma; \cdot \vdash A \oplus B \Downarrow \Rightarrow^- \mathbf{inr}\, t \mid \Delta} \oplus 2_L^+$$

By induction on the premises of these rules, we have that:

$$\Gamma \vdash A \Rightarrow^- t \mid \Delta \tag{ih1}$$

$$\Gamma \vdash B \Rightarrow^{-} t \mid \Delta \tag{ih2}$$

from case 3 of the lemma. From which, we can construct the following instantiations of the $\oplus 1_R^-$ and $\oplus 2_R^-$ rule in the non-focusing calculus, respectively:

$$\frac{\Gamma \vdash A \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{-} \mathbf{inl}\, t \mid \Delta}\ \oplus 1_R^-$$

$$\frac{\Gamma \vdash B \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^{-} \mathbf{inr}\, t \mid \Delta}\ \oplus 2_R^-$$

c) Case Unit_R^-
In the case of the right synchronous rule for unit introduction, the synthesis rule has the form:

$$\frac{}{\Gamma; \varnothing \vdash \mathsf{Unit} \Downarrow \Rightarrow^{-} () \mid \Gamma}\ \mathsf{Unit}_R^-$$

From which, we can construct the following instantiation of the Unit_R^- synthesis rule in the non-focusing calculus:

$$\frac{}{\Gamma, \Omega \vdash \mathsf{Unit} \Rightarrow^{-} () \mid \Gamma}\ \mathsf{Unit}_R^-$$

d) Case $\Box_R^-$
In the case of the right synchronous rule for graded modality introduction, the synthesis rule has the form:

$$\frac{\Gamma; \varnothing \vdash A \Downarrow \Rightarrow^{-} t \mid \Delta}{\Gamma; \varnothing \vdash \Box_r A \Downarrow \Rightarrow^{-} t \mid \Gamma - r \cdot (\Gamma - \Delta)}\ \Box_R^-$$

By induction on the premise, we have that:

$$\Gamma \vdash A \Rightarrow^{-} t \mid \Delta \tag{ih}$$

from case 1 of the lemma. From which, we can construct the following instantiation of the $\Box_R^-$synthesis rule in the non-focusing calculus:

$$\frac{\Gamma \vdash A \Rightarrow^{-} t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^{-} [t] \mid \Gamma - r \cdot (\Gamma - \Delta)}\ \Box_R^-$$

e) Case $\Downarrow_R^-$
In the case of the right synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

$$\frac{\Gamma; \varnothing \vdash A \Uparrow \Rightarrow^{-} t \mid \Delta}{\Gamma; \varnothing \vdash A \Downarrow \Rightarrow^{-} t \mid \Delta}\ \Downarrow_R^-$$

By induction on the premise, we have that:

$$\Gamma \vdash A \Rightarrow^{-} t \mid \Delta \tag{ih}$$

from case 1 of the lemma.

4. Case 4. Left Sync

   a) Case $\multimap_L^-$
   In the case of the left synchronous rule for application, the synthesis rule has the form:

$$\frac{\Gamma; x_2 : B \Downarrow \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1; \emptyset \vdash A \Downarrow \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma; x_1 : A \multimap B \Downarrow \vdash C \Rightarrow^{-} [(x_1\, t_2)/x_2]t_1 \mid \Delta_2} \multimap_L^-$$

   By induction on the first premise, we have that:

$$\Gamma, x_2 : B \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \tag{ih1}$$

   from case 4 of the lemma. By induction on the third premise, we have that:

$$\Delta_1 \vdash A \Rightarrow^{-} t_2 \mid \Delta_2 \tag{ih2}$$

   from case 3 of the lemma. From which, we can construct the following instantiation of the $\multimap_L^-$ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{-} t_1 \mid \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1 \vdash A \Rightarrow^{-} t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{-} [(x_1\, t_2)/x_2]t_1 \mid \Delta_2} \multimap_L^-$$

   b) Case $\textsc{der}^-$
   In the case of the left asynchronous rule for dereliction, the synthesis rule has the form:

$$\frac{\begin{array}{c}\Gamma; x :_s A, y : A \Downarrow \vdash B \Rightarrow^{-} t \mid \Delta, x :_{s'} A \\ y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s + 1\end{array}}{\Gamma; x :_r A \Downarrow \vdash B \Rightarrow^{-} [x/y]t \mid \Delta, x :_{s'} A} \textsc{der}^-$$

   By induction on the first premise, we have that:

$$\Gamma, x :_s A, y : A \vdash B \Rightarrow^{-} t \mid \Delta, x :_{s'} A \tag{ih}$$

   from case 4 of the lemma. From which, we can construct the following instantiation of the $\textsc{der}^-$ synthesis rule in the non-focusing calculus:

$$\frac{\begin{array}{c}\Gamma, x :_s A, y : A \vdash B \Rightarrow^{-} t \mid \Delta, x :_{s'} A \\ y \notin |\Delta| \qquad \exists s.\, r \sqsupseteq s + 1\end{array}}{\Gamma, x :_r A \vdash B \Rightarrow^{-} [x/y]t \mid \Delta, x :_{s'} A} \textsc{der}^-$$

c) Case LinVar$^-$
In the case of the left synchronous rule for linear variable synthesis, the synthesis rule has the form:

$$\frac{}{\Gamma; x : A \Downarrow \vdash A \Rightarrow^- x \mid \Gamma}\ \textsc{LinVar}^-$$

From which, we can construct the following instantiation of the LinVar$^-$ synthesis rule in the non-focusing calculus:

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^- x \mid \Gamma}\ \textsc{LinVar}^-$$

d) Case GrVar$^-$
In the case of the left synchronous rule for graded variable synthesis, the synthesis rule has the form:

$$\frac{\exists s.\, r \sqsubseteq s + 1}{\Gamma; x :_r A \Downarrow \vdash A \Rightarrow^- x \mid \Gamma, x :_s A}\ \textsc{GrVar}^-$$

From which, we can construct the following instantiation of the GrVar$^-$ synthesis rule in the non-focusing calculus:

$$\frac{\exists s.\, r \sqsubseteq s + 1}{\Gamma, x :_r A \vdash A \Rightarrow^- x \mid \Gamma, x :_s A}\ \textsc{GrVar}^-$$

e) Case $\Downarrow_L^-$
In the case of the left synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

$$\frac{\begin{array}{c}\Gamma; x : A \Uparrow \vdash C \Rightarrow^- t \mid \Delta \\ A \text{ not atomic and not } \textsc{Left Sync}\end{array}}{\Gamma; x : A \Downarrow \vdash C \Rightarrow^- t \mid \Delta}\ \Downarrow_L^-$$

By induction on the premise, we have that:

$$\Gamma, x : A \vdash C \Rightarrow^- t \mid \Delta \qquad \text{(ih)}$$

from case 2 of the lemma.

5. Case 5. Focus Right: focus$_R^-$
In the case of the focusing rule for transitioning from a left asynchronous judgement to a right synchronous judgement, the synthesis rule has the form:

$$\frac{\Gamma; \varnothing \vdash C \Downarrow \Rightarrow^- t \mid \Delta \qquad C \text{ not atomic}}{\Gamma; \varnothing \Uparrow \vdash C \Rightarrow^- t \mid \Delta}\ \textsc{focus}_R^-$$

By induction on the first premise, we have that:

$$\Gamma \vdash C \Rightarrow^- t \mid \Delta \qquad \text{(ih)}$$

from case 2 of the lemma.

6. Case 6. Focus Left focus_L^-
   In the case of the focusing rule for transitioning from a left asynchronous judgement to a left synchronous judgement, the synthesis rule has the form:

$$\frac{\Gamma; x : A \Downarrow \vdash C \Rightarrow^- t \mid \Delta}{\Gamma, x : A; \varnothing \Uparrow \vdash C \Rightarrow^- t \mid \Delta}\ \text{FOCUS}_L^-$$

   By induction on the first premise, we have that:

$$\Gamma, x : A \vdash C \Rightarrow^- t \mid \Delta \qquad \text{(ih)}$$

   from case 2 of the lemma.

□

B.1.5 *Soundness of Focusing for the Additive Linear Graded Synthesis Calculus*

Lemma 3.5.2 (Soundness of focusing for additive synthesis)**.** For all contexts Γ, Ω and types A, B then:

1. RIGHT ASYNC:	$\Gamma; \Omega \vdash A \Uparrow \Rightarrow^+ t \mid \Delta$	$\iff$	$\Gamma, \Omega \vdash A \Rightarrow^+ t \mid \Delta$
2. LEFT ASYNC:	$\Gamma; \Omega \Uparrow \vdash A \Rightarrow^+ t \mid \Delta$	$\iff$	$\Gamma, \Omega \vdash B \Rightarrow^+ t \mid \Delta$
3. RIGHT SYNC:	$\Gamma; \varnothing \vdash A \Downarrow \Rightarrow^+ t \mid \Delta$	$\iff$	$\Gamma \vdash A \Rightarrow^+ t \mid \Delta$
4. LEFT SYNC:	$\Gamma; x : A \Downarrow \vdash B \Rightarrow^+ t \mid \Delta$	$\iff$	$\Gamma, x : A \vdash B \Rightarrow^+ t \mid \Delta$
5. FOCUS RIGHT:	$\Gamma; \varnothing \Uparrow \vdash A \Rightarrow^+ t \mid \Delta$	$\iff$	$\Gamma \vdash B \Rightarrow^+ t \mid \Delta$
6. FOCUS LEFT:	$\Gamma, x : A; \varnothing \Uparrow \vdash B \Uparrow \Rightarrow^+ t \mid \Delta$	$\iff$	$\Gamma, x : A \mid B \Rightarrow^+ t \mid \Delta$

i.e. t has type A under context Δ, which contains assumptions with grades reflecting their use in t.

Proof. 1. Case 1. Right Async:

a) Case $\multimap_R^+$
   In the case of the right asynchronous rule for abstraction introduction, the synthesis rule has the form:

$$\frac{\Gamma; \Omega, x : A \vdash B \Uparrow \Rightarrow t \mid \Delta, x : A}{\Gamma; \Omega \vdash A \multimap B \Uparrow \Rightarrow \lambda x.t \mid \Delta}\ \multimap_R^+$$

   By induction on the premise, we have that:

$$(\Gamma, \Omega), x : A \vdash B \Rightarrow^+ t \mid \Delta, x : A \qquad \text{(ih)}$$

   from case 1 of the lemma. From which, we can construct the following instantiation of the $\multimap_R^+$ synthesis rule in the non-focusing calculus:

$$\frac{(\Gamma, \Omega), x : A \vdash B \Rightarrow^+ t \mid \Delta, x : A}{\Gamma, \Omega \vdash A \multimap B \Rightarrow^+ \lambda x.t \mid \Delta}\ \multimap_{\text{R}}^+$$

b) Case $\Uparrow_R^+$ In the case of the right asynchronous rule for transition to a left asynchronous judgement, the synthesis rule has the form:

$$\frac{\Gamma; \Omega \Uparrow \vdash C \Rightarrow t \mid \Delta \qquad C \text{ not RIGHT ASYNC}}{\Gamma; \Omega \vdash C \Uparrow \Rightarrow t \mid \Delta} \Uparrow_R^+$$

By induction on the first premise, we have that:

$$\Gamma, \Omega \vdash C \Rightarrow^+ t \mid \Delta$$

from case 2 of the lemma.

2. Case 2. Left Async:

a) Case $\otimes_L^+$
In the case of the left asynchronous rule for pair elimination, the synthesis rule has the form:

$$\frac{\Gamma; \Omega, x_1 : A, x_2 : B \vdash C \Rightarrow t_2 \mid \Delta, x_1 : A, x_2 : B}{\Gamma; \Omega, x_3 : A \otimes B \vdash C \Rightarrow \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta, x_3 : A \otimes B} \otimes_L^+$$

By induction on the premise, we have that:

$$(\Gamma, \Omega), x_1 : A, x_2 : B \vdash C \Rightarrow^+ t_2 \mid \Delta, x_1 : A, x_2 : B \qquad \text{(ih)}$$

from case 2 of the lemma. From which, we can construct the following instantiation of the $\otimes_L^+$ synthesis rule in the non-focusing calculus:

$$\frac{(\Gamma, \Omega), x_1 : A, x_2 : B \vdash C \Rightarrow^+ t_2 \mid \Delta, x_1 : A, x_2 : B}{\Gamma, (\Omega, x_3 : A \otimes B) \vdash C \Rightarrow^+ \mathbf{let}\,(x_1, x_2) = x_3\,\mathbf{in}\,t_2 \mid \Delta, x_3 : A \otimes B} \otimes_{\mathrm{L}}^+$$

b) Case $\oplus_L^+$
In the case of the left asynchronous rule for sum elimination, the synthesis rule has the form:

$$\frac{\begin{array}{c}\Gamma; \Omega, x_2 : A \Uparrow \vdash C \Rightarrow t_1 \mid \Delta_1, x_2 : A \\ \Gamma; \Omega, x_3 : B \Uparrow \vdash C \Rightarrow t_2 \mid \Delta_2, x_3 : B\end{array}}{\Gamma; \Omega, x_1 : A \oplus B \Uparrow \vdash C \Rightarrow^- \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid \Delta_1 \sqcup \Delta_2, x_1 : A \oplus B} \oplus_L^+$$

By induction on the premises, we have that:

$$(\Gamma, \Omega), x_2 : A \vdash C \Rightarrow^+ t_1 \mid \Delta_1, x_2 : A \qquad \text{(ih1)}$$

$$(\Gamma, \Omega), x_3 : B \vdash C \Rightarrow^+ t_2 \mid \Delta_2, x_3 : B \qquad \text{(ih2)}$$

from case 2 of the lemma. From which, we can construct the following instantiation of the $\oplus_L^+$ synthesis rule in the non-focusing calculus:

$$\frac{\begin{array}{c}(\Gamma, \Omega), x_2 : A \vdash C \Rightarrow^+ t_1 \mid \Delta_1, x_2 : A \\ (\Gamma, \Omega), x_3 : B \vdash C \Rightarrow^+ t_2 \mid \Delta_2, x_3 : B\end{array}}{\Gamma, (\Omega, x_1 : A \oplus B) \vdash C \Rightarrow^+ \mathbf{case}\ x_1\ \mathbf{of\ inl}\ x_2 \to t_1;\ \mathbf{inr}\ x_3 \to t_2 \mid (\Delta_1 \sqcup \Delta_2), x_1 : A \oplus B} \oplus_{\mathrm{L}}^+$$

c) Case Unit_L^+
In the case of the left asynchronous rule for unit elimination, the synthesis rule has the form:

$$\frac{\Gamma; \varnothing \vdash C \Rightarrow t \mid \Delta}{\Gamma; x : \mathsf{Unit} \vdash C \Rightarrow \mathbf{let}\,() = x\,\mathbf{in}\,t \mid \Delta, x : \mathsf{Unit}}\ \mathsf{Unit}_L^+$$

By induction on the premise, we have that:

$$\Gamma \vdash C \Rightarrow^+ t \mid \Delta \qquad \text{(ih)}$$

from case 2 of the lemma. From which, we can construct the following instantiation of the Unit_L^+ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma \vdash C \Rightarrow^+ t \mid \Delta}{\Gamma, x : \mathsf{Unit} \vdash C \Rightarrow^+ \mathbf{let}\,() = x\,\mathbf{in}\,t \mid \Delta, x : \mathsf{Unit}}\ \mathsf{Unit}_{\mathrm{L}}^+$$

d) Case $\square_L^+$
In the case of the left asynchronous rule for graded modality elimination, the synthesis rule has the form:

$$\frac{\begin{array}{c}\Gamma; \Omega, x_2 :_r A \Uparrow \vdash B \Rightarrow t \mid \Delta \\ \mathit{if}\ x_2 :_s A \in \Delta\ \mathit{then}\ s \sqsubseteq r\ \mathit{else}\ 0 \sqsubseteq r\end{array}}{\Gamma; \Omega, x_1 : \square_r A \vdash B \Rightarrow \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid (\Delta \backslash x_2), x_1 : \square_r A}\ \square_L^+$$

By induction on the first premise, we have that:

$$(\Gamma, \Omega), x_2 :_r A \vdash B \Rightarrow^+ t \mid \Delta \qquad \text{(ih)}$$

from case 2 of the lemma. From which, we can construct the following instantiation of the $\square_L^+$ synthesis rule in the non-focusing calculus:

$$\frac{\begin{array}{c}(\Gamma, \Omega), x_2 :_r A \vdash B \Rightarrow^+ t \mid \Delta \\ \mathit{if}\ x_2 :_s A \in \Delta\ \mathit{then}\ s \sqsubseteq r\ \mathit{else}\ 0 \sqsubseteq r\end{array}}{\Gamma, (\Omega, x_1 : \square_r A) \vdash B \Rightarrow^+ \mathbf{let}\,[x_2] = x_1\,\mathbf{in}\,t \mid (\Delta \backslash x_2), x_1 : \square_r A}\ \square_{\mathrm{L}}^+$$

e) Case $\Uparrow_L^+$
In the case of the left asynchronous rule for transitioning an assumption from the focusing context Ω to the non-focusing context Γ, the synthesis rule has the form:

$$\frac{\Gamma, x : A; \Omega \Uparrow \vdash C \Rightarrow t \mid \Delta \qquad A \text{ not } \textsc{Left Async}}{\Gamma; \Omega, x : A \Uparrow \vdash C \Rightarrow t \mid \Delta}\ \Uparrow_L^+$$

By induction on the first premise, we have that:

$$\Gamma, x : A, \Omega \vdash C \Rightarrow^+ t \mid \Delta \qquad \text{(ih)}$$

from case 2 of the lemma.

3. Case 3. Right Sync:

   a) Case $\otimes_R^+$
   In the case of the right synchronous rule for pair introduction, the synthesis rule has the form:

$$\frac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t_1 \mid \Delta_1 \qquad \Gamma;\varnothing \vdash B \Downarrow \Rightarrow t_2 \mid \Delta_2}{\Gamma;\varnothing \vdash A \otimes B \Downarrow \Rightarrow (t_1, t_2) \mid \Delta_1 + \Delta_2} \otimes_R^+$$

   By induction on the premises, we have that:

$$\Gamma \vdash A \Rightarrow^+ t_1 \mid \Delta_1 \tag{ih1}$$

$$\Gamma \vdash B \Rightarrow^+ t_2 \mid \Delta_2 \tag{ih2}$$

   from case 3 of the lemma. From which, we can construct the following instantiation of the $\otimes_R^+$ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma \vdash A \Rightarrow^+ t_1 \mid \Delta_1 \qquad \Gamma \vdash B \Rightarrow^+ t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^+ (t_1, t_2) \mid \Delta_1 + \Delta_2} \otimes_{\mathrm{R}}^+$$

   b) Case $\oplus 1_R^+$ and $\oplus 2_R^+$
   In the case of the right synchronous rules for sum introduction, the synthesis rules have the form:

$$\frac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta}{\Gamma;\varnothing \vdash A \oplus B \Downarrow \Rightarrow \mathbf{inl}\, t \mid \Delta} \oplus 1_L^+$$

$$\frac{\Gamma;\varnothing \vdash B \Downarrow \Rightarrow t \mid \Delta}{\Gamma;\varnothing \vdash A \oplus B \Downarrow \Rightarrow \mathbf{inr}\, t \mid \Delta} \oplus 2_L^+$$

   By induction on the premises of the rules, we have that:

$$\Gamma \vdash A \Rightarrow^+ t \mid \Delta \tag{ih1}$$

$$\Gamma \vdash B \Rightarrow^+ t \mid \Delta \tag{ih2}$$

   from case 3 of the lemma. From which, we can construct the following instantiations of the $\oplus 1_R^+$ and $\oplus 2_R^+$ synthesis rules in the non-focusing calculus, respectively:

$$\frac{\Gamma \vdash A \Rightarrow^+ t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inl}\, t \mid \Delta} \oplus_{\mathrm{R}} 1^+$$

$$\frac{\Gamma \vdash B \Rightarrow^+ t \mid \Delta}{\Gamma \vdash A \oplus B \Rightarrow^+ \mathbf{inr}\, t \mid \Delta} \mathrm{R} \oplus_{\mathrm{R}} 2^+$$

c) Case Unit_R^+
In the case of the right synchronous rule for unit introduction, the synthesis rule has the form:

$$\frac{}{\Gamma;\varnothing \vdash \mathsf{Unit} \Rightarrow () \mid \varnothing}\ \mathsf{Unit}_R^+$$

From which, we can construct the following instantiation of the Unit_R^+ synthesis rule in the non-focusing calculus:

$$\frac{}{\Gamma \vdash \mathsf{Unit} \Rightarrow^+ () \mid \varnothing}\ \mathsf{Unit}_{\mathrm{R}}^+$$

d) Case $\Box_R^+$
In the case of the right synchronous rule for graded modality introduction, the synthesis rule has the form:

$$\frac{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta}{\Gamma;\varnothing \vdash \Box_r A \Downarrow \Rightarrow [t] \mid r \cdot \Delta}\ \Box_R^+$$

By induction on the premise, we have that:

$$\Gamma \vdash A \Rightarrow^+ t \mid \Delta \qquad \text{(ih)}$$

from case 1 of the lemma. From which, we can construct the following instantiation of the $\Box_R^+$ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma \vdash A \Rightarrow^+ t \mid \Delta}{\Gamma \vdash \Box_r A \Rightarrow^+ [t] \mid r \cdot \Delta}\ \Box_{\mathrm{R}}^+$$

e) Case $\Downarrow_R^+$
In the case of the right synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

$$\frac{\Gamma;\varnothing \vdash A \Uparrow \Rightarrow t \mid \Delta}{\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta}\ \Downarrow_R^+$$

By induction on the premise, we have that:

$$\Gamma \vdash A \Rightarrow^+ t \mid \Delta \qquad \text{(ih)}$$

from case 1 of the lemma.

4. Case 4. Left Sync

a) Case $\multimap_L^+$
In the case of the left synchronous rule for application, the synthesis rule has the form:

$$\frac{\Gamma; x_2 : B \Downarrow \vdash C \Rightarrow t_1 \mid \Delta_1, x_2 : B \qquad \Gamma; \emptyset \vdash A \Downarrow \Rightarrow t_2 \mid \Delta_2}{\Gamma; x_1 : A \multimap B \Downarrow \vdash C \Rightarrow [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B} \multimap_L^+$$

By induction on the first premise, we have that:

$$\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1 \mid \Delta_1, x_2 : B \tag{ih1}$$

from case 4 of the lemma. By induction on the second premise, we have that:

$$\Gamma \vdash A \Rightarrow^+ t_2 \mid \Delta_2 \tag{ih2}$$

from case 3 of the lemma. From which, we can construct the following instantiation of the $\multimap_L^+$synthesis rule in the non-focusing calculus:

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^+ t_1 \mid \Delta_1, x_2 : B \qquad \Gamma \vdash A \Rightarrow^+ t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^+ [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B} \multimap_L^+$$

b) Case DER^+
In the case of the left asynchronous rule for dereliction, the synthesis rule has the form:

$$\frac{\Gamma; x :_s A, y : A \Downarrow \vdash B \Rightarrow t \mid \Delta, y : A}{\Gamma; x :_s A \Downarrow \vdash B \Rightarrow [x/y]t \mid \Delta + x :_1 A} \text{DER}^+$$

By induction on the premise, we have that:

$$\Gamma, x :_s A, y : A \vdash B \Rightarrow^+ t \mid \Delta, y : A \tag{ih}$$

from case 4 of the lemma. From which, we can construct the following instantiation of the DER^+ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma, x :_s A, y : A \vdash B \Rightarrow^+ t \mid \Delta, y : A}{\Gamma, x :_s A \vdash B \Rightarrow^+ [x/y]t \mid \Delta + x :_1 A} \text{DER}^+$$

c) Case LINVAR^+
In the case of the left synchronous rule for linear variable synthesis, the synthesis rule has the form:

$$\frac{}{\Gamma; x : A \vdash A \Rightarrow x \mid x : A} \text{LINVAR}^+$$

From which, we can construct the following instantiation of the LINVAR^+ in the non-focusing calculus:

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow^+ x \mid x : A} \text{LINVAR}^+$$

d) Case $\textsc{GrVar}^+$
In the case of the left synchronous rule for graded variable synthesis, the synthesis rule has the form:

$$\frac{}{\Gamma; x :_r A \vdash A \Rightarrow x \mid x :_1 A}\ \textsc{GrVar}^+$$

From which, we can construct the following instantiation of the $\textsc{GrVar}^+$ synthesis rule in the non-focusing calculus:

$$\frac{}{\Gamma, x :_r A \vdash A \Rightarrow^+ x \mid\ x :_1 A}\ \textsc{GrVar}^+$$

e) Case $\Downarrow_L^+$
In the case of the left synchronous rule for transitioning back to an asynchronous judgement, the synthesis rule has the form:

$$\frac{\Gamma; x : A \Uparrow \vdash C \Rightarrow t \mid \Delta \qquad A \text{ not atomic and not } \textsc{Left Sync}}{\Gamma; x : A \Downarrow \vdash C \Rightarrow t \mid \Delta}\ \Downarrow_L^+$$

By induction on the premise, we have that:

$$\Gamma, x : A \vdash C \Rightarrow^+ t \mid\ \Delta \tag{ih}$$

from case 2 of the lemma.

5. Case 5. Focus Right: focus_R^+
In the case of the focusing rule for transitioning from a left asynchronous judgement to a right synchronous judgement, the synthesis rule has the form:

$$\frac{\Gamma; \varnothing \vdash C \Downarrow \Rightarrow t \mid \Delta \qquad C \text{ not atomic}}{\Gamma; \varnothing \Uparrow \vdash C \Rightarrow t \mid \Delta}\ \textsc{focus}_R^+$$

By induction on the first premise, we have that:

$$\Gamma \vdash C \Rightarrow^+ t \mid\ \Delta \tag{ih}$$

from case 2 of the lemma.

6. Case 6. Focus Left: focus_L^+
In the case of the focusing rule for transitioning from a left asynchronous judgement to a left synchronous judgement, the synthesis rule has the form:

$$\frac{\Gamma; x : A \Downarrow \vdash C \Rightarrow t \mid \Delta}{\Gamma, x : A; \varnothing \Uparrow \vdash C \Rightarrow t \mid \Delta}\ \textsc{focus}_L^+$$

By induction on the first premise, we have that:

$$\Gamma, x : A \vdash C \Rightarrow^+ t \mid\ \Delta \tag{ih}$$

from case 2 of the lemma.

□

B.1.6 *Soundness of Focusing for the Additive Pruning Linear Graded Synthesis Calculus*

Lemma 3.5.3 (Soundness of focusing for additive pruning synthesis)**.** For all contexts Γ, Ω and types A, B then:

1. RIGHT ASYNC:	$\Gamma;\Omega \vdash A \Uparrow \Rightarrow^{\pm} t \mid \Delta$	$\iff$	$\Gamma,\Omega \vdash A \Rightarrow^{\pm} t \mid \Delta$
2. LEFT ASYNC:	$\Gamma;\Omega \Uparrow \vdash A \Rightarrow^{\pm} t \mid \Delta$	$\iff$	$\Gamma,\Omega \vdash B \Rightarrow^{\pm} t \mid \Delta$
3. RIGHT SYNC:	$\Gamma;\emptyset \vdash A \Downarrow \Rightarrow^{\pm} t \mid \Delta$	$\iff$	$\Gamma \vdash A \Rightarrow^{\pm} t \mid \Delta$
4. LEFT SYNC:	$\Gamma;x : A \Downarrow \vdash B \Rightarrow^{\pm} t \mid \Delta$	$\iff$	$\Gamma,x : A \vdash B \Rightarrow^{\pm} t \mid \Delta$
5. FOCUS RIGHT:	$\Gamma;\emptyset \Uparrow \vdash A \Rightarrow^{\pm} t \mid \Delta$	$\iff$	$\Gamma \vdash B \Rightarrow^{\pm} t \mid \Delta$
6. FOCUS LEFT:	$\Gamma,x : A;\emptyset \Uparrow \vdash B \Uparrow \Rightarrow^{\pm} t \mid \Delta$	$\iff$	$\Gamma,x : A \mid B \Rightarrow^{\pm} t \mid \Delta$

i.e. t has type A under context Δ, which contains assumptions with grades reflecting their use in t.

Proof. 1. Case: 1. Right Async: The proofs for right asynchronous rules are equivalent to those of lemma (3.5.2)

2. Case 2. Left Async: The proofs for left asynchronous rules are equivalent to those of lemma (3.5.2)

3. Case 3. Right Sync: The proofs for right synchronous rules are equivalent to those of lemma (3.5.2), except for the case of the $\otimes_R^{\pm}$rule:

   a) Case $\otimes_R^{\pm}$
   In the case of the right synchronous rule for pair introduction, the synthesis rule has the form:

$$\frac{\Gamma;\emptyset \vdash A \Downarrow \Rightarrow^{\pm} t_1 \mid \Delta_1 \qquad \Gamma - \Delta_1 \vdash B \Downarrow \Rightarrow^{\pm} t_2 \mid \Delta_2}{\Gamma;\emptyset \vdash A \otimes B \Downarrow \Rightarrow^{\pm} (t_1, t_2) \mid \Delta_1 + \Delta_2} \otimes_R^{\pm}$$

   By induction on the premises, we have that:

$$\Gamma \vdash A \Rightarrow^{+} t_1 \mid \Delta_1 \tag{ih1}$$

$$\Gamma - \Delta_1 \vdash B \Rightarrow^{+} t_2 \mid \Delta_2 \tag{ih2}$$

   from case 3 of the lemma. From which, we can construct the following instantiation of the $\otimes_R^{\pm}$ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma \vdash A \Rightarrow^{+} t_1 \mid \Delta_1 \qquad \Gamma - \Delta_1 \vdash B \Rightarrow^{+} t_2 \mid \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow^{+} (t_1, t_2) \mid \Delta_1 + \Delta_2} \otimes_{\mathrm{R}}^{\pm}$$

4. Case 4. Left Sync: The proofs for left synchronous rules are equivalent to those of lemma (3.5.2), except for the case of the $\multimap^{\pm}_{L}$ rule:

   a) Case $\multimap^{\pm}_{L}$
   In the case of the left synchronous rule for application, the synthesis rule has the form:

$$\frac{\Gamma; x_2 : B \Downarrow \vdash C \Rightarrow^{\pm} t_1 \mid \Delta_1, x_2 : B \qquad \Gamma - \Delta_1; \emptyset \vdash A \Downarrow \Rightarrow^{\pm} t_2 \mid \Delta_2}{\Gamma; x_1 : A \multimap B \Downarrow \vdash C \Rightarrow^{\pm} [(x_1\ t_2)/x_2] \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap^{\pm}_{L}$$

   By induction on the first premise, we have that:

$$\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1 \mid \Delta_1, x_2 : B \tag{ih1}$$

   from case 4 of the lemma. By induction on the second premise, we have that:

$$\Gamma \vdash A \Rightarrow^{+} t_2 \mid \Delta_2 \tag{ih2}$$

   from case 3 of the lemma. From which, we can construct the following instantiation of the $\multimap^{\pm}_{L}$ synthesis rule in the non-focusing calculus:

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow^{+} t_1 \mid \Delta_1, x_2 : B \qquad \Gamma - \Delta_1 \vdash A \Rightarrow^{+} t_2 \mid \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow^{+} [(x_1\ t_2)/x_2]t_1 \mid (\Delta_1 + \Delta_2), x_1 : A \multimap B}\ \multimap^{\pm}_{L}$$

5. Case 5. Right Focus: focus^{+}_{R} - The proof for right focusing rule is equivalent to that of lemma (3.5.2)

6. Case 6. Left Focus: focus^{+}_{L} - The proof for left focusing rule is equivalent to that of lemma (3.5.2)

□

B.2 PROOFS FOR THE DERIVING MECHANISM

This section contains the proofs relating to Chapter 5. We begin by presenting a typed equational theory for use in the proofs that *push* and *pull* are the inverse of each other (Section B.2.1). The type soundness proofs of *push* and *pull* then follow in Section B.2.3, followed by the aforementioned inverse proofs in Section B.2.4.

B.2.1 *Typed Equational Theory*

Figure B.1 defines an equational theory for the linear base Granule typing calculus used in Chapter 5. The equational theory is typed, and we provide the typed forms of the rules also. For those rules which are type restricted, we include the full typed-equality judgment here. The type-ability of these equations relies on previous work on the Granule language which proves that pattern matching and substitution are well typed Orchard et al. [2019].

$$
\begin{array}{ll}
\beta: & (\lambda x.t_2)\, t_1 \;\equiv\; t_2[t_1/x] \\
\eta: & \lambda x.t\, x \;\equiv\; t \quad (x\#t) \\
\beta\textsc{Let}: & \textbf{letrec}\ x \;=\; t_1\ \textbf{in}\ t_2 \equiv\; t_2[\textbf{letrec}\ x \;=\; t_1\ \textbf{in}\ t_1/x] \\
\textsc{DistLet}: & f\,(\textbf{letrec}\ x \;=\; t_1\ \textbf{in}\ t_2) \equiv\; \textbf{letrec}\ x \;=\; t_1\ \textbf{in}\ (f\, t_2) \\
\beta\textsc{Case}: & \textbf{case}\ t\ \textbf{of}\ \overline{p_i \to t_i} \;\equiv\; (t \rhd p_j)t_j \quad (\mathsf{minimal}(j)) \\
\eta\textsc{Case}: & \textbf{case}\ t_1\ \textbf{of}\ \overline{p_i \to [p_i/z]t_2} \;\equiv\; [t_1/z]t_2 \\
\textsc{AssocCase}: & \textbf{case}\ (\textbf{case}\ t\ \textbf{of}\ \overline{p_i \to t_i})\ \textbf{of}\ \overline{p'_i \to t'_i} \\
& \qquad \equiv\; \textbf{case}\ t\ \textbf{of}\ \overline{p_i \to (\textbf{case}\ t_i\ \textbf{of}\ \overline{p'_i \to t'_i})} \\
\textsc{DistCase}: & f\,(\textbf{case}\ t\ \textbf{of}\ \overline{p_i \to t_i}) \equiv\; \textbf{case}\ t\ \textbf{of}\ \overline{p_i \to (f\, t_i)} \\
\textsc{Assoc}\Box: & \textbf{case}\ [\textbf{case}\ t\ \textbf{of}\ \overline{p_i \to t_i}]\ \textbf{of}\ \overline{[p'_i] \to t'_i} \\
& \qquad \equiv\; \textbf{case}\ [t]\ \textbf{of}\ \overline{[p_i] \to \textbf{case}\ [t_i]\ \textbf{of}\ \overline{[p'_i] \to t'_i}} \quad (\mathsf{lin}(p_i))
\end{array}
$$

Figure B.1: Equational theory for linear base Granule

The β and η rules follow the standard rules from the λ-calculus, where # is a *freshness* predicate, denoting that variable x does not appear inside term t.

For recursive **letrec** bindings, the β_{letrec} rule substitutes any occurrence of the bound variable x in t_2 with **letrec** $x \;=\; t_1$ **in** t_1, ensuring that recursive uses of x inside t_1 can be substituted with t_1 through subsequent β_{letrec} reduction. The LetRecDistrib rule allows distributivity of functions over **letrec** expressions, stating that if a function f can be applied to the entire **letrec** expression, then this is equivalent to applying f to just the body term t_2.

Term elimination is via **case**, requiring rules for both β- and η-equality on case expressions, as well as rules for associativity and distributivity. In β_{case}, a term t is matched against a pattern p_j in the context of the term t_j through the use of the partial function $(t \;\rhd\; p_j)t_j = t'$ which may substitute terms bound in p_j into t_j to

yield t' if the match is successful. This partial function is defined inductively:

$$\frac{}{(t \rhd \_)t' = t'} \rhd_{-} \qquad \frac{}{(t \rhd x)t' = [t/x]t'} \rhd_{var}$$

$$\frac{(t \rhd p)t' = t''}{([t] \rhd [p])t' = t''} \rhd_{\Box} \qquad \frac{(t_i \rhd p_i)t'_i = t'_{i+1}}{(C\ t_1 ... t_n \rhd C\ p_1 \ldots p_n)t'_1 = t'_{n+1}} \rhd_{C}$$

As a predicate to the β_{case} rule, we require that j be minimal, i.e. the first pattern p_j in $p_1 ... p_n$ for which $(t \rhd p_j)t_j = t'$ is defined. Rule η_{case} states that if all branches of the case expression share a common term t_2 which differs between branches only in the occurrences of terms that match the pattern used, then we can substitute t_1 for the pattern inside t_2.

Associativity of case expressions is provided by the CaseAssoc rule. This rule allows us to restructure nested case expressions such that the output terms t_i of the inner case may be matched against the patterns of the outer case, to achieve the same resulting output terms t'_i. The [CaseAssoc] rule provides a graded alternative to the CaseAssoc rule, where the nested case expression is graded, provided that the patterns p'_i of the outer case expression are also graded. Notably, this rule only holds when the patterns of the inner case expression are linear (i.e., variable or constant) so that there are no nested box patterns, represented via the $\mathsf{lin}(p_i)$ predicate. As with **letrec**, distributivity of functions over a case expression is given by CaseDistrib.

Lastly generalisation of an arbitrary boxed pattern to a variable is permitted through the CaseGen rule. Here, a boxed pattern $[p_i]$ and the output term of the case may be converted to a variable if the output term is equivalent to the pattern inside the box. The term t being matched against must therefore have a grade approximatable by 1, as witnessed by the predicate $1 \sqsubseteq r$ in the typing derivation.

In AssocCase the predicate $\mathsf{lin}(p)$ classifies those patterns which are *linear*, which are those which are variables or constructor patterns only.

B.2.1.1 *Derived Rules*

Proposition B.2.1 ('Case push' property)**.**

$$\frac{\Gamma \vdash t : A \qquad r \vdash p_i : A \rhd \Delta_i \qquad \Gamma', \Delta_i \vdash t_i : B}{s \cdot r \cdot \Gamma + s \cdot \Gamma' \vdash [\textbf{\textit{case}}\ t\ \textbf{\textit{of}}\ \overline{[p_i] \to t_i}] \equiv \textbf{\textit{case}}\ [t]\ \textbf{\textit{of}}\ \overline{[p_i] \to [t_i]} : \Box_s B}\ \textsc{Push}$$

Proof. Applying β_{case} and congruence over promotion, to the left-hand side of the case push equation yields:

$$[\textbf{case}\ t\ \textbf{of}\ [p_i] \to t_i] = [(t \rhd p_j)t_j]$$

$$\frac{\Gamma_1, x : A \vdash t_2 : B \qquad \Gamma_2 \vdash t_1 : A}{\Gamma_1 + \Gamma_2 \vdash (\lambda x.t_2)\, t_1 \equiv [t_1/x]t_2 : B} \quad \beta$$

$$\frac{\Gamma \vdash t : A \multimap B \qquad [x \# t]}{\Gamma \vdash \lambda x.t\, x \equiv t : A \multimap B} \quad \eta$$

$$\frac{\Gamma_1, x : A \vdash t_1 : A \qquad \Gamma_2, x : A \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{letrec}\ x\ =\ t_1\ \mathbf{in}\ t_2 \equiv [\mathbf{letrec}\ x\ =\ t_1\ \mathbf{in}\ t_1/x]t_2 : B} \quad \beta Let$$

$$\frac{\Gamma_1, x : A \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B \qquad \Gamma_3 \vdash f : B \multimap W}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash f\,(\mathbf{letrec}\ x\ =\ t_1\ \mathbf{in}\ t_2) \equiv \mathbf{letrec}\ x\ =\ t_1\ \mathbf{in}\ (f\, t_2) : W} \quad \textsc{DistLet}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : A \qquad - \vdash p_i : A \rhd \Delta_i \\ \Gamma_2, \Delta_i \vdash t_i : B\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}\ t\ \mathbf{of}\ \overline{p_i \to t_i} \equiv (t \rhd p_j)t_j : B} \quad \beta_{\textsc{Case}}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t_1 : A \qquad - \vdash p_i : A \rhd \Delta_i \\ \Gamma_2, z : A \vdash t_2 : B\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}\ t_1\ \mathbf{of}\ \overline{p_i \to [p_i/z]t_2} \equiv [t_1/z]t_2 : B} \quad \eta_{\textsc{Case}}$$

$$\frac{\Gamma \vdash t : \Box_r A \qquad r \vdash p_i : A \rhd \Delta_i \qquad \Delta_i \vdash p_i : A \qquad 1 \sqsubseteq r}{\Gamma \vdash \mathbf{case}\ t\ \mathbf{of}\ \overline{[p_i] \to p_i} \equiv \mathbf{case}\ t\ \mathbf{of}\ [x] \to x : A} \quad \textsc{GenCase}$$

$$\frac{\Gamma \vdash t : A \qquad - \vdash p_i : A \rhd \Delta_i \qquad \Gamma', \Delta_i' \vdash t_i : B \qquad - \vdash p_i' : B \rhd \Delta_i' \qquad \Gamma'', \Delta_i' \vdash t_i' : W}{\Gamma + \Gamma' + \Gamma'' \vdash \mathbf{case}\ (\mathbf{case}\ t\ \mathbf{of}\ \overline{p_i \to t_i})\ \mathbf{of}\ \overline{p_i' \to t_i'} \equiv \mathbf{case}\ t\ \mathbf{of}\ \overline{p_i \to (\mathbf{case}\ t_i\ \mathbf{of}\ \overline{p_i' \to t_i'})} : W} \quad \textsc{AssocCase}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : A \qquad - \vdash p_i : A \rhd \Delta_i \qquad \Gamma', \Delta_i' \vdash t_i : B \\ r \vdash p_i' : B \rhd \Delta_i' \qquad \Gamma'', \Delta_i' \vdash t_i' : W \qquad \mathsf{lin}(p)\end{array}}{r \cdot (\Gamma + \Gamma') + \Gamma'' \vdash \mathbf{case}\ [\mathbf{case}\ t\ \mathbf{of}\ \overline{p_i \to t_i}]\ \mathbf{of}\ \overline{[p_i'] \to t_i'} \equiv \mathbf{case}\ [t]\ \mathbf{of}\ \overline{[p_i] \to \mathbf{case}\ [t_i]\ \mathbf{of}\ \overline{[p_i'] \to t_i'}} : W} \quad \textsc{Assoc}\Box$$

$$\frac{\Gamma_1 \vdash t : A \qquad - \vdash p_i : A \rhd \Delta_i \qquad \Gamma_2, \Delta_i \vdash t_i : B \qquad \Gamma_3 \vdash f : B \multimap W}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash f\,(\mathbf{case}\ t\ \mathbf{of}\ \overline{p_i \to t_i}) \equiv \mathbf{case}\ t\ \mathbf{of}\ \overline{p_i \to (f\, t_i)} : W} \quad \textsc{DistCase}$$

Figure B.2: Typed equational theory for linear base Granule

for the smallest j. Applying β_{case} to the right-hand side of the case push equation yields:

$$\textbf{case}\ [t]\ \textbf{of}\ [p_i] \to [t_i] = ([t] \rhd [p_i])[t_j]$$

for the same smallest j (since the patterns pi are the same).

By PATSEMUNBOX, then we have the derivation of pattern matching:

$$\frac{(t \rhd p_i)[t_j] = t''}{([t] \rhd [p_i])[t_j] = t''}\ \textsc{patSemunbox}$$

therefore $\textbf{case}\ [t]\ \textbf{of}\ [p_i] \to [t_i] = ([t] \rhd [p_i])[t_j] = (t \rhd p_i)[t_j]$.

Then by Proposition B.2.2 (below), $(t \rhd p_i)[t_j] = [(t \rhd p_j)t_j]$, yielding case push. □

Proposition B.2.2 (Pattern matching distributes with promotion)**.** *For all t, p, t' then:*

$$(t \rhd p)[t'] = [(t \rhd p)t']$$

Proof. By induction on syntactic pattern matching:

- (wild) $(t \rhd \_)[t'] = [t']$ and $[(t \rhd \_)t'] = [t']$.
- (var) $(t \rhd x)[t'] = [t/x][t'] = [[t/x]t']$ and $[(t \rhd x)t'] = [[t/x]t']$
- (unbox)

  $$\frac{(t \rhd p)t' = t''}{([t] \rhd [p])t' = t''}\ \textsc{patSemunbox}$$

  By induction then $(t \rhd p)[t'] = [(t \rhd p)t']$ therefore $([t] \rhd [p])[t'] = [([t] \rhd [p])t']$ since this rule preserves its result in the conclusion.
- (constr)

  $$\frac{(t_i \rhd p_i)t_i = t_{i+1}}{(C\ t_0\ ..\ t_n \rhd C\ p_0\ ..\ p_n)t_0 = t_{n+1}}\ \textsc{patSemconstr}$$

  By induction, similarly to the above case, but across multiple terms.

□

B.2.2 *Functor Derivation*

Definition B.2.1 (Deriving functor)**.** Given a function $f : \alpha \multimap \beta$ then there is a function $[\![\mathsf{F}\overline{\alpha}]\!]_{\mathsf{fmap}}(f) : \mathsf{F}\ \alpha \multimap \mathsf{F}\ \beta$ derived from the type $\mathsf{F}\overline{\alpha}$ as follows:

$$
\begin{array}{lcl}
[\![\mathsf{Unit}]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ z & = & \mathbf{case}\ z\ \mathbf{of}\ () \to () \\
[\![\alpha]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ z & = & f\ z \\
[\![X]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ z & = & (\Sigma(X)\ f)\ z \\
[\![\Box_r A]\!]_{\mathsf{fmap}}(f)\ z & = & \mathbf{case}\ z\ \mathbf{of}\ [y] \to [[\![A]\!]_{\mathsf{fmap}}(f)\ y] \\
[\![A \oplus B]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ z & = & \mathbf{case}\ z\ \mathbf{of}\ \ \mathsf{inl}\ x \to \mathsf{inl}\ [\![A]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ x; \\
 & & \qquad\qquad\qquad \mathsf{inr}\ y \to \mathsf{inr}\ [\![B]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ y \\
[\![A \otimes B]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ z & = & \mathbf{case}\ z\ \mathbf{of}\ (x, y) \to ([\![A]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ x, [\![B]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ y) \\
[\![A \multimap B]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ z & = & \lambda x.[\![B]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ (z\ x) \\
[\![\mu X.A]\!]^{\Sigma}_{\mathsf{fmap}}(f)\ z & = & \mathbf{letrec}\ g = [\![A]\!]^{\Sigma, X \mapsto g : (\alpha \multimap \beta) \multimap \mu X.A \multimap (\mu X.A)\overrightarrow{[\alpha/\beta]}}_{\mathsf{fmap}}(f)\ \mathbf{in}\ g\ z
\end{array}
$$

B.2.3 *Type Soundness of* push *and* pull

The following shows that the calculation of *push* and *pull* distributive laws is well-typed.

Proposition 1 (Type soundness of $[\![\mathsf{F}\ \overline{\alpha_i}]\!]^{\Sigma}_{\mathsf{push}}$)**.** $[\![\mathsf{F}\overline{\alpha_i}]\!]^{\Sigma}_{\mathsf{push}} : \Box_r \mathsf{F}\overline{\alpha_i} \to \mathsf{F}(\overline{\Box_r \alpha_i})$

Proof.

- $[\![\mathsf{Unit}]\!]^{\Sigma}_{\mathsf{push}} : \Box_r \mathsf{Unit} \to \mathsf{Unit}$ (i.e. $\mathsf{F}\ \overline{\alpha_i} = \mathsf{Unit}$).

$$
\dfrac{\dfrac{}{\varnothing \vdash () : \mathsf{Unit}}\ \textsc{con} \qquad \dfrac{\dfrac{|\mathsf{Unit}| = 1}{r \vdash\ () : \mathsf{Unit} \rhd \varnothing}\ [\textsc{pcon}]}{- \vdash\ [()] : \mathsf{Unit} \rhd \varnothing}\ [\textsc{pbox}]}{z : \Box_r \mathsf{Unit} \vdash \mathbf{case}\ z\ \mathbf{of}\ [()] \to () : \mathsf{Unit}}\ \textsc{case}
$$

- $[\![X]\!]^{\Sigma}_{\mathsf{push}} : \Box_r X \to X\overrightarrow{[\Box_r \alpha_i / \alpha_i]}$ (i.e. $\mathsf{F}\ \overline{\alpha_i} = X$).

$$
\dfrac{\dfrac{X : \Box_r(\mu X.A) \multimap (\mu X.A)\overrightarrow{[\Box_r \alpha_i/\alpha_i]} \in \Sigma}{\Sigma \vdash \Sigma(X)\ : \Box_r(\mu X.A) \multimap (\mu X.A)\overrightarrow{[\Box_r \alpha_i/\alpha_i]}}\ \textsc{lookup} \qquad \dfrac{}{z : \Box_r(\mu X.A) \vdash z : \Box_r(\mu X.A)}\ \textsc{var}}{\Sigma, z : \Box_r(\mu X.A) \vdash \Sigma(X)\ z : (\mu X.A)\overrightarrow{[\Box_r \alpha_i/\alpha_i]}}\ \textsc{app}
$$

- $[\![\alpha_j]\!]^{\Sigma}_{\mathsf{push}} : \Box_{r_j} \alpha_j \to \Box_{r_j} \alpha_j$ (i.e. $\mathsf{F}\ \overline{\alpha_i} = \alpha$).

$$
\dfrac{}{z : \Box_{r_j} \alpha_j \vdash z : \Box_{r_j} \alpha_j}\ \textsc{var}
$$

- $[\![A \oplus B]\!]^{\Sigma}_{\mathsf{push}} : \Box_r(A \oplus B) \to (A\overrightarrow{[\Box_r\alpha_i/\alpha_i]} \oplus B\overrightarrow{[\Box_r\alpha_i/\alpha_i]})$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}\,\textsc{var}}{x :_1 A \vdash x : A}\,\textsc{der}}{x :_r A \vdash [x] : \Box_r A}\,\textsc{Pr}}{x :_r A \vdash [\![A]\!]^{\Sigma}_{\mathsf{push}}([x]) : A\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}\,\textsc{app}}{x :_r A \vdash \mathbf{inr}\,[\![A]\!]^{\Sigma}_{\mathsf{push}}([x]) : A\overrightarrow{[\Box_r\alpha_i/\alpha_i]} \oplus B\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}\,\textsc{Con} \qquad \text{(B.1)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : B \vdash y : B}\,\textsc{var}}{y :_1 B \vdash y : B}\,\textsc{der}}{y :_r B \vdash [y] : \Box_r B}\,\textsc{Pr}}{y :_r B \vdash [\![B]\!]^{\Sigma}_{\mathsf{push}}([y]) : B\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}\,\textsc{app}}{y :_r B \vdash \mathbf{inr}\,[\![B]\!]^{\Sigma}_{\mathsf{push}}([y]) : A\overrightarrow{[\Box_r\alpha_i/\alpha_i]} \oplus B\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}\,\textsc{Con} \qquad \text{(B.2)}$$

$$\dfrac{\dfrac{\dfrac{}{r \vdash x : A \rhd x :_r A}\,[\textsc{Pvar}] \qquad |A \oplus B| > 1 \Rightarrow 1 \sqsubseteq r}{r \vdash \mathbf{inl}\,(x) : A \oplus B \rhd x :_r A}\,[\textsc{Pcon}]}{- \vdash [\mathbf{inl}\,(x)] : \Box_r A \oplus B \rhd x :_r A}\,[\textsc{Pbox}] \qquad \text{(B.3)}$$

$$\dfrac{\dfrac{\dfrac{}{r \vdash y : B \rhd y :_r B}\,[\textsc{Pvar}] \qquad |A \oplus B| > 1 \Rightarrow 1 \sqsubseteq r}{r \vdash \mathbf{inr}\,(y) : A \oplus B \rhd y :_r B}\,[\textsc{Pcon}]}{- \vdash [\mathbf{inr}\,(y)] : \Box_r A \oplus B \rhd y :_r B}\,[\textsc{Pbox}] \qquad \text{(B.4)}$$

$$\mathbf{case}\ z\ \mathbf{of}\ [\mathbf{inl}\,(x)] \to \mathbf{inl}\,[\![A]\!]^{\Sigma}_{\mathsf{push}}([x]); [\mathbf{inr}\,(y)] \to \mathbf{inr}\,[\![B]\!]^{\Sigma}_{\mathsf{push}}([y]) \qquad \text{(B.5)}$$

$$\dfrac{z : \Box_r(A \oplus B) \qquad \text{(B.1)} \quad \text{(B.2)} \quad \text{(B.3)} \quad \text{(B.4)}}{\vdash \text{(B.5)} : A\overrightarrow{[\Box_r\alpha_i/\alpha_i]} \oplus B\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}\,\textsc{Case}$$

- $[\![A \otimes B]\!]^{\Sigma}_{\mathsf{push}} : \Box_r(A \otimes B) \to (A\overrightarrow{[\Box_r\alpha_i/\alpha_i]} \otimes B\overrightarrow{[\Box_r\alpha_i/\alpha_i]})$

$$\dfrac{\dfrac{}{\varnothing \vdash [\![A]\!]^{\Sigma}_{\mathsf{push}} : \Box_r A \multimap A\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}\,\textsc{push} \qquad \dfrac{\dfrac{\dfrac{}{x : A \vdash x : A}\,\textsc{var}}{x :_1 A \vdash x : A}\,\textsc{der}}{x :_r A \vdash [x] : \Box_r A}\,\textsc{pr}}{x :_r A \vdash [\![A]\!]^{\Sigma}_{\mathsf{push}}([x]) : A\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}\,\textsc{app}$$

(B.6)

$$\dfrac{\dfrac{}{\emptyset \vdash [\![B]\!]^{\Sigma}_{\mathsf{push}} : \Box_r B \multimap B[\overrightarrow{\Box_r \alpha_i / \alpha_i}]}\ \textsc{push} \qquad \dfrac{\dfrac{\dfrac{}{y : B \vdash y : B}\ \textsc{var}}{y :_1 B \vdash y : B}\ \textsc{der}}{y :_r B \vdash [y] : \Box_r B}\ \textsc{pr}}{y :_r B \vdash [\![B]\!]^{\Sigma}_{\mathsf{push}}([y]) : B[\overrightarrow{\Box_r \alpha_i / \alpha_i}]}\ \textsc{app}$$

(B.7)

$$\dfrac{\text{(B.6)} \qquad \text{(B.7)}}{x :_r A, y :_r B \vdash ([\![A]\!]^{\Sigma}_{\mathsf{push}}([x]), [\![B]\!]^{\Sigma}_{\mathsf{push}}([y])) : A[\overrightarrow{\Box_r \alpha_i / \alpha_i}] \otimes B[\overrightarrow{\Box_r \alpha_i / \alpha_i}]}\ \textsc{Con}$$

(B.8)

$$\dfrac{\dfrac{\dfrac{}{r \vdash x : A \rhd x :_r A}\ [\textsc{Pvar}]}{r \vdash y : B \rhd y :_r B}\ [\textsc{Pvar}] \quad |A \otimes B| = 1}{r \vdash (x,y) : A \otimes B \rhd x :_r A, y :_r B}\ [\textsc{PCon}]}{- \vdash [(x,y)] : \Box_r A \otimes B \rhd x :_r A, y :_r B}\ [\textsc{Pbox}]$$

(B.9)

$$\textbf{case}\ z\ \textbf{of}\ [(x,y)] \to ([\![A]\!]^{\Sigma}_{\mathsf{push}}([x]), [\![B]\!]^{\Sigma}_{\mathsf{push}}([y])) \tag{B.10}$$

$$\dfrac{\text{(B.8)} \qquad \text{(B.9)}}{z : \Box_r(A \otimes B) \vdash \text{(B.10)} : A[\overrightarrow{\Box_{r_i} \alpha_i / \alpha_i}] \otimes B[\overrightarrow{\Box_{r_i} \alpha_i / \alpha_i}]}\ \textsc{Case}$$

- $[\![A \multimap B]\!]^{\Sigma}_{\mathsf{push}} : \Box_r(A \multimap B) \to (A[\overrightarrow{\Box_r \alpha_i / \alpha_i}] \multimap B[\overrightarrow{\Box_{r_i} \alpha_i / \alpha_i}])$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : A \multimap B \vdash f : A \multimap B}\ \textsc{var}}{f :_1 A \multimap B \vdash f : A \multimap B}\ \textsc{der} \qquad \dfrac{\dfrac{}{x : A \vdash x : A}\ \textsc{var}}{x :_1 A \vdash x : A}\ \textsc{der}}{f :_1 A \multimap B, x :_1 A \vdash f\, x : B}\ \textsc{app}}{f :_r A \multimap B, x :_{\prod_1^n r_i} A \vdash [f\, x] : \Box_r B}\ \textsc{Pr} \tag{B.11}$$

$$\dfrac{\dfrac{}{\emptyset \vdash [\![B]\!]^{\Sigma}_{\mathsf{push}} : \Box_r B \multimap B[\overrightarrow{\Box_r \alpha_i / \alpha_i}]}\ \textsc{push} \qquad \text{(B.11)}}{f :_r A \multimap B, x :_{\prod_1^n r_i} A \vdash [\![B]\!]^{\Sigma}_{\mathsf{push}}[f\, x] : B[\overrightarrow{\Box_r \alpha_i / \alpha_i}]}\ \textsc{app} \tag{B.12}$$

$$\dfrac{\dfrac{\dfrac{}{\prod^n r_i \vdash x : A \rhd x :_{\prod_1^n r_i} A}\ [\textsc{Pvar}]}{1 \vdash [x] : \Box_r A \rhd x :_{\prod_1^n r_i} A}\ [\textsc{Pbox}]}{- \vdash [x] : \Box_r A \rhd x :_{\prod_1^n r_i} A}$$

(B.13)

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash [\![A]\!]^{\Sigma}_{\mathsf{pull}} : A \multimap \Box_{\prod_1^n r_i} A}\,\textsc{pull} \qquad \dfrac{}{y : A \vdash y : A}\,\textsc{var}}{y : A \vdash [\![A]\!]^{\Sigma}_{\mathsf{pull}}(y) : \Box_{\prod_1^n r_i} A}\,\textsc{app}}{y : A, f :_r A \multimap B \vdash \mathbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{pull}}(y)\ \mathbf{of}\ [x] \to [\![B]\!]^{\Sigma}_{\mathsf{push}}[f\,x] : B[\overrightarrow{\Box_r \alpha_i / \alpha_i}]}\,\textsc{Case}$$

(B.12) (B.13)

(B.14)

$$\dfrac{\dfrac{}{r \vdash f : (A \multimap B) \rhd f :_r A \multimap B}\,[\textsc{Pvar}]}{- \vdash [f] : \Box_r (A \multimap B) \rhd f :_r A \multimap B}\,[\textsc{Pbox}] \tag{B.15}$$

$$\mathbf{case}\ z\ \mathbf{of}\ [f] \to \mathbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{pull}}(y)\ \mathbf{of}\ [x] \to [\![B]\!]^{\Sigma}_{\mathsf{push}}[f\,x] \tag{B.16}$$

$$\dfrac{\dfrac{\text{(B.14)} \qquad \text{(B.15)}}{z : \Box_r (A \multimap B), y : A \vdash \text{(B.16)} : B[\overrightarrow{\Box_r \alpha_i / \alpha_i}]}\,\textsc{Case}}{z : \Box_r (A \multimap B) \vdash \lambda y.\text{(B.16)} : A[\overrightarrow{\Box_r \alpha_i / \alpha_i}] \multimap B[\overrightarrow{\Box_r \alpha_i / \alpha_i}]}\,\textsc{abs}$$

- $[\![\mu X.A]\!]^{\Sigma}_{\mathsf{push}} : (\mu X.\Box_r A) \to (\mu X.A[\overrightarrow{\Box_{r_i} \alpha_i / \alpha_i}])$ (i.e. $\mathsf{F}\ \overline{\alpha_i} = \mu X.A$).

$$\dfrac{}{\Sigma, f : \mu X.\Box_r A \multimap (\mu X.A[\overrightarrow{\Box_r \alpha_i / \alpha_i}]) \vdash f : \mu X.\Box_r A \multimap (\mu X.A[\overrightarrow{\Box_r \alpha_i / \alpha_i}])}\,\textsc{var} \tag{B.17}$$

$$\dfrac{\text{(B.17)} \qquad \dfrac{}{\Sigma, z : \mu X.\Box_r A \vdash z : \mu X.\Box_r A}\,\textsc{var}}{\Sigma, f : \mu X.\Box_r A \multimap (\mu X.A[\overrightarrow{\Box_r \alpha_i / \alpha_i}]), z : \mu X.\Box_r A \vdash f\,z : (\mu X.A[\overrightarrow{\Box_r \alpha_i / \alpha_i}])}\,\textsc{app} \tag{B.18}$$

$$\dfrac{\dfrac{}{\Sigma \vdash [\![A]\!]^{\Sigma, X \mapsto f \mu X.\Box_r A \multimap (\mu X.A[\overrightarrow{\Box_r \alpha_i / \alpha_i}])}_{\mathsf{push}} : \mu X.\Box_r A \multimap (\mu X.A[\overrightarrow{\Box_r \alpha_i / \alpha_i}])}\,\textsc{push} \qquad \text{(B.31)}}{\Sigma, z : (\mu X.\Box_r A) \vdash \mathbf{letrec}\ f\ =\ [\![A]\!]^{\Sigma, X \mapsto f \mu X.\Box_r A \multimap (\mu X.A[\overrightarrow{\Box_r \alpha_i / \alpha_i}])}_{\mathsf{push}}\ \mathbf{in}\ f\,z : \mu X.A[\overrightarrow{\Box_r \alpha_i / \alpha_i}]}\,\textsc{letrec}$$

□

Proposition 2 (Type soundness of $[\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{pull}}$)**.** $\mathsf{F}\ (\overline{\Box_{r_i} \alpha_i}) \to \Box_{\prod_{i=1}^n r_i} (\mathsf{F}\ \overline{\alpha_i})$

Proof.

- $[\![\mathsf{Unit}]\!]^{\Sigma}_{\mathsf{pull}} : \mathsf{Unit} \to \Box_{(\prod_1^n r_i)} \mathsf{Unit}$ (i.e. $\mathsf{F}\ \overline{\alpha_i} = \mathsf{Unit}$).

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash () : \mathsf{Unit}}\,\textsc{con}}{\varnothing \vdash [()] : \Box_{\prod_1^n r_i} \mathsf{Unit}}\,\textsc{pr} \qquad \dfrac{|\mathsf{Unit} = 1|}{- \vdash () : \mathsf{Unit} \rhd \varnothing}\,\textsc{Pcon}}{z : \mathsf{Unit} \vdash \mathbf{case}\ z\ \mathbf{of}\ () \to [()] : \Box_{\prod_1^n r_i} \mathsf{Unit}}\,\textsc{case}$$

- $[\![X]\!]^{\Sigma}_{\mathsf{pull}} : X \to \Box_{(\prod_1^n r_i)} X$ (i.e. $\mathsf{F}\ \overline{\alpha_i} = X$).

$$\dfrac{\dfrac{X : \mu X.A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \multimap \Box_{\prod_1^n r_i}(\mu X.A) \in \Sigma}{\Sigma \vdash \Sigma(X)\ : \mu X.A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \multimap \Box_{\prod_1^n r_i}(\mu X.A)}\ \textsc{lookup} \qquad \dfrac{}{z : (\mu X.A)[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \vdash z : (\mu X.A)[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}]}\ \textsc{var}}{\Sigma, z : (\mu X.A)[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \vdash \Sigma(X)\ z : \Box_{\prod_1^n r_i}(\mu X.A)}\ \textsc{app}$$

- $[\![\alpha_j]\!]_{\mathsf{pull}} : \Box_{r_j}\alpha_j \to \Box_{(\prod_1^n r_i)}\alpha_j$ (i.e. $\mathsf{F}\ \overline{\alpha_i} = \alpha$).

$$\dfrac{\dfrac{}{\Sigma, z : \Box_{r_j}\alpha_j \vdash z : \Box_{r_j}\alpha_j}\ \textsc{var}}{\Sigma, z : \Box_{r_j}\alpha_j \vdash z : \Box_{(\prod_1^n r_i)}\alpha_j}\ \textsc{approx}$$

- $[\![(A \oplus B)[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}]]\!]^{\Sigma}_{\mathsf{pull}} : A \oplus B \to \Box_{(\prod_1^n r_i)} A \oplus B$ (i.e. $\mathsf{F}\ \overline{\alpha_i} = A \oplus B$).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x' : A \vdash x' : A}\ \textsc{var}}{x' :_1 A \vdash x' : A}\ \textsc{der}}{x' :_1 A \vdash \mathbf{inl}\,(x') : A \oplus B}\ \textsc{Con}}{x' :_{\prod_1^n r_i} A \vdash [\mathbf{inl}\,(x')] : \Box_{\prod_1^n r_i} A \oplus B}\ \textsc{pr} \tag{B.19}$$

$$\dfrac{\dfrac{}{\varnothing \vdash [\![A]\!]^{\Sigma}_{\mathsf{pull}} : A \multimap \Box_{\prod_1^n r_i} A}\ \textsc{pull} \qquad \dfrac{\dfrac{\dfrac{}{x : A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \vdash x : A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}]}\ \textsc{var}}{x : A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \vdash [\![A]\!]^{\Sigma}_{\mathsf{pull}}(x) : \Box_{\prod_1^n r_i} A}\ \textsc{app} \qquad (\text{B.19}) \qquad \dfrac{\dfrac{}{\prod_1^n r_i \vdash x' : A \rhd x' :_{\prod_1^n r_i} A}\ [\textsc{pvar}]}{- \vdash [x'] : \Box_{\prod_1^n r_i} A \rhd x' :_{\prod_1^n r_i} A}\ [\textsc{pbox}]}{x : A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \vdash \mathbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{pull}}(x)\ \mathbf{of}\ [x'] \to [\mathbf{inl}\,(x')] : \Box_{\prod_1^n r_i} A \oplus B}\ \textsc{case} \tag{B.20}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y' : B \vdash y' : B}\ \textsc{var}}{y' :_1 B \vdash y' : B}\ \textsc{der}}{y' :_1 B \vdash \mathbf{inr}\,(y') : A \oplus B}\ \textsc{Con}}{y' :_{\prod_1^n r_i} B \vdash [\mathbf{inr}\,(y')] : \Box_{\prod_1^n r_i} A \oplus B}\ \textsc{pr} \tag{B.21}$$

$$\dfrac{\dfrac{}{\varnothing \vdash [\![B]\!]^{\Sigma}_{\mathsf{pull}} : B \multimap \Box_{\prod_1^n r_i} B}\ \textsc{pull} \qquad \dfrac{\dfrac{\dfrac{}{y : B[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \vdash y : B[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}]}\ \textsc{var}}{y : B[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \vdash [\![B]\!]_{\mathsf{pull}}(y) : \Box_{\prod_1^n r_i} B}\ \textsc{app} \qquad (\text{B.21}) \qquad \dfrac{\dfrac{}{\prod_1^n r_i \vdash y' : B \rhd y' :_{\prod_1^n r_i} B}\ [\textsc{pvar}]}{- \vdash [y'] : \Box_{\prod_1^n r_i} B \rhd y' :_{\prod_1^n r_i} B}\ [\textsc{pbox}]}{y : B[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \vdash \mathbf{case}\ [\![B]\!]^{\Sigma}_{\mathsf{pull}}(y)\ \mathbf{of}\ [y'] \to [\mathbf{inr}\,(x'_2)] : \Box_{\prod_1^n r_i} A \oplus B}\ \textsc{case} \tag{B.22}$$

$$\dfrac{\dfrac{}{- \vdash x : A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \rhd x : A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}]}\ \textsc{pvar}}{- \vdash \mathbf{inl}\,(x) : (A \oplus B)[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \rhd x : A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}]}\ \textsc{Pcon} \tag{B.23}$$

$$\frac{\overline{- \vdash y : B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \rhd y : B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}]}\ \textsc{pvar}}{- \vdash \mathbf{inr}\,(x) : (A \oplus B)[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \rhd y : B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}]}\ \textsc{Pcon} \tag{B.24}$$

$$\mathbf{case}\ z\ \mathbf{of}\ \mathbf{inl}\,(x) \to (\mathbf{case}\ [\![A]\!]^{\Sigma}_{\mathsf{pull}}(x)\ \mathbf{of}\ [x'] \to [\mathbf{inl}\,(x')]); \mathbf{inr}\,(y) \to (\mathbf{case}\ [\![B]\!]^{\Sigma}_{\mathsf{pull}}(y)\ \mathbf{of}\ [y'] \to [\mathbf{inr}\,(y')]) \tag{B.25}$$

$$\frac{\text{(B.20)} \qquad \text{(B.22)} \qquad\qquad \text{(B.23)} \qquad \text{(B.24)}}{z : A \oplus B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \vdash \text{(B.25)} : \square_{\prod_1^n r_i} A \oplus B}\ \textsc{case}$$

- $[\![A \otimes B]\!]^{\Sigma}_{\mathsf{pull}} : (A \otimes B)[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \to \square_{(\prod_1^n r_i)}(A \otimes B)$ (i.e. $\mathsf{F}\ \overline{\alpha_i} = A \otimes B$).

$$\frac{\dfrac{\dfrac{\overline{x' : A \vdash x' : A}\ \textsc{var}}{x' :_1 A \vdash x' : A}\ \textsc{der} \qquad \dfrac{\overline{y' : B \vdash y' : B}\ \textsc{var}}{y' :_1 B \vdash y' : B}\ \textsc{der}}{x' :_1 A, y' :_1 B \vdash (x', y') : A \otimes B}\ \textsc{Con}}{x' :_{\prod_1^n r_i} A, y' :_{\prod_1^n r_i} B \vdash [(x', y')] : \square_{\prod_1^n r_i} A \otimes B}\ \textsc{pr} \tag{B.26}$$

$$\frac{\dfrac{\dfrac{\dfrac{}{\prod_1^n r_i \vdash x' : A \rhd x' :_{\prod_1^n r_i} A}\ [\textsc{pvar}]}{- \vdash [x'] : \square_{\prod_1^n r_i} A \rhd x' :_{\prod_1^n r_i} A}\ [\textsc{pbox}] \qquad \dfrac{\dfrac{}{\prod_1^n r_i \vdash y' : B \rhd y' :_{\prod_1^n r_i} B}\ [\textsc{pvar}]}{- \vdash [y'] : \square_{\prod_1^n r_i} B \rhd y' :_{\prod_1^n r_i} B}\ [\textsc{pbox}]\ |A \otimes B| = 1}{- \vdash ([x'], [y']) : (\square_{\prod_1^n r_i} A) \otimes (\square_{\prod_1^n r_i} B) \rhd x' :_{\prod_1^n r_i} A, y' :_{\prod_1^n r_i} B}\ [\textsc{Pcon}] \tag{B.27}$$

$$\frac{\overline{\varnothing \vdash [\![A]\!]^{\Sigma}_{\mathsf{pull}} : A \multimap \square_{\prod_1^n r_i} A}\ \textsc{pull} \qquad \overline{x : A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \vdash x : A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}]}\ \textsc{var}}{x : A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \vdash [\![A]\!]^{\Sigma}_{\mathsf{pull}}(x) : \square_{\prod_1^n r_i} A}\ \textsc{app} \tag{B.28}$$

$$\frac{\overline{\varnothing \vdash [\![B]\!]^{\Sigma}_{\mathsf{pull}} : B \multimap \square_{\prod_1^n r_i} B}\ \textsc{pull} \qquad \overline{y : B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \vdash y : B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}]}\ \textsc{var}}{y : B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \vdash [\![B]\!]^{\Sigma}_{\mathsf{pull}}(y) : \square_{\prod_1^n r_i} B}\ \textsc{app} \tag{B.29}$$

$$\dfrac{\dfrac{}{x : A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}], y : B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \vdash (\llbracket A\rrbracket^{\Sigma}_{\mathsf{pull}}(x), \llbracket B\rrbracket^{\Sigma}_{\mathsf{pull}}(y)) : (\square_{\prod_1^n r_i} A) \otimes (\square_{\prod_1^n r_i} B)}\ \textsc{pair}\quad (B.26)\quad (B.27)}{x : A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}], y : B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \vdash \mathbf{case}\ (\llbracket A\rrbracket^{\Sigma}_{\mathsf{pull}} x, \llbracket B\rrbracket^{\Sigma}_{\mathsf{pull}} y)\ \mathbf{of}\ ([x'], [y']) \to [(x', y')] : \square_{\prod_1^n r_i} A \otimes B}\ \textsc{case}$$

(B.28) (B.29)

(B.30)

$$\dfrac{\dfrac{\dfrac{}{- \vdash x : A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \rhd x : A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}]}\ \textsc{pvar} \quad \dfrac{}{- \vdash y : B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \rhd y : B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}]}\ \textsc{pvar}}{- \vdash (x, y) : (A \otimes B)[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \rhd x : A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}], y : B[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}]}\ \textsc{pcon}}{z : (A \otimes B)[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \vdash \mathbf{case}\ z\ \mathbf{of}\ (x, y) \to (\mathbf{case}\ (\llbracket A\rrbracket^{\Sigma}_{\mathsf{pull}} x, \llbracket B\rrbracket^{\Sigma}_{\mathsf{pull}} y)\ \mathbf{of}\ ([x'], [y']) \to [(x', y')]) : \square_{\prod_1^n r_i} A \otimes B}\ \textsc{case}$$

- $\llbracket \mu X.A\rrbracket^{\Sigma}_{\mathsf{pull}} : (\mu X.A)[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \to \square_{(\prod_1^n r_i)}(\mu X.A)$ (i.e. $\mathsf{F}\ \overline{\alpha_i} = \mu X.A$).

$$\dfrac{\dfrac{}{\Sigma, f : \mu X.A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \multimap \square_{\prod_1^n r_i}(\mu X.A) \vdash f : \mu X.A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \multimap \square_{\prod_1^n r_i}(\mu X.A)}\ \textsc{var} \quad \dfrac{}{\Sigma, z : \mu X.A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \vdash z : \mu X.A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}]}\ \textsc{var}}{\Sigma, f : \mu X.A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \multimap \square_{\prod_1^n r_i}(\mu X.A), z : \mu X.A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \vdash f\, z : \square_{\prod_1^n r_i}(\mu X.A)}\ \textsc{app}$$

(B.31)

$$\dfrac{\dfrac{}{\Sigma \vdash \llbracket A\rrbracket^{\Sigma, X \mapsto f\mu X.A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \multimap \square_{\prod_1^n r_i}(\mu X.A)}_{\mathsf{pull}} : \mu X.A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \multimap \square_{\prod_1^n r_i}(\mu X.A)}\ \textsc{pull} \quad (B.31)}{\Sigma, z : (\mu X.A)[\overrightarrow{r_i\,\alpha_i/\alpha_i}] \vdash \mathbf{letrec}\ f\ =\ \llbracket A\rrbracket^{\Sigma, X \mapsto f\mu X.A[\overrightarrow{\square_{r_i}\alpha_i/\alpha_i}] \multimap \square_{\prod_1^n r_i}(\mu X.A)}_{\mathsf{pull}}\ \mathbf{in}\ f\, z : \square_{\prod_1^n r_i}\mu X.A}\ \textsc{letrec}$$

□

B.2.4 *Inverse Property of the Distributive Laws*

Proposition 5.3.1 (Pull is right inverse to push). *For all n-arity types* F *which do not contain function types, then for type variables* $(\alpha_i)_{i \in 1 \le i \le n}$ *and for all grades* $r \in \mathcal{R}$ *where* $1 \sqsubseteq r$ *if* $|\mathsf{F}\overline{\alpha_i}| > 1$*, then:*

$$\llbracket \mathsf{F}\ \overline{\alpha_i}\rrbracket_{\mathsf{pull}}(\llbracket \mathsf{F}\ \overline{\alpha_i}\rrbracket_{\mathsf{push}}) = id\ : \square_r \mathsf{F}\overline{\alpha_i} \multimap \square_r \mathsf{F}\overline{\alpha_i}$$

Proof. By induction on the syntax of the type $\mathsf{F}\overline{\alpha_i}$ which we denote by T in the following. We first prove a subresult that for $\llbracket T\rrbracket^{\Sigma'}_{\mathsf{pull}}(\llbracket T\rrbracket^{\Sigma}_{\mathsf{push}}\ z) \equiv z$, which by function extensionality then gives us $\llbracket T\rrbracket^{\Sigma'}_{\mathsf{pull}}(\llbracket T\rrbracket^{\Sigma}_{\mathsf{push}}) \equiv id$, under the assumption that for all X, every $f \in \Sigma(X)$ and $g \in \Sigma'(X)$ then $g \circ f = id$, in order to apply the recursive argument.

- $T = \mathsf{Unit}$

$$
\begin{array}{lll}
& [\![\mathsf{Unit}]\!]^{\Sigma'}_{\mathsf{pull}}([\![\mathsf{Unit}]\!]^{\Sigma}_{\mathsf{push}}\ z) & \\
\equiv & [\![\mathsf{Unit}]\!]^{\Sigma'}_{\mathsf{pull}}(\mathbf{case}\ z\ \mathbf{of}\ [()] \to ()) & \{\textit{defn. } [\![\mathsf{Unit}]\!]^{\Sigma}_{\mathsf{push}}\} \\
\equiv & \mathbf{case}\ (\mathbf{case}\ z\ \mathbf{of}\ [()] \to ())\ \mathbf{of}\ () \to [()] & \{\textit{defn. } [\![\mathsf{Unit}]\!]^{\Sigma'}_{\mathsf{pull}}\} \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ [()] \to \mathbf{case}\ ()\ \mathbf{of}\ () \to [()] & \{\textit{case assoc.}\} \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ [()] \to [()] & \{\beta_{case}\} \\
\equiv & z & \{\eta_{case}\}
\end{array}
$$

- $T = \alpha$

$$
\begin{array}{lll}
& [\![\alpha]\!]^{\Sigma'}_{\mathsf{pull}}([\![\alpha]\!]^{\Sigma}_{\mathsf{push}}\ z) & \\
\equiv & [\![\alpha]\!]^{\Sigma'}_{\mathsf{pull}}(z) & \{\textit{defn. } [\![\alpha]\!]^{\Sigma}_{\mathsf{push}}\} \\
\equiv & z & \{\textit{defn. } [\![\alpha]\!]^{\Sigma'}_{\mathsf{pull}}\}
\end{array}
$$

- $T = X$

$$
\begin{array}{lll}
& [\![X]\!]^{\Sigma'}_{\mathsf{pull}}([\![X]\!]^{\Sigma}_{\mathsf{push}}\ z) & \\
\equiv & [\![X]\!]^{\Sigma'}_{\mathsf{pull}}(\Sigma(X)z) & \{\textit{defn. } [\![X]\!]^{\Sigma}_{\mathsf{push}}\} \\
\equiv & \Sigma'(X)(\Sigma(X)z) & \{\textit{defn. } [\![X]\!]^{\Sigma'}_{\mathsf{pull}}\} \\
\equiv & z & \{\textit{recursion assumption}\}
\end{array}
$$

- $T = A \oplus B$:

$$
\begin{array}{lll}
& [\![A \oplus B]\!]^{\Sigma'}_{\mathsf{pull}}([\![A \oplus B]\!]^{\Sigma}_{\mathsf{push}}\ z) & \\
\equiv & [\![A \oplus B]\!]^{\Sigma'}_{\mathsf{pull}}(\mathbf{case}\ z\ \mathbf{of}\ \ [\mathsf{inl}\ x] \to \mathsf{inl}\ [\![A]\!]^{\Sigma}_{\mathsf{push}}[x];\ \ [\mathsf{inr}\ y] \to \mathsf{inr}\ [\![B]\!]^{\Sigma}_{\mathsf{push}}[y]\ \ ) & \{\textit{defn. } [\![A \oplus B]\!]^{\Sigma}_{\mathsf{push}}\} \\
\equiv & \ \ \mathbf{case}\ (\mathbf{case}\ z\ \mathbf{of}\ \ [\mathsf{inl}\ x] \to \mathsf{inl}\ [\![A]\!]^{\Sigma}_{\mathsf{push}}[x];\ \ [\mathsf{inr}\ y] \to \mathsf{inr}\ [\![B]\!]^{\Sigma}_{\mathsf{push}}[y]\ \ )\ \mathbf{of} & \{\textit{defn. } [\![A \oplus B]\!]^{\Sigma'}_{\mathsf{pull}}\} \\
& \quad \mathsf{inl}\ x \to \mathbf{case}\ [\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x\ \mathbf{of}\ [u] \to [\mathsf{inl}\ u]; & \\
& \quad \mathsf{inr}\ y \to \mathbf{case}\ [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y\ \mathbf{of}\ [v] \to [\mathsf{inr}\ v] & \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ \ [\mathsf{inl}\ x] \to \mathbf{case}\ \mathsf{inl}\ [\![A]\!]^{\Sigma}_{\mathsf{push}}[x]\ \mathbf{of}\ \ \mathsf{inl}\ x \to \mathbf{case}\ [\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x\ \mathbf{of}\ [u] \to [\mathsf{inl}\ u];\ ; & \{\textit{case assoc.}\} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\ y \to \mathbf{case}\ [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y\ \mathbf{of}\ [v] \to [\mathsf{inr}\ v] & \\
& \qquad\quad [\mathsf{inr}\ y] \to \mathbf{case}\ \mathsf{inr}\ [\![B]\!]^{\Sigma}_{\mathsf{push}}[y]\ \mathbf{of}\ \ \mathsf{inl}\ x \to \mathbf{case}\ [\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x\ \mathbf{of}\ [u] \to [\mathsf{inl}\ u]; & \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}\ y \to \mathbf{case}\ [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y\ \mathbf{of}\ [v] \to [\mathsf{inr}\ v] & \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ \ [\mathsf{inl}\ x] \to \mathbf{case}\ [\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ [\![A]\!]^{\Sigma}_{\mathsf{push}}[x]\ \mathbf{of}\ [u] \to [\mathsf{inl}\ u]; & \{\beta_{case}\} \\
& \qquad\quad [\mathsf{inr}\ y] \to \mathbf{case}\ [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ [\![B]\!]^{\Sigma}_{\mathsf{push}}[y]\ \mathbf{of}\ [v] \to [\mathsf{inr}\ v] & \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ \ [\mathsf{inl}\ x] \to \mathbf{case}\ [x]\ \mathbf{of}\ [u] \to [\mathsf{inl}\ u]; & \{\textit{induction}\} \\
& \qquad\quad [\mathsf{inr}\ y] \to \mathbf{case}\ [y]\ \mathbf{of}\ [v] \to [\mathsf{inr}\ v] & \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ \ [\mathsf{inl}\ x] \to [\mathsf{inl}\ x];\ \ [\mathsf{inr}\ y] \to [\mathsf{inr}\ y] & \{\beta_{case}\} \\
\equiv & z & \{\eta_{case}\}
\end{array}
$$

$T = A \otimes B$:

$$
\begin{array}{lll}
& [\![A \otimes B]\!]^{\Sigma'}_{\mathsf{pull}}([\![A \otimes B]\!]^{\Sigma}_{\mathsf{push}}\ z) & \\
\equiv & [\![A \otimes B]\!]^{\Sigma'}_{\mathsf{pull}}(\mathbf{case}\ z\ \mathbf{of}\ [(x,y)] \to ([\![A]\!]^{\Sigma}_{\mathsf{push}}[x], [\![B]\!]^{\Sigma}_{\mathsf{push}}[y])) & \{\textit{defn. } [\![A \otimes B]\!]^{\Sigma}_{\mathsf{push}}\} \\
\equiv & \mathbf{case}\ (\mathbf{case}\ z\ \mathbf{of}\ [(x,y)] \to ([\![A]\!]^{\Sigma}_{\mathsf{push}}[x], [\![B]\!]^{\Sigma}_{\mathsf{push}}[y]))\ \mathbf{of}\ (x,y) \to & \\
& \quad \mathbf{case}\ ([\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x, [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y)\ \mathbf{of}\ ([u],[v]) \to [(u,v)] & \{\textit{defn. } [\![A \otimes B]\!]^{\Sigma'}_{\mathsf{pull}}\} \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ [(x,y)] \to \mathbf{case}\ ([\![A]\!]^{\Sigma}_{\mathsf{push}}[x], [\![B]\!]^{\Sigma}_{\mathsf{push}}[y])\ \mathbf{of}\ (x,y) \to & \\
& \qquad\qquad \mathbf{case}\ ([\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x, [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y)\ \mathbf{of}\ ([u],[v]) \to [(u,v)] & \{\textit{case assoc.}\} \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ [(x,y)] \to (\mathbf{case}\ ([\![A]\!]^{\Sigma'}_{\mathsf{pull}}[\![A]\!]^{\Sigma}_{\mathsf{push}}\ [x], [\![B]\!]^{\Sigma'}_{\mathsf{pull}}[\![B]\!]^{\Sigma}_{\mathsf{push}}\ [y])\ \mathbf{of}\ ([u],[v]) \to [(u,v)]) & \{\beta_{case}\} \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ [(x,y)] \to (\mathbf{case}\ ([x],[y])\ \mathbf{of}\ ([u],[v]) \to [(u,v)]) & \{\textit{induction}\} \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ [(x,y)] \to [(x,y)] & \{\beta_{case}\} \\
\equiv & z & \{\eta_{case}\}
\end{array}
$$

$T = \mu X.A$:

$$
\begin{array}{lll}
& [\![\mu X.A]\!]^{\Sigma'}_{\mathsf{pull}}([\![\mu X.A]\!]^{\Sigma}_{\mathsf{push}}\ z) & \\
\equiv & [\![\mu X.A]\!]^{\Sigma'}_{\mathsf{pull}}(\mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X\mapsto f:\mu X.\Box_r A\multimap(\mu X.A)\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}_{\mathsf{push}}\ \mathbf{in}\ f\ z) & \{\textit{defn. } [\![\mu X.A]\!]^{\Sigma}_{\mathsf{push}}\} \\
\equiv & \mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X\mapsto f':\mu X.A\overrightarrow{[\Box_{r_i}\alpha_i/\alpha_i]}\multimap\Box_{\prod_1^n r_i}(\mu X.A)}_{\mathsf{pull}}\ \mathbf{in} & \\
& \quad f'(\mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X\mapsto f:\mu X.\Box_r A\multimap(\mu X.A)\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}_{\mathsf{push}}\ \mathbf{in}\ f\ z) & \{\textit{defn. } [\![\mu X.A]\!]^{\Sigma'}_{\mathsf{pull}}\} \\
\equiv & \mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X\mapsto f':\mu X.A\overrightarrow{[\Box_{r_i}\alpha_i/\alpha_i]}\multimap\Box_{\prod_1^n r_i}(\mu X.A)}_{\mathsf{pull}}\ \mathbf{in} & \\
& \quad \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X\mapsto f:\mu X.\Box_r A\multimap(\mu X.A)\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}_{\mathsf{push}}\ \mathbf{in}\ f'(f\ z) & \{\textit{let dist. }\} \\
\equiv & \mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X\mapsto f':\mu X.A\overrightarrow{[\Box_{r_i}\alpha_i/\alpha_i]}\multimap\Box_{\prod_1^n r_i}(\mu X.A)}_{\mathsf{pull}}\ \mathbf{in} & \\
& \quad \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X\mapsto f:\mu X.\Box_r A\multimap(\mu X.A)\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}_{\mathsf{push}}\ \mathbf{in}\ f'([\![A]\!]^{\Sigma, X\mapsto f:\mu X.\Box_r A\multimap(\mu X.A)\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}_{\mathsf{push}}\ z) & \{\beta_{letrec}\} \\
\equiv & \mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X\mapsto f'}_{\mathsf{pull}}\ \mathbf{in} & \\
& \quad \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X\mapsto f:\mu X.\Box_r A\multimap(\mu X.A)\overrightarrow{[\Box_r\alpha_i/\alpha_i]}}_{\mathsf{push}}\ \mathbf{in}\ [\![A]\!]^{\Sigma', X\mapsto f'}_{\mathsf{pull}}([\![A]\!]^{\Sigma, X\mapsto f}_{\mathsf{push}}\ z) & \{\beta_{letrec}\} \\
\equiv & \mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X\mapsto f'}_{\mathsf{pull}}\ \mathbf{in} & \\
& \quad [\![A]\!]^{\Sigma', X\mapsto f'}_{\mathsf{pull}}([\![A]\!]^{\Sigma, X\mapsto \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X\mapsto f}_{\mathsf{push}}\ \mathbf{in}\ f}_{\mathsf{push}}\ z) & \{\beta_{letrec}\} \\
\equiv & [\![A]\!]^{\Sigma', X\mapsto \mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X\mapsto f'}_{\mathsf{pull}}\ \mathbf{in}\ f'}_{\mathsf{pull}}([\![A]\!]^{\Sigma, X\mapsto \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X\mapsto f}_{\mathsf{push}}\ \mathbf{in}\ f}_{\mathsf{push}}\ z) & \{\beta_{letrec}\} \\
\equiv & z & \{\textit{induction}\}
\end{array}
$$

□

Proposition 5.3.2 (Pull is left inverse to push)**.** *For all n-arity types* F *which do not contain function types, then for type variables* $(\alpha_i)_{i\in 1\leq i\leq n}$ *and for all grades* $r \in \mathcal{R}$ *where* $1 \sqsubseteq r$ *if* $|\mathsf{F}\overline{\alpha_i}| > 1$*, then:*

$$[\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{push}}([\![\mathsf{F}\ \overline{\alpha_i}]\!]_{\mathsf{pull}}) = id\ : \mathsf{F}(\Box_r\overline{\alpha_i}) \multimap \mathsf{F}(\Box_r\overline{\alpha_i})$$

Proof. By induction on the syntax of the type $\mathsf{F}\overline{\alpha_i}$ which we denote by T in the following. The following proof is for $[\![T]\!]^{\Sigma}_{\mathsf{push}}([\![T]\!]^{\Sigma}_{\mathsf{pull}}\ z) \equiv z$, which by function extensionality then gives us $[\![T]\!]^{\Sigma}_{\mathsf{push}}([\![T]\!]^{\Sigma}_{\mathsf{pull}}) \equiv id$, under the assumption that for all X, every $f \in \Sigma(X)$ and $g \in \Sigma'(X)$ then $g \circ f = id$, in order to apply the recursive argument.

- $T = 1$

$$
\begin{array}{lll}
& [\![\mathsf{Unit}]\!]^{\Sigma}_{\mathsf{push}}([\![\mathsf{Unit}]\!]^{\Sigma'}_{\mathsf{pull}}\ z) & \\
\equiv & [\![\mathsf{Unit}]\!]^{\Sigma}_{\mathsf{push}}(\mathbf{case}\ z\ \mathbf{of}\ ()\to[()]) & \{\textit{defn. } [\![\mathsf{Unit}]\!]^{\Sigma'}_{\mathsf{pull}}\} \\
\equiv & \mathbf{case}\ (\mathbf{case}\ z\ \mathbf{of}\ ()\to[()])\ \mathbf{of}\ [()]\to() & \{\textit{defn. } [\![1]\!]^{\Sigma}_{\mathsf{push}}\} \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ ()\to\mathbf{case}\ [()]\ \mathbf{of}\ [()]\to() & \{\textit{case assoc.}\} \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ ()\to() & \{\beta_{case}\} \\
\equiv & z & \{\eta_{case}\}
\end{array}
$$

- $T = \alpha$

$$
\begin{array}{lll}
& [\![\alpha]\!]^{\Sigma}_{\mathsf{push}}([\![\alpha]\!]^{\Sigma'}_{\mathsf{pull}}\ z) & \\
\equiv & [\![\alpha]\!]^{\Sigma}_{\mathsf{push}}(z) & \{\textit{defn. } [\![\alpha]\!]^{\Sigma'}_{\mathsf{pull}}\} \\
\equiv & z & \{\textit{defn. } [\![\alpha]\!]^{\Sigma}_{\mathsf{push}}\}
\end{array}
$$

- $T = X$

$$
\begin{array}{lll}
& [\![X]\!]^{\Sigma}_{\mathsf{push}}([\![X]\!]^{\Sigma'}_{\mathsf{pull}}\ z) & \\
\equiv & [\![X]\!]^{\Sigma}_{\mathsf{push}}(\Sigma'(X)z) & \{\textit{defn. } [\![X]\!]^{\Sigma'}_{\mathsf{pull}}\} \\
\equiv & \Sigma(X)(\Sigma'(X)z) & \{\textit{defn. } [\![X]\!]^{\Sigma}_{\mathsf{push}}\} \\
\equiv & z & \{\textit{recursion assumption}\}
\end{array}
$$

- $T = A \oplus B$

$$
\begin{array}{lll}
& [\![A \oplus B]\!]^{\Sigma}_{\mathsf{push}}([\![A \oplus B]\!]^{\Sigma'}_{\mathsf{pull}}\ z) & \\
\equiv & [\![A \oplus B]\!]^{\Sigma}_{\mathsf{push}}\mathbf{case}\ z\ \mathbf{of}\ \ \mathsf{inl}\ x \to \mathbf{case}\ [\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x\ \mathbf{of}\ [u] \to [\mathsf{inl}\ u]; & \{\textit{defn. } [\![A \oplus B]\!]^{\Sigma'}_{\mathsf{pull}}\} \\
& \qquad\qquad\qquad\qquad\quad \mathsf{inr}\ y \to \mathbf{case}\ [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y\ \mathbf{of}\ [v] \to [\mathsf{inr}\ v] & \\
\equiv & \mathbf{case}\ (\mathbf{case}\ z\ \mathbf{of}\ \ \mathsf{inl}\ x \to \mathbf{case}\ [\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x\ \mathbf{of}\ [u] \to [\mathsf{inl}\ u];\ )\ \mathbf{of}\ \ [\mathsf{inl}\ x] \to \mathsf{inl}\ [\![A]\!]^{\Sigma}_{\mathsf{push}}[x]; & \{\textit{defn. } [\![A \oplus B]\!]^{\Sigma}_{\mathsf{push}}\} \\
& \qquad\qquad\qquad\quad \mathsf{inr}\ y \to \mathbf{case}\ [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y\ \mathbf{of}\ [v] \to [\mathsf{inr}\ v] \qquad\quad [\mathsf{inr}\ y] \to \mathsf{inr}\ [\![B]\!]^{\Sigma}_{\mathsf{push}}[y] & \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ \ \mathsf{inl}\ x \to \mathbf{case\ case}\ [\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x\ \mathbf{of}\ [u] \to [\mathsf{inl}\ u]\ \mathbf{of}\ \ [\mathsf{inl}\ x] \to \mathsf{inl}\ [\![A]\!]^{\Sigma}_{\mathsf{push}}[x];\ ; & \{\textit{case assoc.}\} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\mathsf{inr}\ y] \to \mathsf{inr}\ [\![B]\!]^{\Sigma}_{\mathsf{push}}[y] & \\
& \qquad\qquad\quad \mathsf{inr}\ y \to \mathbf{case\ case}\ [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y\ \mathbf{of}\ [v] \to [\mathsf{inr}\ v]\ \mathbf{of}\ \ [\mathsf{inl}\ x] \to \mathsf{inl}\ [\![A]\!]^{\Sigma}_{\mathsf{push}}[x]; & \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\mathsf{inr}\ y] \to \mathsf{inr}\ [\![B]\!]^{\Sigma}_{\mathsf{push}}[y] & \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ \ \mathsf{inl}\ x \to \mathsf{inl}\ [\![A]\!]^{\Sigma}_{\mathsf{push}}[\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x; & \{\beta_{case}\} \\
& \qquad\qquad\quad \mathsf{inr}\ y \to \mathsf{inr}\ [\![B]\!]^{\Sigma}_{\mathsf{push}}[\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y & \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ \ \mathsf{inl}\ x \to \mathsf{inl}\ x; & \{\textit{induction}\} \\
& \qquad\qquad\quad \mathsf{inr}\ y \to \mathsf{inr}\ y & \\
\equiv & z & \{\eta_{case}\}
\end{array}
$$

- $T = A \otimes B$

$$
\begin{array}{lll}
& [\![A \otimes B]\!]^{\Sigma}_{\mathsf{push}}([\![A \otimes B]\!]^{\Sigma'}_{\mathsf{pull}}\ z) & \\
\equiv & [\![A \otimes B]\!]^{\Sigma}_{\mathsf{push}}(\mathbf{case}\ z\ \mathbf{of}\ (x,y) \to \mathbf{case}\ ([\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x, [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y)\ \mathbf{of}\ ([u],[v]) \to [(u,v)]) & \{\textit{defn. } [\![A \otimes B]\!]^{\Sigma'}_{\mathsf{pull}}\} \\
\equiv & \mathbf{case}\ (\mathbf{case}\ z\ \mathbf{of}\ (x,y) \to & \\
& \quad \mathbf{case}\ ([\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x, [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y)\ \mathbf{of}\ ([u],[v]) \to [(u,v)])\ \mathbf{of}\ [(x,y)] \to ([\![A]\!]^{\Sigma}_{\mathsf{push}}[x], [\![B]\!]^{\Sigma}_{\mathsf{push}}[y]) & \{\textit{defn. } [\![A \otimes B]\!]^{\Sigma}_{\mathsf{push}}\} \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ (x,y) \to & \\
& \quad \mathbf{case}\ ([\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x, [\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y)\ \mathbf{of}\ ([u],[v]) \to \mathbf{case}\ [(u,v)]\ \mathbf{of}\ [(x,y)] \to ([\![A]\!]^{\Sigma}_{\mathsf{push}}[x], [\![B]\!]^{\Sigma}_{\mathsf{push}}[y]) & \{\textit{case assoc.}\} \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ (x,y) \to ([\![A]\!]^{\Sigma}_{\mathsf{push}}[\![A]\!]^{\Sigma'}_{\mathsf{pull}}\ x, [\![B]\!]^{\Sigma}_{\mathsf{push}}[\![B]\!]^{\Sigma'}_{\mathsf{pull}}\ y) & \{\beta_{case}\} \\
\equiv & \mathbf{case}\ z\ \mathbf{of}\ (x,y) \to (x,y) & \{\textit{induction}\} \\
\equiv & z & \{\eta_{case}\}
\end{array}
$$

- $T = \mu X.A$

$$
\begin{aligned}
& [\![\mu X.A]\!]^{\Sigma}_{\mathsf{push}}([\![\mu X.A]\!]^{\Sigma'}_{\mathsf{pull}}\ z) \\
\equiv\ & [\![\mu X.A]\!]^{\Sigma}_{\mathsf{push}}(\mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X \mapsto f' : \mu X.A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \multimap \Box_{\prod_1^n r_i}(\mu X.A)}_{\mathsf{pull}}\ \mathbf{in}\ f'\ z) && \{\textit{defn. } [\![\mu X.A]\!]^{\Sigma'}_{\mathsf{pull}}\} \\
\equiv\ & \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \mu X.\Box_r A \multimap (\mu X.A)[\overrightarrow{\Box_r \alpha_i/\alpha_i}]}_{\mathsf{push}}\ \mathbf{in} \\
& \qquad f\ (\mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X \mapsto f' : \mu X.A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \multimap \Box_{\prod_1^n r_i}(\mu X.A)}_{\mathsf{pull}}\ \mathbf{in}\ f'\ z) && \{\textit{defn. } [\![\mu X.A]\!]^{\Sigma}_{\mathsf{push}}\} \\
\equiv\ & \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \mu X.\Box_r A \multimap (\mu X.A)[\overrightarrow{\Box_r \alpha_i/\alpha_i}]}_{\mathsf{push}}\ \mathbf{in} \\
& \qquad \mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X \mapsto f' : \mu X.A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \multimap \Box_{\prod_1^n r_i}(\mu X.A)}_{\mathsf{pull}}\ \mathbf{in}\ f\ (f'\ z) && \{\textit{let dist. }\} \\
\equiv\ & \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \mu X.\Box_r A \multimap (\mu X.A)[\overrightarrow{\Box_r \alpha_i/\alpha_i}]}_{\mathsf{push}}\ \mathbf{in} \\
& \qquad \mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X \mapsto f' : \mu X.A[\overrightarrow{\Box_{r_i}\alpha_i/\alpha_i}] \multimap \Box_{\prod_1^n r_i}(\mu X.A)}_{\mathsf{pull}}\ \mathbf{in}\ f\ ([\![A]\!]^{\Sigma', X \mapsto f'}_{\mathsf{pull}}\ z) && \{\beta_{letrec}\} \\
\equiv\ & \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \mu X.\Box_r A \multimap (\mu X.A)[\overrightarrow{\Box_r \alpha_i/\alpha_i}]}_{\mathsf{push}}\ \mathbf{in} \\
& \qquad \mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X \mapsto f'}_{\mathsf{pull}}\ \mathbf{in}\ [\![A]\!]^{\Sigma, X \mapsto f : \mu X.\Box_r A \multimap (\mu X.A)[\overrightarrow{\Box_r \alpha_i/\alpha_i}]}_{\mathsf{push}}\ ([\![A]\!]^{\Sigma', X \mapsto f'}_{\mathsf{pull}}\ z) && \{\beta_{letrec}\} \\
\equiv\ & \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f : \mu X.\Box_r A \multimap (\mu X.A)[\overrightarrow{\Box_r \alpha_i/\alpha_i}]}_{\mathsf{push}}\ \mathbf{in} \\
& \qquad [\![A]\!]^{\Sigma, X \mapsto f : \mu X.\Box_r A \multimap (\mu X.A)[\overrightarrow{\Box_r \alpha_i/\alpha_i}]}_{\mathsf{push}}\ ([\![A]\!]^{\Sigma', X \mapsto \mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X \mapsto f'}_{\mathsf{pull}}\ \mathbf{in}\ f'}_{\mathsf{pull}}\ z) && \{\beta_{letrec}\} \\
\equiv\ & [\![A]\!]^{\Sigma, X \mapsto \mathbf{letrec}\ f = [\![A]\!]^{\Sigma, X \mapsto f}_{\mathsf{push}}\ \mathbf{in}\ f}_{\mathsf{push}}\ ([\![A]\!]^{\Sigma', X \mapsto \mathbf{letrec}\ f' = [\![A]\!]^{\Sigma', X \mapsto f'}_{\mathsf{pull}}\ \mathbf{in}\ f'}_{\mathsf{pull}}\ z) && \{\beta_{letrec}\} \\
\equiv\ & z && \{\textit{induction}\}
\end{aligned}
$$

□

B.3 PROOFS FOR THE FULLY GRADED SYNTHESIS CALCULUS

This section contains the soundness proof for the fully graded synthesis calculus presented in Chapter 4, as well as the proof of soundness for focusing this calculus.

B.3.1 *Soundness of the Fully Graded Synthesis Calculus*

Theorem 4.2.1 (Soundness of synthesis)**.** Given a particular pre-ordered semiring $\mathcal{R}$ parametrising the calculi, then:

1. For all contexts Γ and Δ, types A, terms t:

$$\Sigma; \Gamma \vdash A \Rightarrow t \mid \Delta \quad \Longrightarrow \quad \Sigma; \Delta \vdash t : A$$

   i.e. t has type A under context Δ whose grades capture variable use in t.

2. At the top-level, for all type schemes $\forall \overline{\alpha : \kappa}.A$ and terms t then:

$$\emptyset; \emptyset \vdash \forall \overline{\alpha : \kappa}.A \Rightarrow t \mid \emptyset \quad \Longrightarrow \quad \emptyset; \emptyset \vdash t : \forall \overline{\alpha : \kappa}.A$$

Proof. Induction on the synthesis rules. We consider the cases of the lemma in order, first proving soundness for synthesis of open terms from types, followed by soundness of synthesis for closed term from type schemes.

1. a) Case VAR
   For synthesis of a variable term, we have the derivation:

$$\frac{\Sigma \vdash A : \text{Type}}{\Sigma; \Gamma, x :_r A \vdash A \Rightarrow x \mid 0 \cdot \Gamma, x :_1 A}\ \textsc{Var}$$

From the premise, we have that:

$$\Sigma \vdash A : \text{Type}$$

from which we can construct the following typing derivation, matching the above conclusion:

$$\frac{\Sigma \vdash A : \text{Type}}{\Sigma; 0 \cdot \Gamma, x :_1 A \vdash x : A}\ \textsc{Var}$$

b) Case DEF
For synthesis of a top-level definition usage, we have the derivation:

$$\frac{(x : \forall \overline{\alpha : \kappa}.A') \in D \qquad \Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')}{\Sigma; \Gamma \vdash A \Rightarrow x \mid 0 \cdot \Gamma}\ \textsc{Def}$$

From the premise, we have that:

$$\Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')$$

from which we can construct the following typing derivation, matching the above conclusion:

$$\frac{(x : \forall \overline{\alpha : \kappa}.A') \in D \qquad \Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')}{\Sigma; 0 \cdot \Gamma \vdash x : A}\ \textsc{Def}$$

c) Case $\to_{\text{R}}$
For synthesis of an abstraction term, we have the derivation:

$$\frac{\Sigma; \Gamma, x :_q A \vdash B \Rightarrow t \mid \Delta, x :_r A \qquad r \sqsubseteq q}{\Sigma; \Gamma \vdash A^q \to B \Rightarrow \lambda x.t \mid \Delta}\ \to_{\text{R}}$$

By induction on the premise, we have:

$$\Sigma; \Delta, x :_r A \vdash t : B \qquad \text{(ih)}$$

and that:

$$r \sqsubseteq q$$

from which we can construct the following typing derivation, matching the conclusion:

$$\frac{\dfrac{\Sigma; \Delta, x :_r A \vdash t : B \qquad r \sqsubseteq q}{\Sigma; \Delta, x :_q A \vdash t : B}\ \textsc{Approx}}{\Sigma; \Delta \vdash \lambda x.t : A^q \to B}\ \textsc{Abs}$$

d) Case $\to_{\mathrm{L}}$
For synthesising an application, we have the derivation:

$$\frac{\begin{array}{c}\Sigma; \Gamma, x_1 :_{r_1} A^q \to B, x_2 :_{r_1} B \vdash C \Rightarrow t_1 \mid \Delta_1, x_1 :_{s_1} A^q \to B, x_2 :_{s_2} B \\ \Sigma; \Gamma, x_1 :_{r_1} A^q \to B \vdash A \Rightarrow t_2 \mid \Delta_2, x_1 :_{s_3} A^q \to B \qquad \Sigma \vdash A^q \to B : \mathrm{Type}\end{array}}{\Sigma; \Gamma, x_1 :_{r_1} A^q \to B \vdash C \Rightarrow [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + s_2 \cdot q \cdot \Delta_2), x_1 :_{s_2+s_1+(s_2 \cdot q \cdot s_3)} A^q \to B} \to_{\mathrm{L}}$$

By induction on the premises, we obtain the following typing judgements:

$$\Sigma; \Delta_1, x_1 :_{s_1} A^q \to B, x_2 :_{s_2} B \vdash t_1 : C \qquad \text{(ih)}$$
$$\Sigma; \Delta_2, x_1 :_{s_3} A^q \to B \vdash t_2 : A \qquad \text{(ih)}$$

and from the premises, we have that:

$$\Sigma \vdash A^q \to B : \mathrm{Type}$$

from which we can construct the following derivation, making use of the admissibility of substitution:

$$\frac{\dfrac{\Sigma \vdash A^q \to B : \mathrm{Type}}{\Sigma; x_1 :_1 A^q \to B \vdash x_1 : A^q \to B}\,\textsc{Var} \qquad \Sigma; \Delta_2, x_1 :_{s_3} A^q \to B \vdash t_2 : A}{\Sigma; q \cdot \Delta_2, x_1 :_{1+(q \cdot s_3)} A^q \to B \vdash x_1\, t_2 : B}\,\textsc{App} \tag{B.32}$$

$$\frac{(\text{B.32}) \qquad \Sigma; \Delta_1, x_1 :_{s_1} A^q \to B, x_2 :_{s_2} B \vdash t_1 : C}{\Sigma; (\Delta_1 + s_2 \cdot q \cdot \Delta_2), x_1 :_{s_2+s_1+(s_2 \cdot q \cdot s_3)} A^q \to B \vdash [(x_1\, t_2)/x_2]t_1 : C}\,\textsc{Subst}$$

making use of the distributivity property of semirings, along with unitality of 1 and commutativity of $+$, such that $s_1 + s_2 \cdot (1 + (q \cdot s_3)) = s_1 + (s_2 \cdot 1) + (s_2 \cdot q \cdot s_3) = s_2 + s_1 + (s_2 \cdot q \cdot s_3)$.

e) Case $\mathrm{C_R}$
For synthesising a constructor introduction, we have the derivation:

$$\frac{\begin{array}{l}(C : \forall \overline{\alpha : \kappa}.B_1'^{\,q_1} \to \ldots \to B_n'^{\,q_n} \to K\,\vec{A}') \in D \\ \Sigma \vdash B_1{}^{q_1} \to \ldots \to B_n{}^{q_n} \to K\,\vec{A} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.B_1'^{\,q_1} \to \ldots \to B_n'^{\,q_n} \to K\,\vec{A}') \\ \Sigma; \Gamma \vdash B_i \Rightarrow t_i \mid \Delta_i\end{array}}{\Sigma; \Gamma \vdash K\,\vec{A} \Rightarrow C\,t_1 \ldots t_n \mid 0 \cdot \Gamma + (q_1 \cdot \Delta_1) + \ldots + (q_n \cdot \Delta_n)}\,\mathrm{C_R}$$

By induction on the premises, we obtain the following typing judgements:

$$\Sigma; \Delta_1 \vdash t_1 : B_1 \quad , \ldots, \quad \Sigma; \Delta_n \vdash t_n : B_n \qquad \text{(ih)}$$

from which we can construct the following derivation, matching the above conclusion:

$$\frac{(C : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\vec{A'}) \in D \qquad \Sigma \vdash {B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A} = \text{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\vec{A'})}{\Sigma; 0 \cdot \Gamma \vdash C : {B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A}}\ \textsc{Con} \tag{B.33}$$

$$\frac{\dfrac{\dfrac{(\text{B.33}) \qquad \Sigma; \Delta_1 \vdash t_1 : B_1}{\Sigma; 0 \cdot \Gamma + q_1 \cdot \Delta_1 \vdash C\,t_1 : {B_2}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A}}\ \textsc{App}}{\vdots}}{\Sigma; 0 \cdot \Gamma + q_1 \cdot \Delta_1 + ... + q_{n-1} \cdot \Delta_{n-1} \vdash C\,t_1 ... t_{n-1} : {B_n}^{q_n} \to K\,\vec{A}}\ \textsc{App} \tag{B.34}$$

$$\frac{(\text{B.34}) \qquad \Sigma; \Delta_n \vdash t_n : B_n}{\Sigma; 0 \cdot \Gamma + q_1 \cdot \Delta_1 + ... + q_n \cdot \Delta_n \vdash C\,t_1 ... t_n : K\,\vec{A}}\ \textsc{App}$$

f) Case C_L
For synthesising a case statement, we have the derivation:

$$\frac{\begin{array}{l}(C_i : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1^i} \to ... \to {B'_n}^{q_n^i} \to K\,\vec{A'}) \in D \qquad \Sigma \vdash K\,\vec{A} : \text{Type} \\ \Sigma \vdash {B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A} = \text{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\vec{A'}) \\ \Sigma; \Gamma, x :_r K\,\vec{A}, y_1^i :_{r \cdot q_1^i} B_1, ..., y_n^i :_{r \cdot q_1^i} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\,\vec{A}, y_1^i :_{s_1^i} B_1, ..., y_n^i :_{s_n^i} B_n \\ \exists {s'}_j^i.\, s_j^i \sqsubseteq {s'}_j^i \cdot q_j^i \sqsubseteq r \cdot q_j^i \qquad s_i = {s'}_1^i \sqcup ... \sqcup {s'}_n^i \qquad |K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup ... \sqcup s_m\end{array}}{\Sigma; \Gamma, x :_r K\,\vec{A} \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y_1^i ... y_n^i \mapsto t_i} \mid (\Delta_1 \sqcup ... \sqcup \Delta_m), x :_{(r_1 \sqcup ... \sqcup r_m) + (s_1 \sqcup ... \sqcup s_m)} K\,\vec{A}}\ \text{C}_\text{L}$$

By induction on the premises we obtain the following typing judgements:

$$\Sigma; \Delta_i, x :_{r_i} K\,\vec{A}, y_1^i :_{s_1^i} B_1, ..., y_n^i :_{s_n^i} B_n \vdash t_i : B \qquad \text{(ih)}$$

We have by the definition of $\sqcup$:

i. $\Delta_i \sqsubseteq (\Delta_1 \sqcup ... \sqcup \Delta_m)$
ii. $r_i \sqsubseteq r_1 \sqcup ... \sqcup r_m$

and from the premises of the synthesis rule:

iii. ${s'}_j^i \sqsubseteq s_1 \sqcup ... \sqcup s_m$
iv. $s_j^i \sqsubseteq {s'}_j^i \cdot q_j^i$
v. $|K\,A| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup ... \sqcup s_m$
vi. $\Sigma \vdash K\,\vec{A} : \text{Type}$

and from rule $\kappa_{\to}$ we have that

$$\Sigma \vdash K\,\vec{A} : \text{Type} \Rightarrow {B_1}^{q_1^i} \to \ldots \to {B_n}^{q_n^i} \to K\,\vec{A} : \text{Type}$$

thus we have that

vii. $\Sigma \vdash B_j : \text{Type}$

We then construct the following three derivations towards the goal:

$$\frac{(g)}{\Sigma; q_j^i \cdot s_1 \sqcup \ldots \sqcup s_m \vdash\ y_j^i : B_j \rhd y_j^i :_{q_j^i \cdot s_1 \sqcup \ldots \sqcup s_m} B_j}\ \textsc{PVar} \tag{B.35}$$

$$\frac{\begin{array}{c}(C : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\,\vec{A'}) \in D \\ \Sigma \vdash {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K\,\vec{A} = \text{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\,\vec{A'}) \\ \text{(B.35)} \qquad (e)\end{array}}{\Sigma; s_1 \sqcup \ldots \sqcup s_m \vdash\ C_i\ y_1^i \ldots y_n^i : K\,\vec{A} \rhd y_j^i :_{q_j^i \cdot s_1 \sqcup \ldots \sqcup s_m} B_j, \ldots, y_n^i :_{q_n^i \cdot s_1 \sqcup \ldots \sqcup s_m} B_n}\ \textsc{PCon} \tag{B.36}$$

and

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma; \Delta_i, x :_{r_i} K\,\vec{A}, y_1^i :_{s_1^i} B_1, \ldots, y_n^i :_{s_n^i} B_n \vdash t_i : B}\ \textsc{induction} \quad (d)}{\Sigma; \Delta_i, x :_{r_i} K\,\vec{A}, y_1^i :_{q_1^i \cdot s'^i_j} B_1, \ldots, y_n^i :_{q_n^i \cdot s'^i_j} B_n \vdash t_i : B}\ \textsc{Approx} \quad (c)}{\Sigma; \Delta_i, x :_{r_i} K\,\vec{A}, y_1^i :_{q_1^i \cdot s_1 \sqcup \ldots \sqcup s_m} B_1, \ldots, y_n^i :_{q_n^i \cdot s_1 \sqcup \ldots \sqcup s_m} B_n \vdash t_i : B}\ \textsc{Approx} \quad (b)}{\Sigma; \Delta_i, x :_{(r_1 \sqcup \ldots \sqcup r_m)} K\,\vec{A}, y_1^i :_{q_1^i \cdot s_1 \sqcup \ldots \sqcup s_m} B_1, \ldots, y_n^i :_{q_n^i \cdot s_1 \sqcup \ldots \sqcup s_m} B_n \vdash t_i : B}\ \textsc{Approx} \quad (a)}{\Sigma; (\Delta_1 \sqcup \ldots \sqcup \Delta_m), x :_{(r_1 \sqcup \ldots \sqcup r_m)} K\,\vec{A}, y_1^i :_{q_1^i \cdot s_1 \sqcup \ldots \sqcup s_m} B_1, \ldots, y_n^i :_{q_n^i \cdot s_1 \sqcup \ldots \sqcup s_m} B_n \vdash t_i : B}\ \textsc{Approx} \tag{B.37}$$

$$\frac{\dfrac{\dfrac{(f)}{\Sigma; x :_1 K\,\vec{A} \vdash x : K\,A}\ \textsc{Var} \qquad \text{(B.36)} \qquad \text{(B.37)}}{\Sigma; ((\Delta_1 \sqcup \ldots \sqcup \Delta_m), x :_{(r_1 \sqcup \ldots \sqcup r_m)} K\,\vec{A}) + x :_{(s_1 \sqcup \ldots \sqcup s_m \cdot 1)} K\,\vec{A} \vdash \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y_1^i \ldots y_n^i \mapsto t_i} : B}\ \textsc{Case}}{\Sigma; (\Delta_1 \sqcup \ldots \sqcup \Delta_m), x :_{(r_1 \sqcup \ldots \sqcup r_m) + (s_1 \sqcup \ldots \sqcup s_m)} K\,\vec{A} \vdash \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y_1^i \ldots y_n^i \mapsto t_i} : B}\ \equiv$$

g) Case $\Box_{\text{R}}$

For synthesising a promotion, we have the derivation:

$$\frac{\Sigma; \Gamma \vdash A \Rightarrow t \mid \Delta}{\Sigma; \Gamma \vdash \Box_r A \Rightarrow [t] \mid r \cdot \Delta}\ \Box_{\text{R}}$$

By induction on the premise we have:

$$\Sigma; \Delta \vdash t : A$$

From which we can construct the following derivation, matching the above conclusion:

$$\frac{\Sigma; \Delta \vdash t : A}{r \cdot \Delta \vdash [t] : \Box_r A}\ \textsc{Pr}$$

h) Case $\Box_{\mathrm{L}}$
For synthesising an unboxing, we have the derivation:

$$\frac{\begin{array}{l}\Sigma; \Gamma, y :_{r \cdot q} A, x :_r \Box_q A \vdash B \Rightarrow t \mid \Delta, y :_{s_1} A, x :_{s_2} \Box_q A \\ \exists s_3.\, s_1 \sqsubseteq s_3 \cdot q \sqsubseteq r \cdot q \qquad \Sigma \vdash \Box_q A : \mathrm{Type}\end{array}}{\Sigma; \Gamma, x :_r \Box_q A \vdash B \Rightarrow \mathbf{case}\ x\ \mathbf{of}\ [y] \to t \mid \Delta, x :_{s_3 + s_2} \Box_q A}\ \Box_{\mathrm{L}}$$

By induction on the premise we have:

$$\Sigma; \Delta, y :_{s_1} A, x :_{s_2} \Box_q A \vdash t : B \tag{ih}$$

and from the premises we have that:

viii. $s_1 \sqsubseteq s_3 \cdot q$

ix. $\Sigma \vdash \Box_q A : \mathrm{Type}$

and through the $\kappa_{\Box}$ rule, we have that:

$$\Sigma \vdash \Box_q A : \mathrm{Type} \Rightarrow \Sigma \vdash A : \mathrm{Type}$$

From this we can construct the following derivation, towards the goal:

$$\frac{(i)}{\Sigma; x :_1 \Box_q A \vdash x : \Box_q A}\ \textsc{Var} \tag{B.38}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \vdash A : \mathrm{Type}}{\Sigma; s_3 \cdot q \vdash y : A \rhd y :_{s_3 \cdot q} A}\ \textsc{PVar}}{\Sigma; s_3 \vdash [y] : \Box_q A \rhd y :_{s_3 \cdot q} A}\ \textsc{PBox} \qquad \dfrac{\Sigma; \Delta, y :_{s_1} A, x :_{s_2} \Box_q A \vdash t : B \qquad s_1 \sqsubseteq s_3 \cdot q}{\Sigma; \Delta, y :_{s_3 \cdot q} A, x :_{s_2} \Box_q A \vdash t : B}\ \textsc{Approx}}{\Sigma; \Delta, x :_{s_3 + s_2} \Box_q A \vdash \mathbf{case}\ x\ \mathbf{of}\ [y] \to t : B}\ \textsc{Case}$$

(B.38)

i) Case μR
For synthesising a recursive data type introduction form, we have the derivation:

$$\frac{D; \Sigma; \Gamma \vdash A[\mu X.A / X] \Rightarrow t \mid\ \Delta}{D; \Sigma; \Gamma \vdash \mu X.A \Rightarrow t \mid\ \Delta}\ \mu_{\mathrm{R}}$$

By induction on the premise we have:

$$D; \Sigma; \Delta \vdash t : A[\mu X.A / X] \tag{ih}$$

from which we can construct the derivation, matching the form of the lemma, leveraging the equirecursivity of our types:

$$\frac{D; \Sigma; \Delta \vdash t : A[\mu X.A/X]}{D; \Sigma; \Delta \vdash t : \mu X.A}\ \mu_1$$

j) Case μL
For synthesising a recursive data type elimination form, we have the derivation:

$$\frac{D; \Sigma; \Gamma, x :_r A[\mu X.A/X] \vdash B \Rightarrow t \mid \Delta}{D; \Sigma; \Gamma, x :_r \mu X.A \vdash B \Rightarrow t \mid \Delta}\ \mu_{\mathrm{L}}$$

By induction on the premise we have:

$$D; \Sigma; \Delta, x :_r A[\mu X.A/X] \vdash t : B \qquad \text{(ih)}$$

from which we can construct the derivation:

$$\frac{\dfrac{\Sigma \vdash \mu X.A : \mathrm{Type}}{D; \Sigma; x :_r \mu X.A \vdash x :_1 \mu X.A}\ \textsc{Var}}{D; \Sigma; x :_r \mu X.A \vdash x :_1 A[\mu X.A]}\ \mu_2$$

and by using lemma 4.1.1 on :

$$D; \Sigma; x : A[\mu X.A/X] \vdash t : B$$

we obtain the following, matching the above conclusion:

$$D; \Sigma; D, x : \mu X.A \vdash [x/x]t : B = D; \Sigma; D, x : \mu X.A \vdash t : B$$

2. a) Case TopLevel
For the top-level of synthesis, we have the derivation:

$$\frac{\overline{\alpha : \kappa}; \emptyset \vdash A \Rightarrow t \mid \emptyset}{\emptyset; \emptyset \vdash \forall \overline{\alpha : \kappa}.A \Rightarrow t \mid \emptyset}\ \textsc{TopLevel}$$

from induction on the premise, we have that:

$$\overline{\alpha : \kappa}; \emptyset \vdash t : A \qquad \text{(ih)}$$

from which we can construct the following typing derivation, matching the above conclusion:

$$\frac{\overline{\alpha : \kappa}; \emptyset \vdash t : A}{\emptyset; \emptyset \vdash t : \forall \overline{\alpha : \kappa}.A}\ \textsc{TopLevel}$$

□

B.3.2 *Soundness of Focusing for the Fully Graded Synthesis Calculus*

Lemma 4.4.1 (Soundness of focusing for graded-base synthesis)**.** For all contexts Γ, Ω and types A:

1. RIGHT ASYNC :	$D;\Sigma;\Gamma;\Omega \vdash A \Uparrow \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma,\Omega \vdash A \Rightarrow t \mid \Delta$
2. LEFT ASYNC :	$D;\Sigma;\Gamma;\Omega \Uparrow \vdash B \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma,\Omega \vdash B \Rightarrow t \mid \Delta$
3. RIGHT SYNC :	$D;\Sigma;\Gamma;\varnothing \vdash A \Downarrow \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma \vdash A \Rightarrow t \mid \Delta$
4. LEFT SYNC :	$D;\Sigma;\Gamma;x :_r A \Downarrow \vdash B \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma,x :_r A \vdash B \Rightarrow t \mid \Delta$
5. FOCUS RIGHT :	$D;\Sigma;\Gamma;\varnothing \Uparrow \vdash B \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma \vdash B \Rightarrow t \mid \Delta$
6. FOCUS LEFT :	$D;\Sigma;\Gamma,x :_r A;\varnothing \Uparrow \vdash B \Rightarrow t \mid \Delta$	$\iff$	$D;\Sigma;\Gamma,x :_r A \vdash B \Rightarrow t \mid \Delta$

i.e. t has type A under context Δ, which contains variables with grades reflecting their use in t.

Proof. 1. Case: 1. TopLevel:

a) Case TOPLEVEL
In the case of the right asynchronous rule for abstraction introduction, the synthesis rule has the form:

$$\frac{D;\overline{\alpha : \kappa};\varnothing;\varnothing \vdash A \Uparrow \Rightarrow t \mid \varnothing}{D;\varnothing;\varnothing;\varnothing \vdash \forall \overline{\alpha : \kappa}.A \Uparrow \Rightarrow t \mid \varnothing}\ \textsc{TopLevel}$$

By induction on the first premise, we have that:

$$\overline{\alpha : \kappa};\varnothing \vdash A \Rightarrow t \mid \varnothing \qquad \text{(ih)}$$

from case 1 of the lemma. From which, we can construct the following instantiation of the TOPLEVEL synthesis rule in the non-focusing calculus:

$$\frac{\overline{\alpha : \kappa};\varnothing \vdash A \Rightarrow t \mid \varnothing}{\varnothing;\varnothing \vdash \forall \overline{\alpha : \kappa}.A \Rightarrow t \mid \varnothing}\ \textsc{TopLevel}$$

2. Case: 2. Right Async:

a) Case $\to_{\mathrm{R}}$
In the case of the right asynchronous rule for abstraction introduction, the synthesis rule has the form:

$$\frac{D;\Sigma;\Gamma;\Omega, x :_q A \vdash B \Uparrow \Rightarrow t \mid \Delta, x :_r A \qquad r \sqsubseteq q}{D;\Sigma;\Gamma;\Omega \vdash A^q \to B \Uparrow \Rightarrow \lambda x.t \mid \Delta}\ \to_{\mathrm{R}}$$

By induction on the premise, we have that:

$$\Sigma;(\Gamma,\Omega), x :_q A \vdash B \Rightarrow t \mid \Delta, x :_r A \qquad \text{(ih)}$$

from case 2 of the lemma. From which, we can construct the following instantiation of the $\to_{\mathrm{R}}$ synthesis rule in the non-focusing calculus:

$$\frac{\Sigma; (\Gamma, \Omega), x :_q A \vdash B \Rightarrow t \mid \Delta, x :_r A \qquad r \sqsubseteq q}{\Sigma; \Gamma, \Omega \vdash A \to B \Rightarrow \lambda x.t \mid \Delta} \to_{\mathrm{R}}$$

b) Case $\Uparrow_{\mathrm{R}}$
In the case of the right asynchronous rule for transition to a left asynchronous judgement, the synthesis rule has the form:

$$\frac{D; \Sigma; \Gamma; \Omega \Uparrow \vdash B \Rightarrow t \mid \Delta \qquad B \text{ not right async}}{D; \Sigma; \Gamma; \Omega \vdash B \Uparrow \Rightarrow t \mid \Delta} \Uparrow_{\mathrm{R}}$$

By induction on the first premise, we have that:

$$\Sigma; \Gamma, \Omega \vdash B \Rightarrow t \mid \Delta \tag{ih}$$

from case 3 of the lemma.

3. Case 3. Left Async:

a) Case $\textsc{Con}_{\mathrm{L}}$
In the case of the left asynchronous rule for constructor elimination, the synthesis rule has the form:

$$\frac{\begin{array}{l}(C_i : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1^i} \to \ldots \to {B'_n}^{q_n^i} \to K\,\vec{A}') \in D \qquad \Sigma \vdash K\,\vec{A} : \mathrm{Type} \\ \Sigma \vdash {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K\,\vec{A} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\,\vec{A}') \\ D; \Sigma; \Gamma; \Omega, x :_r K\,\vec{A}, y_1^i :_{r \cdot q_1^i} B_1, \ldots, y_n^i :_{r \cdot q_1^i} B_n \Uparrow \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\,\vec{A}, y_1^i :_{s_1^i} B_1, \ldots, y_n^i :_{s_n^i} B_n \\ \exists {s'}^i_j.\, s_j^i \sqsubseteq {s'}^i_j \cdot q_j^i \sqsubseteq r \cdot q_j^i \qquad s_i = {s'}^i_1 \sqcup \ldots \sqcup {s'}^i_n \qquad |K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup \ldots \sqcup s_m\end{array}}{D; \Sigma; \Gamma; \Omega, x :_r K\,\vec{A} \Uparrow \vdash B \Rightarrow \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ y_1^i \ldots y_n^i \mapsto t_i} \mid (\Delta_1 \sqcup \ldots \sqcup \Delta_m), x :_{(r_1 \sqcup \ldots \sqcup r_m) + (s_1 \sqcup \ldots \sqcup s_m)} K\,\vec{A}}$$

By induction on the second premise, we have that:

$$\Sigma; (\Gamma, \Omega), x :_r K\,\vec{A}, y_1^i :_{r \cdot q_1^i} B_1, \ldots, y_n^i :_{r \cdot q_1^i} B_n \Uparrow \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\,\vec{A}, y_1^i :_{s_1^i} B_1, \ldots, y_n^i :_{s_n^i} B_n \tag{ih}$$

from case 3 of the lemma. From the second premise, we have that:

$$\Sigma \vdash K\,\vec{A} : \mathrm{Type}$$

From which we can construct the following instantiation of the Con_L rule in the non-focusing calculus:

$$\frac{\begin{array}{c}(C_i : \forall\overline{\alpha : \kappa}.{B'_1}^{q^i_1} \to ... \to {B'_n}^{q^i_n} \to K\,\vec{A'}) \in D \\ \Sigma \vdash K\,\vec{A} : \text{Type} \\ \Sigma \vdash {B_1}^{q_1} \to ... \to {B_n}^{q_n} \to K\,\vec{A} = \text{inst}(\forall\overline{\alpha : \kappa}.{B'_1}^{q_1} \to ... \to {B'_n}^{q_n} \to K\,\vec{A'}) \\ \Sigma; (\Gamma, \Omega), x :_r K\,\vec{A}, y^i_1 :_{r\,\cdot\,q^i_1} B_1, ..., y^i_n :_{r\,\cdot\,q^i_1} B_n \vdash B \Rightarrow t_i \mid \Delta_i, x :_{r_i} K\,\vec{A}, y^i_1 :_{s^i_1} B_1, ..., y^i_n :_{s^i_n} B_n \\ \exists {s'}^i_j.\, s^i_j \sqsubseteq {s'}^i_j \cdot q^i_j \sqsubseteq r \cdot q^i_j \\ s_i = {s'}^i_1 \sqcup ... \sqcup {s'}^i_n \qquad |K\,\vec{A}| > 1 \Rightarrow 1 \sqsubseteq s_1 \sqcup ... \sqcup s_m\end{array}}{\Sigma; (\Gamma, \Omega), x :_r K\,\vec{A} \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ y^i_1 ... y^i_n \mapsto t_i} \mid (\Delta_1 \sqcup ... \sqcup \Delta_m), x :_{(r_1 \sqcup ... \sqcup r_m) + (s_1 \sqcup ... \sqcup s_m)} K\,\vec{A}}\ \text{Con}_\text{L}$$

b) Case $\Box_\text{L}$
In the case of the left asynchronous rule for graded modality elimination, the synthesis rule has the form:

$$\frac{\begin{array}{c}D; \Sigma; \Gamma; \Omega, y :_{r\,\cdot\,q} A, x :_r \Box_q A \Uparrow \vdash B \Rightarrow t \mid \Delta, y :_{s_1} A, x :_{s_2} \Box_q A \\ \exists s_3.\, s_1 \sqsubseteq s_3 \cdot q \sqsubseteq r \cdot q \qquad \Sigma \vdash \Box_q A : \text{Type}\end{array}}{D; \Sigma; \Gamma; \Omega, x :_r \Box_q A \Uparrow \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ [y] \to t \mid \Delta, x :_{s_3 + s_2} \Box_q A}\ \Box_\text{L}$$

By induction on the first premise, we have that:

$$\Sigma; (\Gamma, \Omega), y :_{r\,\cdot\,q} A, x :_r \Box_q A \vdash B \Rightarrow t \mid \Delta, y :_{s_1} A, x :_{s_2} \Box_q A \qquad \text{(ih)}$$

from case 3 of the lemma. From the third premise, we have that:

$$\Sigma \vdash \Box_q A : \text{Type}$$

From which, we can construct the following instantiation of the $\Box_\text{L}$ synthesis rule in the non focusing calculus:

$$\frac{\begin{array}{c}\Sigma; (\Gamma, \Omega), y :_{r\,\cdot\,q} A, x :_r \Box_q A \vdash B \Rightarrow t \mid \Delta, y :_{s_1} A, x :_{s_2} \Box_q A \\ \exists s_3.\, s_1 \sqsubseteq s_3 \cdot q \sqsubseteq r \cdot q \qquad \Sigma \vdash \Box_q A : \text{Type}\end{array}}{\Sigma; (\Gamma, \Omega), x :_r \Box_q A \vdash B \Rightarrow \textbf{case}\ x\ \textbf{of}\ [y] \to t \mid \Delta, x :_{s_3 + s_2} \Box_q A}\ \Box_\text{L}$$

c) Case μ_L
In the case of the left asynchronous rule for recursive data type elimination, the synthesis rule has the form:

$$\frac{D; \Sigma; \Gamma; \Omega, x :_r A[\mu X.A/X] \Uparrow \vdash B \Rightarrow t \mid\ \Delta}{D; \Sigma; \Gamma; \Omega, x :_r \mu X.A \Uparrow \vdash B \Rightarrow t \mid\ \Delta}\ \mu_\text{L}$$

By induction on the first premise, we have that:

$$\Sigma; (\Gamma, \Omega), x :_r A[\mu X.A/X] \vdash B \Rightarrow t \mid\ \Delta \qquad \text{(ih)}$$

from case 3 of the lemma. From which, we can construct the following instantiation of the μ_{L} synthesis rule in the non focusing calculus:

$$\frac{\Sigma; (\Gamma, \Omega), x :_r A[\mu X.A/X] \vdash B \Rightarrow t \mid \ \Delta}{\Sigma; (\Gamma, \Omega), x :_r \mu X.A \vdash B \Rightarrow t \mid \ \Delta}\ \mu_{\mathrm{L}}$$

d) Case $\Uparrow_{\mathrm{L}}$
In the case of the left asynchronous rule for transitioning an assumption from the focusing context Ω to the non-focusing context Γ, the synthesis rule has the form:

$$\frac{D; \Sigma; \Gamma, x :_r A; \Omega \Uparrow \vdash B \Rightarrow t \mid \Delta \qquad \text{A not left async}}{D; \Sigma; \Gamma; \Omega, x :_r A \Uparrow \vdash B \Rightarrow t \mid \Delta}\ \Uparrow_{\mathrm{L}}$$

By induction on the first premise, we have that:

$$\Sigma; \Gamma, x : A, \Omega \vdash C \Rightarrow t \mid \Delta \tag{ih}$$

from case 3 of the lemma.

4. Case 4. Right Sync:

a) Case C_{R}
In the case of the right synchronous rule for constructor introduction, the synthesis rule has the form:

$$\frac{\begin{array}{c}(C : \forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\,\vec{A'}) \in D \\ \Sigma \vdash {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K\,\vec{A} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\,\vec{A'}) \\ \Sigma; \Gamma; \emptyset \vdash B_i \Downarrow \Rightarrow t_i \mid \ \Delta_i\end{array}}{\Sigma; \Gamma; \emptyset \vdash K\,\vec{A} \Downarrow \Rightarrow C\,t_1 \ldots t_n \mid \ \Delta_1 + \ldots + \Delta_n}\ \mathrm{C}_{\mathrm{R}}$$

By induction on the second premise, we have that:

$$\Sigma; \Gamma \vdash B_i \Rightarrow t_i \mid \Delta_i \tag{ih}$$

from case 4 of the lemma. From which, we can construct the following instantiation of the C_{R} synthesis rule in the non-focusing calculus:

$$\frac{\begin{array}{c}(C : \forall \overline{\alpha : \kappa}.{B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K\,\vec{A}) \in D \\ \Sigma \vdash {B_1}^{q_1} \to \ldots \to {B_n}^{q_n} \to K\,\vec{A} = \mathrm{inst}(\forall \overline{\alpha : \kappa}.{B'_1}^{q_1} \to \ldots \to {B'_n}^{q_n} \to K\,\vec{A'}) \\ \Sigma; \Gamma \vdash B_i \Rightarrow t_i \mid \Delta_i\end{array}}{\Sigma; \Gamma \vdash K\,\vec{A} \Rightarrow C\,t_1 \ldots t_n \mid 0 \cdot \Gamma + (q_1 \cdot \Delta_1) + \ldots + (q_n \cdot \Delta_n)}\ \mathrm{Con}_{\mathrm{R}}$$

b) Case $\Box_{\mathrm{R}}$

In the case of the right synchronous rule for graded modality introduction, the synthesis rule has the form:

$$\frac{\Sigma; \Gamma; \emptyset \vdash A \Downarrow \Rightarrow t \mid \ \Delta}{\Sigma; \Gamma; \emptyset \vdash \Box_r A \Downarrow \Rightarrow [t] \mid \ r \cdot \Delta}\ \Box_{\mathrm{R}}$$

By induction on the premises, we have that:

$$\Sigma; \Gamma \vdash A \Rightarrow t \mid \Delta \tag{ih}$$

from case 4 of the lemma. From which, we can construct the following instantiation of the μR synthesis rule in the non-focusing calculus:

$$\frac{\Sigma; \Gamma \vdash A \Rightarrow t \mid \Delta}{\Sigma; \Gamma \vdash \Box_r A \Rightarrow [t] \mid r \cdot \Delta} \ \Box_{\mathrm{R}}$$

c) Case μ_{R}
In the case of the right synchronous rule for recursive data type introduction, the synthesis rule has the form:

$$\frac{D; \Sigma; \Gamma; \emptyset \vdash A[\mu X.A/X] \Downarrow \Rightarrow t \mid \ \Delta}{D; \Sigma; \Gamma; \emptyset \vdash \mu X.A \Downarrow \Rightarrow t \mid \ \Delta} \ \mu_{\mathrm{R}}$$

By induction on the premises, we have that:

$$\Sigma; \Gamma \vdash A[\mu X.A/X] \Rightarrow t \mid \ \Delta \qquad \text{(ih)}$$

from case 4 of the lemma. From which, we can construct the following instantiation of the μ_{R} synthesis rule in the non-focusing calculus:

$$\frac{D; \Sigma; \Gamma \vdash A[\mu X.A/X] \Rightarrow t \mid \ \Delta}{D; \Sigma; \Gamma \vdash \mu X.A \Rightarrow t \mid \ \Delta} \ \mu_{\mathrm{R}}$$

d) Case $\Downarrow_{\mathrm{R}}$
In the case of the right synchronous rule for transitioning from the right focusing phase to an asynchronous right phase, the synthesis rule has the form:

$$\frac{D; \Sigma; \Gamma; \emptyset \vdash A \Uparrow \Rightarrow t \mid \Delta}{D; \Sigma; \Gamma; \emptyset \vdash A \Downarrow t \mid \ \Delta} \ \Downarrow_{\mathrm{R}}$$

By induction on the first premise, we have that:

$$\Sigma; \Gamma \vdash A \Rightarrow t \mid \Delta \qquad \text{(ih)}$$

from case 4 of the lemma.

5. Case 5. Left Sync:

a) Case $\rightarrow_{\mathrm{L}}$
In the case of the left synchronous rule for application, the synthesis rule has the form:

$$\frac{\begin{array}{c}\Sigma; \Gamma, x_1 :_{r_1} A^q \to B; x_2 :_{r_1} B \Downarrow\vdash C \Rightarrow t_1 \mid \Delta_1, x_1 :_{s_1} A^q \to B, x_2 :_{s_2} B \\ \Sigma; \Gamma, x_1 :_{r_1} A^q \to B; \emptyset \vdash A \Downarrow \Rightarrow t_2 \mid \ \Delta_2, x_1 :_{s_3} A^q \to B \\ \Sigma \vdash A^q \to B : \mathrm{Type}\end{array}}{\Sigma; \Gamma; x_1 :_{r_1} A^q \to B \Downarrow\vdash C \Rightarrow [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + s_2 \cdot q \cdot \Delta_2), x_1 :_{s_2 + s_1 + (s_2 \cdot q \cdot s_3)} A^q \to B}$$

By induction on the first premise, we have that:

$$\Sigma; \Gamma, x_1 :_{r_1} A^q \to B, x_2 :_{r_1} B \vdash C \Rightarrow t_1 \mid \Delta_1, x_1 :_{s_1} A^q \to B, x_2 :_{s_2} B$$

from case 5 of the lemma. By induction on the second premise, we have that:

$$\Sigma; \Gamma, x_1 :_{r_1} A^q \to B \vdash A \Rightarrow t_2 \mid \ \Delta_2, x_1 :_{s_3} A^q \to B \qquad \text{(ih)}$$

from case 4 of the lemma. From the third premise, we have that:

$$\Sigma \vdash A^q \to B : \text{Type}$$

From which, we can construct the following instantiation of the $\to_{\text{L}}$ synthesis rule in the non-focusing calculus:

$$\frac{\begin{array}{c}\Sigma; \Gamma, x_1 :_{r_1} A^q \to B, x_2 :_{r_1} B \vdash C \Rightarrow t_1 \mid \Delta_1, x_1 :_{s_1} A^q \to B, x_2 :_{s_2} B \\ \Sigma; \Gamma, x_1 :_{r_1} A^q \to B \vdash A \Rightarrow t_2 \mid \Delta_2, x_1 :_{s_3} A^q \to B \\ \Sigma \vdash A^q \to B : \text{Type}\end{array}}{\Sigma; \Gamma, x_1 :_{r_1} A^q \to B \vdash C \Rightarrow [(x_1\, t_2)/x_2]t_1 \mid (\Delta_1 + s_2 \cdot q \cdot \Delta_2), x_1 :_{s_2 + s_1 + (s_2 \cdot q \cdot s_3)} A^q \to B} \to_{\text{L}}$$

b) Case VAR
In the case of the left synchronous rule for variable synthesis, the synthesis rule has the form:

$$\frac{\Sigma \vdash A : \text{Type}}{D; \Sigma; \Gamma; x :_r A \Downarrow \vdash A \Rightarrow x \mid 0 \cdot \Gamma, x :_1 A} \text{ VAR}$$

From the premise, we have that:

$$\Sigma \vdash A : \text{Type}$$

from which, we can construct the following instantiation of the VAR synthesis rule in the non-focusing calculus:

$$\frac{\Sigma \vdash A : \text{Type}}{\Sigma; \Gamma, x :_r A \vdash A \Rightarrow x \mid 0 \cdot \Gamma, x :_1 A} \text{ VAR}$$

c) Case DEF
In the case of the left synchronous rule for synthesis of a top-level definition usage, the synthesis rule has the form:

$$\frac{\Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')}{D, x : \forall \overline{\alpha : \kappa}.A'; \Sigma; \Gamma; \emptyset \Downarrow \vdash A \Rightarrow x \mid 0 \cdot \Gamma} \text{ DEF}$$

From the premise, we have that:

$$\Sigma \vdash A : \text{Type}$$

from which, we can construct the following instantiation of the VAR synthesis rule in the non-focusing calculus:

$$\frac{(x : \forall \overline{\alpha : \kappa}.A') \in D \qquad \Sigma \vdash A = \text{inst}(\forall \overline{\alpha : \kappa}.A')}{\Sigma; \Gamma \vdash A \Rightarrow x \mid 0 \cdot \Gamma} \text{ DEF}$$

d) Case $\Downarrow_{\mathrm{L}}$
In the case of the left synchronous rule for transitioning from the right focusing phase to an asynchronous left phase, the synthesis rule has the form:

$$\frac{D;\Sigma;\Gamma;x :_r A \Uparrow \vdash B \Rightarrow t \mid \Delta \quad \text{A not atomic and not left sync}}{D;\Sigma;\Gamma;x :_r A \Downarrow \vdash B \Rightarrow t \mid \Delta}\ \Downarrow_{\mathrm{L}}$$

By induction on the first premise, we have that:

$$\Sigma;\Gamma, x :_r A \vdash B \Rightarrow t \mid \Delta \tag{ih}$$

from case 5 of the lemma.

6. Case 6. Right Focus:
In the case of the focusing rule for transitioning from a left asynchronous judgement to a right synchronous judgement, the synthesis rule has the form:

$$\frac{D;\Sigma;\Gamma;\varnothing \vdash B \Downarrow \Rightarrow t \mid \ \Delta \qquad \text{B not atomic}}{D;\Sigma;\Gamma;\varnothing \Uparrow \vdash B \Rightarrow t \mid \Delta}\ \mathrm{Foc}_{\mathrm{R}}$$

By induction on the first premise, we have that:

$$\Sigma;\Gamma \vdash C \Rightarrow t \mid \Delta \tag{ih}$$

from case 3 of the lemma.

7. Case 7. Left Focus:
In the case of the focusing rule for transitioning from a left asynchronous judgement to a left synchronous judgement, the synthesis rule has the form:

$$\frac{D;\Sigma;\Gamma;x :_r A \Downarrow \vdash B \Rightarrow t \mid \Delta}{D;\Sigma;\Gamma, x :_r A;\varnothing \Uparrow \vdash B \Rightarrow t \mid \Delta}\ \mathrm{Foc}_{\mathrm{L}}$$

By induction on the first premise, we have that:

$$\Sigma;\Gamma, x :_r A \vdash C \Rightarrow t \mid \Delta \tag{ih}$$

from case 3 of the lemma.

□

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 November 5, 2024 (classicthesis v4.6).