Functional Ownership through Fractional Uniqueness

DANIEL MARSHALL, University of Kent, United Kingdom
DOMINIC ORCHARD, University of Kent, United Kingdom and University of Cambridge, United Kingdom

Ownership and borrowing systems, designed to enforce safe memory management without the need for garbage collection, have been brought to the fore by the Rust programming language. Rust also aims to bring some guarantees offered by functional programming into the realm of performant systems code, but the type system is largely separate from the ownership model, with type and borrow checking happening in separate compilation phases. Recent models such as RustBelt and Oxide aim to formalise Rust in depth, but there is less focus on integrating the basic ideas into more traditional type systems. An approach designed to expose an essential core for ownership and borrowing would open the door for functional languages to borrow concepts found in Rust and other ownership frameworks, so that more programmers can enjoy their benefits.

One strategy for managing memory in a functional setting is through *uniqueness types*, but these offer a coarse-grained view: either a value has exactly one reference, and can be mutated safely, or it cannot, since other references may exist. Recent work demonstrates that *linear* and *uniqueness* types can be combined in a single system to offer restrictions on program behaviour and guarantees about memory usage. We develop this connection further, showing that just as *graded* type systems like those of Granule and Idris generalise linearity, a Rust-like *ownership* model arises as a *graded generalisation of uniqueness*. We combine fractional permissions with grading to give the first account of ownership and borrowing that smoothly integrates into a standard type system alongside linearity and graded types, and extend Granule accordingly with these ideas.

CCS Concepts: • **Theory of computation** → **Type theory**; Linear logic; • **Software and its engineering** → *Memory management*; Formal language definitions; Functional languages.

Additional Key Words and Phrases: graded modal types, ownership, borrowing, fractional permissions

ACM Reference Format:
Daniel Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 131 (April 2024), 31 pages. https://doi.org/10.1145/3649848

1 INTRODUCTION

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

[1] https://www.zdnet.com/article/rust-takes-a-major-step-forward-as-linuxs-second-official-language/
[2] https://foundation.rust-lang.org/news/2021-02-08-hello-world/

Authors' addresses: Daniel Marshall, University of Kent, Canterbury, United Kingdom, dm635@kent.ac.uk; Dominic Orchard, University of Kent, Canterbury, United Kingdom and University of Cambridge, Cambridge, United Kingdom, D.A.Orchard@kent.ac.uk.

ACM 2475-1421/2024/4-ART131
https://doi.org/10.1145/3649848

The intricate ownership system which characterises Rust's approach to memory management is inspired in part by the literature on using *linear types* for tracking resource usage [Wadler 1990], based on *linear logic* [Girard 1987]. Linear types have also been brought into the realm of practical programming in recent years, for example in Haskell via the GHC compiler's linear types extension [Bernardy et al. 2017]. The precise theoretical relationship between linear types and the properties that can be enforced by Rust's borrow checker remains unclear, however, particularly since Rust lies outside the traditional functional paradigm to which linearity is most suited, opting for a more mixed imperative and functional approach.

Linear types require that every value is used exactly once, with the ! modality classifying non-linearity (arbitrary use). Modern resourceful type systems go beyond this coarse restriction of linearity, enabling usage to be classified more finely. Quantitative types, inspired by bounded linear logic [Girard et al. 1992], capture an upper bound on the amount of (re)use of a value by indexing the ! modality with natural numbers or polynomial terms. This quantitative analysis is generalised further by *graded modal types*, as exemplified by the Granule programming language [Orchard et al. 2019], which allows various properties of data use and data flow to be tracked smoothly in a single type system via an indexed modality $!_r$ with some algebraic structure on the indices r [Gaboardi et al. 2016; Orchard et al. 2019]. In this work, we seek a shared understanding between the precise resource reasoning of graded type systems and concepts of uniqueness, borrowing, and ownership as typified by Rust but also appearing in various other languages such as C++.

There already exist many models of ownership given from a type-theoretic perspective. For example, λ_{Rust} developed as part of the RustBelt framework provides a low-level model of Rust's ownership and borrowing systems suitable for formally verifying properties of Rust programs [Jung et al. 2017]. Oxide [Weiss et al. 2021] and FR [Pearce 2021] provide more high-level abstract models. Our focus is *not* on replicating these efforts; we do not aim to directly model all the details of Rust's particular approach to ownership. Rather, we offer a different and complementary perspective: a single system that relates general ownership and borrowing ideas to linear and graded types. This integration enables type, resource, and memory safety to be enforced via a single type system.

A natural place to start is *uniqueness types*, which provide the concept that a uniquely typed value has exactly one reference, and therefore is safe to mutate [Barendsen and Smetsers 1996; de Vries et al. 2008; Smetsers et al. 1994]. This gives a simplistic model of *ownership* without borrowing—a resource with a unique owner may be modified by that owner, but a broken guarantee of uniqueness can never be recovered. Recent work by Marshall et al. [2022] develops a type system that captures the relationship between linear and unique types: linear values are restricted from being copied or discarded in the future, whilst unique values are guaranteed to have never been copied in the past. We build upon this work as our foundation here but leverage the generalisations of grading. In the ownership framework of Mycroft and Voigt [2013], while uniqueness allows for memory-safe *temporal* aliasing, borrowing extends uniqueness by introducing the potential for safe patterns of *spatial* aliasing, dramatically increasing flexibility for the programmer. Just as the non-linearity modality ! can be generalised to a graded modality $!_r$ for fine-grained reasoning about resource usage via r and its algebraic structure, we show that uniqueness types, represented modally, can be 'graded' to capture this idea that borrowing is a controlled relaxation of the uniqueness guarantee.

Our approach allows many ideas from Rust, such as immutable and mutable borrows, partial borrows and reborrowing to be explained explicitly in a functional setting, all through the application of a form of grading based on the elegant fractional permissions of Boyland [2003]. Our work thus integrates and relates, within a unified framework, the substructural type systems of linearity, uniqueness, grading, ownership, and borrowing. This offers a pathway to expressing memory safety properties naturally in functional languages such as Haskell or Idris.

The outline of the paper is as follows. First, in Section 2 we recap the key concepts of uniqueness and borrowing using Rust as a convenient exemplar. We discuss how various programs making use of these ideas will later be rewritten as Granule programs through our unified type system, in order to motivate the rest of our work. We then recap Granule's pre-existing core calculus in Section 3, before moving on to the primary contributions of this paper:

- In Section 4, we connect Granule's uniqueness modality and its graded modal types with ideas from Rust, discussing how unique ownership allows for safe mutation. We generalise the connection between uniqueness and linearity, demonstrating that uniqueness and precise grading can coexist within a single type system. We leverage existential types over 'identifiers' for situations where multiple references pointing to the same value need tracking.
- In Section 5, we extend this idea to allow for multiple immutably borrowed references at a time, by annotating reference types with additional information, similarly to Boyland's fractional permissions. We show that this allows for a more fine-grained approach to tracking uniqueness in much the same way that grading increases expressivity over pure linearity.
  - In Section 5.1, we develop an equational theory for our extension to Granule's type system, showing that the modality for unique ownership induces a *relative functor* over the new modality for mutable and immutable borrowing.
  - In Section 5.2, we discuss how using distributive laws can allow for borrowing only part of a larger data structure while leaving the rest uniquely owned, enabling a much wider variety of practical programming patterns.
- In Section 6, we detail the semantics for the calculus developed thus far, and prove various key properties—both standard notions like progress and preservation but also borrow safety and uniqueness. The semantics is call-by-value, representing a practical system, departing from most previous operational models of grading in the literature.

Sections 7 and 8 discuss the many and varied areas of similar research that have inspired this paper, and look at some possible avenues for future work. All code examples along with the implementation of our type system atop Granule[3] are included in the artefact for this paper [Marshall and Orchard 2024b], and full definitions along with proofs of all theorems can be found in the appendix [Marshall and Orchard 2024a].

2 KEY CONCEPTS IN OWNERSHIP AND BORROWING

We first give an overview of the six key patterns of ownership and borrowing which we capture in our system in the rest of this paper, using simple Rust examples to demonstrate the patterns in action. The examples rely on a single base value of type `Colour(u32, u32, u32)` - a struct containing three unsigned integers, representing a colour with red, green, and blue components. We mark negative examples (those that do not type check) with 7 and positive examples with X.

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

```
let scarlet = Colour(220, 20, 60);
let x = scarlet;
let y = scarlet; // error: use of moved value: ‘scarlet‘
```
Rust 7

[3]The latest release of Granule is available from https://github.com/granule-project/granule/releases.

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

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

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

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

```
let mut viridian = Colour(52, 161, 128);
let x = &mut viridian;
let y = &mut viridian; // error: cannot borrow 'viridian' as mutable more than once
// ... code using 'x' and 'y'
```
Rust 7

Note, however, that the first three lines copied verbatim into a Rust file will in fact compile successfully unless additional code is introduced that makes use of the variables `x` and `y`. This is due to a feature called *non-lexical lifetimes*, through which the Rust compiler is able to *infer* that allowing two mutable borrows is safe as long as one of them is never used. We do not linger on this, since we will not capture this behaviour in our type system. Our goal is to embed some essential ownership and borrowing patterns *explicitly*. However, we will mention the possibility for extending our system with more advanced ideas in Section 8.

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

```
let mut cerulean = Colour(3, 109, 230);
let x = &mut cerulean;
let y = &cerulean;     // error: cannot borrow 'cerulean' as immutable
                       //       because it is also borrowed as mutable
```
Rust 7

In Section 5, we demonstrate that both mutable and immutable borrows can be represented through a graded generalisation of uniqueness typing to represent a borrower's level of access.

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

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

In Section 5.2, we take advantage of Granule's functional nature to present a cohesive way of managing this kind of partial borrow at the type level, including the possibility for working with disjoint components of the original structure concurrently.

Reborrowing. The final pattern we aim to capture in our system, also essential for practical programming, is the notion of *reborrowing*, through which it is possible to create a borrow of an identifier that itself references another value. We will show in Section 5 that this behaviour falls out of our generalisation of uniqueness naturally, without a need for additional constructs. To illustrate the general idea, consider the following example; here, the red component of the `amethyst` value is borrowed mutably as `r`, then another immutable borrow `x` is created pointing to `r`. The value can now only be read through `x`—both `r` and `amethyst` are inaccessible until the borrow is complete.

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

3 CORE CALCULUS

We recap Granule's core calculus which incorporates graded modal types into a linear type theory [Orchard et al. 2019]; we later extend this calculus with our functional model of uniqueness and borrowing. This section covers the syntax and static semantics (type system), whilst Section 6 gives an operational semantics incorporating the various extensions discussed throughout this work.

The calculus extends the linear λ-calculus with multiplicative products and unit, and a *semiring-graded necessity modality* $\square_r A$ where r is an element of a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$ which includes a requirement that $*$ and $+$ must be monotonic with respect to the ordering $\sqsubseteq$. This calculus gives a simplified monomorphic subset of Granule [Orchard et al. 2019], and closely resembles other graded systems from the literature [Abel and Bernardy 2020; Atkey 2018; Bianchini et al. 2023b; Brunel et al. 2014; Choudhury et al. 2021; Gaboardi et al. 2016; McBride 2016; Moon et al. 2021; Petricek et al. 2014]. Our extensions for ownership and borrowing could be made compatible with such systems, as the differences between these calculi do not interfere with the key ideas here.

Beyond existing work, we add existential types and type variables (restricted to a particular kind) to track identifiers associated with resources (whose dataflow paths may fork and join, due to the borrowing patterns we discuss later). These identifiers explicitly name and identify a resource at the type level such that references to different resources cannot be interchanged or joined as they will have different types, distinguished by the identifier. This will be illustrated in depth in Section 5 when we discuss particular examples of resources that can be managed by ownership.

3.1 Syntax

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

$$\begin{aligned} \mathsf{t} ::= {} & \mathsf{x} \mid \lambda \mathsf{x}.\mathsf{t} \mid \mathsf{t}_1\,\mathsf{t}_2 \mid (\mathsf{t}_1, \mathsf{t}_2) \mid \textbf{let}\,(x, y) = \mathsf{t}_1\,\textbf{in}\,\mathsf{t}_2 \mid () \mid \textbf{let}\,() = \mathsf{t}_1\,\textbf{in}\,\mathsf{t}_2 \\ & \mid [\mathsf{t}] \mid \textbf{let}\,[\mathsf{x}] = \mathsf{t}_1\,\textbf{in}\,\mathsf{t}_2 \\ & \mid \textbf{pack}\,\langle \mathsf{id}, \mathsf{t} \rangle \mid \textbf{unpack}\,\langle \mathsf{id}, \mathsf{x} \rangle = \mathsf{t}_1\,\textbf{in}\,\mathsf{t}_2 \end{aligned} \qquad \text{(terms)}$$

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

3.2 Type System

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

$$A, B ::= \mathsf{A} \multimap \mathsf{B} \mid \mathsf{A} \otimes \mathsf{B} \mid \mathsf{unit} \mid \square_r \mathsf{A} \mid \exists \mathsf{id}.\mathsf{A} \qquad \text{(types)}$$

Hence, our type syntax comprises linear function types $\mathsf{A} \multimap \mathsf{B}$, linear multiplicative products $\mathsf{A} \otimes \mathsf{B}$, a linear multiplicative unit (unit), the graded modality $\square_r \mathsf{A}$ where $r \in \mathcal{R}$, and existentially quantified types where $\mathsf{id} : \mathsf{Name}$ for an abstract kind of names Name.

Contexts Γ contain both linear assumptions $x : A$, graded assumptions $\mathsf{x} : [\mathsf{A}]_r$ which have originated from inside a graded modality, and type variables which we write as id due to their restricted purpose here, omitting the kind which is the abstract type Name:

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

Typing of the λ-calculus fragment is then by the following rules:

$$\frac{}{0 \cdot \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{x} : \mathsf{A}}\ \textsc{var} \qquad \frac{\Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{t} : \mathsf{B}}{\Gamma \vdash \lambda \mathsf{x}.\mathsf{t} : \mathsf{A} \multimap \mathsf{B}}\ \textsc{abs} \qquad \frac{\Gamma_1 \vdash \mathsf{t}_1 : \mathsf{A} \multimap \mathsf{B} \qquad \Gamma_2 \vdash \mathsf{t}_2 : \mathsf{A}}{\Gamma_1 + \Gamma_2 \vdash \mathsf{t}_1\,\mathsf{t}_2 : \mathsf{B}}\ \textsc{app}$$

The VAR, ABS, and APP rules are the standard rules of the linear λ-calculus, augmented with a notion of *contraction* captured by the + operation on contexts coming from multiple sub-terms. This operation is defined only when contexts are disjoint with respect to linear assumptions, and on overlapping graded assumptions we add their grades, e.g. $(\Gamma_1, \mathsf{x} : [\mathsf{A}]_r) + (\Gamma_2, \mathsf{x} : [\mathsf{A}]_s) = (\Gamma_1 + \Gamma_2), \mathsf{x} : [\mathsf{A}]_{r+s}$. More explicitly, context addition is declaratively specified as follows:

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

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

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

scaling graded assumptions by r, preserving ids, but undefined if Γ contains linear assumptions.

The rules involving graded modalities are then:

$$\frac{\Gamma \vdash t : A \qquad \neg\mathsf{resourceAllocator}(t)}{r \cdot \Gamma \vdash [t] : \Box_r A}\ \textsc{pr} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \textsc{der}$$

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2 : B}\ \textsc{elim} \qquad \frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ \textsc{approx}$$

The PR rule (*promotion*) introduces a graded modality with grade r, implying that the result of t can be used in an 'r-like' way and thus all of the dependencies of t must be scaled by r to propagate usage to the dependencies, none of which are allowed to be linear. Promotion also contains an explicit restriction that the value to be promoted must not be a *resource allocator*, designed to avoid problems that otherwise arise when promoting values with certain behaviours in a call-by-value setting; we will discuss this in detail when we describe our interface for mutable arrays in Section 5.

The ELIM rule eliminates a graded modality, capturing the idea that a requirement for x to be used in an 'r-like' way in t_2 can be matched with the capability of t_1 described by its graded modal type. In Granule, this construct is folded into pattern matching: we can 'unbox' (eliminate) a graded modality to provide a graded variable to the body of the function (the analogue to t_2 in this rule).

The DER rule (*dereliction*) connects linear typing to graded typing, stating that a requirement for a linear assumption is satisfied by an assumption graded by 1. The APPROX rule converts a grade r to s if s *approximates* r according to the semiring's pre-order $\sqsubseteq$.

One possible choice of semiring is over natural numbers $(\mathbb{N}, *, +, 0, 1, \equiv)$ with *discrete ordering* $\equiv$ such that there is no approximation and we track the exact number of times a term has been used. We could instead use the standard $\leq$ ordering on numbers which would permit approximation, allowing for an *upper bound* on a term's usage. Another useful semiring is *intervals* of natural numbers where a grade `0..1` represents values which can be used *either* zero or one times, capturing the notion of an *affine* value. Other interesting semirings include lattices for security levels [Abel and Bernardy 2020; Gaboardi et al. 2016], hardware schedules [Ghica and Smith 2014], sets for abstract property tracking, and $\{0, 1, \omega\}$ for capturing linear logic with $!A = \Box_\omega A$ [McBride 2016].

Finally, we have the rules for introducing and eliminating tensor products and the multiplicative unit, which are standard, though it is important to remember that products are *linear* and so it is not possible to freely discard either side: both elements must be used when consuming a product.

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

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

Example 3.1. The following gives an example derivation assuming the natural number semiring:

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

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

As a first taste of Granule syntax (which resembles Haskell, apart from the presence of the graded modal type constructs), the following captures the same idea as this example:

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

The type $\Box_r A$ is instead written postfix as `A [r]`, and Granule uses → for its linear function types.

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

$$\frac{\Gamma \vdash t : A \qquad id \notin dom(\Gamma)}{\Gamma \vdash \textbf{pack}\ \langle id', t \rangle : \exists id.A[id/id']}\ \text{PACK} \qquad \frac{\Gamma_1 \vdash t_1 : \exists id.A \qquad \Gamma_2, id, x : A \vdash t_2 : B \qquad id \notin fv(B)}{\Gamma_1 + \Gamma_2 \vdash \textbf{unpack}\ \langle id, x \rangle = t_1\ \textbf{in}\ t_2 : B}\ \text{UNPACK}$$

Note on Linear Haskell. As mentioned at the start of this section, we aim for the extensions we develop throughout this work to be compatible with other graded type systems after some adaptation. One particular system of interest is the calculus underlying the recent extension which introduces linear types to Haskell [Bernardy et al. 2017]. This extension uses a graded type system below the surface in order to implement linearity; all function types are given a 'multiplicity' annotation r written `(a %r -> b)` akin to a type $\Box_r A \multimap B$ here, where the annotation can be either `'One` or `'Many` representing either linear or unrestricted usage respectively. Work on formalising the connection between systems like this where all function types come with a grade (sometimes called 'graded base') and systems such as Granule's where values are linear by default and graded values are wrapped inside a modality (called 'linear base') is ongoing [Vollmer et al. 2024].

4 ONE OF A KIND: UNIQUENESS AND SHARING

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

Uniqueness types were introduced into Granule's core type system in previous work [Marshall et al. 2022], but we will extend this in two ways: first, we generalise the rules of Marshall et al. [2022] to allow for a graded necessity modality parameterised by an arbitrary semiring rather than the simple non-linearity ! modality, and second, we incorporate the identifiers described in Section 3 into the typing, which will become important when multiple references need to be tracked.

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

$$t ::= \ldots \mid \textbf{share}\ t \mid \textbf{clone}\ t_1\ \textbf{as}\ x\ \textbf{in}\ t_2 \qquad A, B ::= \ldots \mid *A \qquad \text{(terms \& types, extended)}$$

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

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \textbf{share}\ t : \Box_r A}\ \text{SHARE} \qquad \frac{1 \sqsubseteq r \qquad \text{cloneable}(A) \qquad \Gamma_1, \overline{id} \vdash t_1 : \Box_r A \qquad \Gamma_2, x : \exists \overline{id'}.*(A[\overline{id'}/\overline{id}]) \vdash t_2 : B}{(\Gamma_1 + \Gamma_2), \overline{id} \vdash \textbf{clone}\ t_1\ \textbf{as}\ x\ \textbf{in}\ t_2 : B}\ \text{CLONE}$$

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

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

Note, however, that cloning requires some additional conditions for soundness. The predicate $\mathsf{cloneable}(\mathsf{A})$ ensures the value is of a resource type—such as a reference—or a product of resources. (The full definition of cloneable is presented in Section 5, where we discuss said resource types in more depth.) This is necessary to prevent values which do not come with identifiers from being managed via ownership, since once we introduce borrowing in Section 5 this would lead to borrows of different values of the same type (such as two floating point numbers) being indistinguishable at the type level and thus interchangeable. The side-condition $1 \sqsubseteq r$ explains that we must be able to accommodate a usage of the input t_1 since **clone** consumes the t_1 value once to copy it.

Our implementation of these ideas in Granule follows the above typing. We recap the initial ownership example from Section 2 here in our new extension of Granule using a data type `Colour` which acts as an alias for a triple of type `(Int, (Int, Int))`.

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

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

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

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

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

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

$$\textbf{clone}\ \mathsf{t}_1\ \textbf{as}\ \mathsf{x}\ \textbf{in}\ (\textbf{clone}\ \mathsf{t}_2\ \textbf{as}\ \mathsf{y}\ \textbf{in}\ \mathsf{t}_3) \equiv \textbf{clone}\ (\textbf{clone}\ \mathsf{t}_1\ \textbf{as}\ \mathsf{x}\ \textbf{in}\ \mathsf{t}_2)\ \textbf{as}\ \mathsf{y}\ \textbf{in}\ \mathsf{t}_3 \quad (x \notin \mathsf{FV}(t_3)) \quad (*\text{assoc})$$

The (β_*) axiom states that sharing a value v (term with no further reductions—see Section 6) and cloning it to create a new owned x in the scope of t is equivalent to substituting the original v for x in t (with its identifiers packed in an existential). The $(*\text{assoc})$ axiom is associativity of cloning.

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

5 IMMUTABLY BORROWED IS FRACTIONALLY UNIQUE

We now generalise from the above system which incorporates uniqueness into the core calculus to a system that allows for the uniqueness guarantees to be temporarily broken in controlled ways, such that we can continue to ensure memory safety and more closely approximate Rust-like ownership and borrowing rules. We take inspiration from the fractional permissions of Boyland [2003] (which themselves served as partial inspiration for Rust's notion of ownership) and from recent literature on integrating linear types with fractional permissions [Makwana and Krishnaswami 2019].

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

$$\mathsf{A}, \mathsf{B} ::= \ldots \mid \&_p \mathsf{A} \qquad \text{(types, extended)}$$

$$p, q ::= * \mid f \qquad (\textit{where } f \in \mathbb{Q}, 0 < f \leq 1) \qquad \text{(permissions)}$$

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

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

$$\mathsf{t} ::= \ldots \mid \textbf{withBorrow}\ \mathsf{t}_1\ \mathsf{t}_2 \mid \textbf{split}\ \mathsf{t} \mid \textbf{join}\ \mathsf{t}_1\ \mathsf{t}_2 \qquad \text{(terms, extended)}$$

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

$$\frac{\Gamma_1 \vdash \mathsf{t}_1 : *\mathsf{A} \qquad \Gamma_2 \vdash \mathsf{t}_2 : \&_1 \mathsf{A} \multimap \&_1 \mathsf{B}}{\Gamma_1 + \Gamma_2 \vdash \textbf{withBorrow}\ \mathsf{t}_1\ \mathsf{t}_2 : *\mathsf{B}}\ \textsc{with}\&$$

Here, we allow for a uniquely owned $*\mathsf{A}$ value to be borrowed and manipulated in a mutable way by some function t_2 which expects a $\&_1 \mathsf{A}$ as input, so long as it returns the mutable borrow as a $\&_1 \mathsf{B}$ in the output so that the original owner can reclaim this as a unique reference ($*\mathsf{B}$). By encapsulating the mutable borrow behaviour inside a continuation, we ensure that it is impossible to construct a closed term with a borrowed type, which means borrowed references to values must always eventually return full access to the owner of said value.

Uniqueness as uniquely borrowed. We absorb the development of Section 4 by the inclusion of $* \in p$ and a type identity $*\mathsf{A} \equiv \&_* \mathsf{A}$. Thus, the borrowing graded modality captures both (uniquely) owned and borrowed values and hence functions polymorphic in their permission can range over both kinds of ownership. Furthermore, this enables us to later define operations that can work over uniquely owned or mutably borrowed values without needing multiple primitives.

Immutable borrows. Mutable borrows, graded with permission 1, provide full access to a value, permitting reads and writes (shown later for resources). By contrast, *immutable borrows* allow multiple references to a value as long as they cannot be mutated (which would otherwise violate memory safety, e.g., allowing data races). The **split** and **join** constructs mediate between mutable and immutable borrows:

$$\frac{\Gamma \vdash \mathsf{t} : \&_p \mathsf{A}}{\Gamma \vdash \textbf{split}\ \mathsf{t} : \&_{\frac{p}{2}} \mathsf{A} \otimes \&_{\frac{p}{2}} \mathsf{A}}\ \textsc{split} \qquad \frac{\Gamma_1 \vdash \mathsf{t}_1 : \&_p \mathsf{A} \qquad \Gamma_2 \vdash \mathsf{t}_2 : \&_q \mathsf{A} \qquad p + q \leq 1}{\Gamma_1 + \Gamma_2 \vdash \textbf{join}\ \mathsf{t}_1\ \mathsf{t}_2 : \&_{p+q} \mathsf{A}}\ \textsc{join}$$

where **split** takes a borrowed reference and splits it into two borrowed references to the same value, each graded by half the original permission.[4] Resources with fractional permission < 1 will

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

only permit immutable operations (shown later) and thus splitting mutably borrowed resources ($p = 1$) leads to two immutably borrowed references to the resource. The dual **join** (re)combines two borrows into one with their permissions added, recombining two immutable borrows back to a mutable borrow when $p + q = 1$. Due to the interfaces for manipulating resources (shown later), the type A above will always contain a resource identifier id, so it is impossible to combine references to two different values. The addition and halving operations are defined only for fractions f, not $*$.

These rules do not encompass every detail of Rust's intricate ownership system; they instead aim to represent the *core* concepts of ownership and borrowing in such a way that they can integrate into an existing graded type system, leaving room for potential extensions. In particular, as was mentioned in Section 2, we only support *lexical* lifetimes, meaning that there is a class of programs which are accepted by Rust's compiler but cannot be represented in this calculus. We focus on being able to represent notions of ownership *explicitly* in the types, rather than using a complex static analysis like Rust's borrow checker to *infer* which programs are safe.

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

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

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

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

The unsafe patterns demonstrated by the `viridian` and `cerulean` examples in Section 2 are also not representable in Granule. It is impossible for two mutable borrows or for a mutable borrow and an immutable borrow to coexist here, since splitting a borrow must reduce the permission in its type by the nature of the **`split`** primitive. We can, however, represent *reborrowing*; this simply involves continuing to split immutable borrows further, where recovering the original value now requires collecting all of the borrows once more. The order in which the borrows are rejoined is immaterial. The following example illustrates this.

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

The benefits of uniqueness typing for memory management are well-trodden in prior work [Marshall et al. 2022]; the idea is that the guarantee of a unique reference both allows safe in-place updates but also obviates the need for garbage collection, since memory can be reclaimed as soon as a reference is deleted. The system described here augments this by allowing increased flexibility while preserving the same guarantees—multiple references can be created via **`split`** but they are tracked precisely through fractional grades so that uniqueness is still guaranteed once they are re**join**ed, with in-place update only allowed for references with unique access (graded 1 or $*$).

Resources. Enforcing which behaviours should be allowed for different permissions on a given resource is mediated by said resource's interface. We will give two in-depth examples in this paper to demonstrate the flexibility of our ownership and borrowing interface. The first is mutable arrays of floats that can be created, read from, written to and deleted (each at differing levels of access). This allows us to illustrate one of the key practical benefits of ownership, which is that unique access to an owned value allows for safe mutation (the original pun behind "Linear Types can Change the World" [Wadler 1990]). The second example will be polymorphic mutable references which store a pointer to a value of any type. First, we extend our calculus with a primitive type of resources, along with some additional base types used to parameterise said resources:

$$\mathsf{A} ::= \ldots \mid \mathsf{Res}_{\mathsf{id}}\ \mathsf{A} \mid \mathbb{N} \mid \mathbb{F} \qquad \mathsf{Res} ::= \mathsf{Array} \mid \mathsf{Ref} \qquad \text{(types, extended)}$$

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

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

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

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

This interface resembles Granule's existing interface for array mutation using modal uniqueness types, but with some crucial differences owing to the introduction of borrowing. First, array types are now indexed by an identifier pointing to a particular array object in the heap, generated by the existential type in **newArray** and within the scope of the CLONE rule. This allows us to keep track of which array is being referenced now that multiple borrows pointing to the same array are permitted to exist. The other key difference is that reading and writing no longer require sole ownership. Reading can be carried out at any permission, since this is safe no matter how many references exist, but writing is restricted to either a mutable-borrowed or uniquely-owned array.

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

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

As per the typing rules of existentials presented in Section 3, once we unpack the existential in the second line the type variable `id` must not occur in the type of the body. In this example, the array is deleted and never returned in the body's result, thus satisfying the type restrictions of **`unpack`**. We can also apply the **`clone`** function to make a deep copy of a unique array as described in Section 4:

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

The interface for polymorphic references is similar, with primitives:

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

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

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

In order to allow for non-linear behaviours, we also provide a **readRef** primitive which requires the value to be of a graded type which permits the additional usage accrued with the appropriate type-level accounting; this can be applied as follows.

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

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

One important benefit of introducing polymorphic references is that they allow standard Granule values such as integers or floating-point numbers to be treated uniquely and managed by the ownership system, since otherwise this would be prevented by the cloneable predicate described in Section 4. This allows for the use of general imperative programming patterns as might be applied in languages like Rust (where ownership is the default) to be transferred to the setting of Granule (where functional patterns are more typical).

We are now able to give the full inductive definition of the cloneable predicate, displayed below. As described in Section 4, this permits only resources and products containing resources to be cloned, ensuring that all cloneable values come with an identifier. Note that in order to preserve both linearity and this condition, for a polymorphic reference to be cloneable it must itself point either to a cloneable value or to a value that is freely copyable, such as a floating-point number.

Definition 5.1 (Copyable predicate). Predicate definition:

$$\frac{}{\mathsf{copyable}(\mathsf{unit})} \qquad \frac{}{\mathsf{copyable}(\mathbb{N})} \qquad \frac{}{\mathsf{copyable}(\mathbb{F})} \qquad \frac{\mathsf{copyable}(A) \quad \mathsf{copyable}(B)}{\mathsf{copyable}(A \otimes B)}$$

Definition 5.2 (Cloneable predicate). Predicate definition:

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

Resource allocators. Here, the restriction introduced on the promotion rule in Section 3 meaning that 'resource allocators' are forbidden from being promoted becomes crucial for ensuring soundness in the call-by-value setting of this work. Consider the following pseudocode (eliding packing and unpacking of existentials), which would be allowed in Granule given unrestricted promotion:

```
1  let [x] : ((*(Array id Float)) [2]) = [newArray 1] in
2  let                           () = deleteArray x in writeArray x 0 1.0
```
Granule 7

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

Under a call-by-name semantics, as in previous work on embedding uniqueness types in Granule [Marshall et al. 2022] (and accessible in Granule via the extension `language CBN`), this program executes successfully and produces an array which contains the value written on line 2. The key is that call-by-name reduction substitutes the call to `newArray` into the two uses of the variable x, and so these point to two entirely separate arrays. However, under the call-by-value semantics of this paper (and Granule's default), the first line is fully evaluated, and so both uses of x point to the same array. Thus the second line attempts to write to an array after it has been deleted, which would cause a runtime error. In a call-by-value setting this program must not be permitted.

The solution we apply here (also used in recent work on graded session types [Marshall and Orchard 2022c]) is to syntactically restrict promotion to terms which do not allocate resources, i.e., that do not use **newArray** or **newRef** in a reduction position. The predicate $\mathsf{resourceAllocator}(t)$ classifies precisely these terms, as a kind of specialised "value restriction"; note in particular that $\neg\mathsf{resourceAllocator}(\lambda x.\mathbf{newArray}\ t)$ since reduction does not happen underneath an abstraction (Section 6 defines the reduction semantics). The appendix includes the full inductive definition.

Other work resolves this same problem relating to promotion of resource allocators through different techniques. Originally, the linear types extension to Haskell got around this difficulty by only ever allocating resources inside a specialised continuation, which is passed around at every step until deallocation to prevent the allocator itself from ever being used non-linearly. More recently, this strategy has been generalised by introducing a "linear constraint" [Spiwack et al. 2022] called `Linearly`. This constraint, which must itself be used in a linear fashion, is assumed whenever a new resource is allocated. A continuation is still necessary for the initial assumption of `Linearly`, but the same qualification may now be used generically for varying resource types.

5.1 Equational Theory

The WITH& typing suggests a close relationship between $\&_{*}A$ and $\&_{1}A$. By analogy to the notion of a relative monad used in prior work [Marshall et al. 2022], $\&_{*}A$ acts as a 'relative functor' with regard to $\&_1 \mathsf{A}$, where **withBorrow** equates to mapping a function involving mutable borrows onto a function between uniquely owned values. The following axioms for functors then hold:

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

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

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

Lastly, **split** and **join** have additional properties which form an isomorphism:

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

$$\textbf{split}\ (\textbf{join}\ \mathsf{t}_1\ \mathsf{t}_2) \equiv (\mathsf{t}_1, \mathsf{t}_2) \qquad (\&\text{resplit})$$

5.2 Divide and Conquer: Partial Views via Distributive Laws

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

Our core calculus provides some capacity for structured data in the form of product types. We introduce two additional constructs for distributing the borrowing graded modality into and out of products, enabling the pattern of partial borrowing:[5]

$$\frac{\Gamma \vdash \mathsf{t} : \&_p(\mathsf{A} \otimes \mathsf{B})}{\Gamma \vdash \textbf{push}\,\mathsf{t} : (\&_p\mathsf{A}) \otimes (\&_p\mathsf{B})}\ \text{PUSH} \qquad \frac{\Gamma \vdash \mathsf{t} : (\&_p\mathsf{A}) \otimes (\&_p\mathsf{B})}{\Gamma \vdash \textbf{pull}\,\mathsf{t} : \&_p(\mathsf{A} \otimes \mathsf{B})}\ \text{PULL}$$

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

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

Granule

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

[5]One might wonder whether the □ modality also distributes in this way; the answer is not in general, though some choices of semiring permit this behaviour. This question has been explored in depth in prior work [Hughes et al. 2021a].

Partial borrowing allows safe mutation of disjoint parts of a single data structure *concurrently* without risking a data race, e.g., using `par` for concurrent threads [Marshall and Orchard 2022c]:

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

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

Building on this core concept in the more expressive setting of Granule's full type system could involve notions such as borrowing from any algebraic data type, or borrowing a 'slice' of multiple values. One low-hanging fruit is to extend the **push** and **pull** primitives in order to derive generic operations for arbitrary data structures, following patterns described in prior work [Hughes et al. 2021b]. We do not formalise this again here, but the approach allows for writing programs like:

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

Here, the borrow modality is distributed into the `Maybe` data type using a derived **push** operation. This general pattern for partial borrowing relates closely to McBride's notion of computing the *derivative* of a data type by finding its type of one-hole contexts [McBride 2001], where the derivative describes the structure that remains after borrowing one element. Under this interpretation, borrowing a slice of multiple values would be akin to taking the derivative of a data type *with respect to* another type—an idea described elsewhere in recent work [Marshall and Orchard 2022b].

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

First, in order to implement this function elegantly we will need two additional basic functions for working with arrays and references. Within Granule it is possible to derive a `writeRef` operation which destructively mutates the value inside a reference, as long as the type of said value allows for it to be discarded freely; this makes use of Granule's existing mechanism for deriving a `drop` operation (of type `t → ()`) for this type of value [Hughes et al. 2021b]. Granule also provides an additional primitive for taking the length of an array, which is useful for iteration but was elided from our formal calculus for brevity. The types of both of these operations are given below.

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

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

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

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

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

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

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

                   in compOut) compIn
```
Granule

The full code of this example is given in the appendix (A.4), including full definitions for all auxiliary functions and an illustration of how this function could be applied in practice to a freshly created pair of an array (which is generated from a length-indexed vector) and a polymorphic reference.

6 SEMANTICS AND METATHEORY

We define an operational semantics here for the core calculus which serves to further explain the details of arrays, mutation, copying, and borrowing. We adapt the approach of Choudhury et al. [2021] and Marshall et al. [2022] for giving an operational semantics to a graded system, with some degree of accounting for grades and references in order to relate the dynamic semantics back to the static semantics of typing (Section 6.3). With the exception of Granule and Multi-Graded Featherweight Java [Bianchini et al. 2023a], much of the preceding work on operational models for graded systems is based on call-by-name. Here we opted for call-by-value for the purpose of describing real-world practical functional languages; call-by-name is prohibitively expensive with unpredictable performance and poor interaction with side effects.

6.1 Preliminary Definitions

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

$$\mathsf{v} ::= (\mathsf{v}_1, \mathsf{v}_2) \mid () \mid [\mathsf{v}] \mid \lambda \mathsf{x}.\mathsf{t} \mid \mathsf{n} \mid p \mid \textbf{pack}\ \langle \mathsf{id}, \mathsf{v} \rangle \qquad \text{(value terms sub-grammar)}$$

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

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

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

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

where $*\mathsf{t}$ represents unique and borrowed terms, **unborrow** t is the inverse to borrowed terms (used to implement **withBorrow**), and ref are references to resources bound in the heap. Furthermore, the syntax for promotion is augmented with an annotation $[\mathsf{t}]_r$ of the grade r at which the term t is promoted, i.e., Church-style with respect to grades, and thus typing this grade-annotated version of promotion produces $r \cdot \Gamma \vdash [\mathsf{t}]_r : \ {}_r\mathsf{A}$. This annotation is only needed to prove that type preservation respects resourcing (as per Choudhury et al. [2021]) and can be ignored when actually computing the reduction behaviour of a term using the operational model.

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

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

$$(\Gamma, \mathsf{ref} : \mathsf{Res}_{\mathsf{id}}\ \mathsf{A}) + (\Gamma', \mathsf{ref} : \mathsf{Res}_{\mathsf{id}}\ \mathsf{A}) = (\Gamma + \Gamma'), \mathsf{ref} : \mathsf{Res}_{\mathsf{id}}\ \mathsf{A} \qquad \text{(runtime context addition)}$$

$$(\Gamma, \mathsf{ref} : \mathsf{Res}_{\mathsf{id}}\ \mathsf{A}) + \Gamma' = (\Gamma + \Gamma'), \mathsf{ref} : \mathsf{Res}_{\mathsf{id}}\ \mathsf{A} \quad (\textit{where}\ \mathsf{ref}\ \square\ \Gamma')$$

$$r \cdot (\Gamma, \mathsf{ref} : \mathsf{Res}_{\mathsf{id}}\ \mathsf{A}) = (r \cdot \Gamma), \mathsf{ref} : \mathsf{Res}_{\mathsf{id}}\ \mathsf{A} \qquad \text{(runtime context multiplication)}$$

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

$$\frac{\gamma \vdash \mathsf{t} : \mathsf{A}}{0 \cdot \Gamma, \gamma \vdash *\mathsf{t} : \&_p \mathsf{A}}\ \text{NEC} \qquad \frac{\Gamma \vdash \mathsf{t} : \&_1 \mathsf{A}}{\Gamma \vdash \textbf{unborrow}\ \mathsf{t} : *\mathsf{A}}\ \text{UNBORROW} \qquad \frac{}{0 \cdot \Gamma, \mathsf{ref} : \mathsf{Res}_{\mathsf{id}}\ \mathsf{A} \vdash \mathsf{ref} : \mathsf{Res}_{\mathsf{id}}\ \mathsf{A}}\ \text{REF}$$

Heaps and configurations. Our semantics does not use syntactic substitution; instead *heaps* map program variables to values, resource references to identifiers id, and identifiers to resource values:

$$H ::= \emptyset \mid H, \mathsf{x} \mapsto_r \mathsf{v} \mid H, \mathsf{ref} \mapsto_p \mathsf{id} \mid H, \mathsf{id} \mapsto v_r \qquad \text{(heaps)}$$

$$v_r ::= \textbf{arr} \mid \textbf{ref}(\mathsf{v}) \qquad \textbf{arr} ::= \mathsf{init} \mid \textbf{arr}[\mathsf{n}] = \mathsf{v} \qquad \text{(heap resource terms)}$$

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

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

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

6.2 Single-Step Reduction

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

$$H_1 \vdash \mathsf{t}_1 \rightsquigarrow_s H_2 \vdash \mathsf{t}_2$$

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

Lambda calculus. The λ-calculus core of the operational semantics has rules:

$$\frac{\exists r'.\, s + r' \sqsubseteq r}{H, \mathsf{x} \mapsto_r \mathsf{v} \vdash \mathsf{x} \rightsquigarrow_s H, \mathsf{x} \mapsto_r \mathsf{v} \vdash \mathsf{v}} \rightsquigarrow_{\text{VAR}} \qquad \frac{\mathsf{y}\#\{H, \mathsf{v}, \mathsf{t}\}}{H \vdash (\lambda \mathsf{x}.\mathsf{t})\, \mathsf{v} \rightsquigarrow_s H, \mathsf{y} \mapsto_s \mathsf{v} \vdash \mathsf{t}[\mathsf{y}/\mathsf{x}]} \rightsquigarrow_\beta$$

$$\frac{H \vdash \mathsf{t}_1 \rightsquigarrow_s H' \vdash \mathsf{t}_1'}{H \vdash \mathsf{t}_1\, \mathsf{t}_2 \rightsquigarrow_s H' \vdash \mathsf{t}_1'\, \mathsf{t}_2} \rightsquigarrow_{\text{APPL}} \qquad \frac{H \vdash \mathsf{t}_2 \rightsquigarrow_s H' \vdash \mathsf{t}_2'}{H \vdash \mathsf{v}\, \mathsf{t}_2 \rightsquigarrow_s H' \vdash \mathsf{v}\, \mathsf{t}_2'} \rightsquigarrow_{\text{APPR}}$$

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

Existential types and names. The semantics of existentials is standard, with a beta rule:

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

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

Tensors and units. Tensor products have the following rules for their introduction and elimination forms, with three congruence rules and one β-rule:

$$\frac{H \vdash \mathsf{t}_1 \rightsquigarrow_s H' \vdash \mathsf{t}_1'}{H \vdash (\mathsf{t}_1, \mathsf{t}_2) \rightsquigarrow_s H' \vdash (\mathsf{t}_1', \mathsf{t}_2)} \rightsquigarrow_{\otimes\text{L}} \quad \frac{H \vdash \mathsf{t}_1 \rightsquigarrow_s H' \vdash \mathsf{t}_1'}{H \vdash \textbf{let}\, (x, y) = \mathsf{t}_1\, \textbf{in}\, \mathsf{t}_2 \rightsquigarrow_s H' \vdash \textbf{let}\, (x, y) = \mathsf{t}_1'\, \textbf{in}\, \mathsf{t}_2} \rightsquigarrow_{\text{LET}\otimes}$$

$$\frac{H \vdash \mathsf{t}_2 \rightsquigarrow_s H' \vdash \mathsf{t}_2'}{H \vdash (\mathsf{v}, \mathsf{t}_2) \rightsquigarrow_s H' \vdash (\mathsf{v}, \mathsf{t}_2')} \rightsquigarrow_{\otimes\text{R}} \quad \frac{\mathsf{x}'\#\{H, \mathsf{v}_1, \mathsf{v}_2, \mathsf{t}\} \qquad \mathsf{y}'\#\{H, \mathsf{v}_1, \mathsf{v}_2, \mathsf{t}\}}{H \vdash \textbf{let}\, (x, y) = (\mathsf{v}_1, \mathsf{v}_2)\, \textbf{in}\, \mathsf{t} \rightsquigarrow_s H, \mathsf{x}' \mapsto_s \mathsf{v}_1, \mathsf{y}' \mapsto_s \mathsf{v}_2 \vdash \mathsf{t}[\mathsf{y}'/\mathsf{y}][\mathsf{x}'/\mathsf{x}]} \rightsquigarrow_{\otimes\beta}$$

In the case of ($\rightsquigarrow_{\otimes\beta}$), we extend the heap with assignments for (fresh names) x' and y' to v_1 and v_2 respectively, continuing on with the body term t. Similarly to the β-rule for functions, the freshening avoids variable capture. We elide the rules for the unit type as they are similar.

Example 6.1. The following gives a reduction sequence for the term $(\lambda \mathsf{x}.(\mathsf{x}, \mathsf{y}))\, (\mathsf{v}, \mathsf{y})$ from Example 3.1 under a heap $H = \mathsf{y} \mapsto_2 \mathsf{v}$, until a normal form is reached:

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

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

$$\frac{H \vdash \mathsf{t} \rightsquigarrow_{s*r} H' \vdash \mathsf{t}'}{H \vdash [\mathsf{t}]_r \rightsquigarrow_s H' \vdash [\mathsf{t}']_r} \rightsquigarrow \qquad \frac{H \vdash \mathsf{t}_1 \rightsquigarrow_s H' \vdash \mathsf{t}_1'}{H \vdash \textbf{let}\,[\mathsf{x}] = \mathsf{t}_1\,\textbf{in}\,\mathsf{t}_2 \rightsquigarrow_s H' \vdash \textbf{let}\,[\mathsf{x}] = \mathsf{t}_1'\,\textbf{in}\,\mathsf{t}_2} \rightsquigarrow_{\text{LET}}$$

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

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

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

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

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

where ref and id are fresh for the heap H. The resulting array is initialised to the empty array (init), and the result is a unique array reference $*\mathsf{ref}$ packed in an existential over id. In the heap, ref is marked with the whole permission 1. Arrays are read and written via:

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

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

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

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

Sharing and cloning. Sharing reduces permissions to 0 and cloning involves copying heap terms:

$$\frac{H \vdash \mathsf{t} \rightsquigarrow_s H' \vdash \mathsf{t}'}{H \vdash \textbf{share}\,\mathsf{t} \rightsquigarrow_s H' \vdash \textbf{share}\,\mathsf{t}'} \rightsquigarrow_{\text{SHARE}} \qquad \frac{\mathsf{dom}(H) \equiv \mathsf{refs}(\mathsf{v})}{H, H' \vdash \textbf{share}\,(*\mathsf{v}) \rightsquigarrow_s ([H]_0), H' \vdash [\mathsf{v}]} \rightsquigarrow_{\text{SHARE}\beta}$$

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

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

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

The $\rightsquigarrow_{\text{CLONE}\beta}$ rule enacts a 'deep copy', where $\mathsf{dom}(H') \equiv \mathsf{refs}(\mathsf{v})$ marks the part of the heap with resource references coming from v. Then $\mathsf{copy}(H')$ copies the resources in this part of the heap, creating a heap fragment H'' and a renaming operator θ which maps from old references to new copied references. This renaming is applied to v in the freshly bound variable y, such that the value

$*(\theta(\mathsf{v}))$ refers to any newly copied resources. Lastly, we pack the renamed unique value with new identifiers $\overline{\mathsf{id}}$ generated by copy.

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

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

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

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

$$\frac{H \vdash \mathsf{t} \rightsquigarrow_s H' \vdash \mathsf{t}'}{H \vdash \textbf{unborrow}\ \mathsf{t} \rightsquigarrow_s H' \vdash \textbf{unborrow}\ \mathsf{t}'} \rightsquigarrow_{\text{UNBORROW}} \qquad \frac{}{H \vdash \textbf{unborrow}\ (*\mathsf{v}) \rightsquigarrow_s H \vdash *\mathsf{v}} \rightsquigarrow_{\text{UN\&}}$$

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

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

$$\frac{}{H \vdash \textbf{push}\ (*(\mathsf{v}_1, \mathsf{v}_2)) \rightsquigarrow_s H \vdash (*\mathsf{v}_1, *\mathsf{v}_2)} \rightsquigarrow_{\text{PUSH}*} \qquad \frac{}{H \vdash \textbf{pull}\ (*\mathsf{v}_1, *\mathsf{v}_2) \rightsquigarrow_s H \vdash *(\mathsf{v}_1, \mathsf{v}_2)} \rightsquigarrow_{\text{PULL}*}$$

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

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

$$\frac{\mathsf{ref}_1\#H \qquad \mathsf{ref}_2\#H}{H, \mathsf{ref} \mapsto_p \mathsf{id}, \mathsf{id} \mapsto \mathsf{v} \vdash \textbf{split}\ (*\mathsf{ref}) \rightsquigarrow_s H, \mathsf{ref}_1 \mapsto_{\frac{p}{2}} \mathsf{id}, \mathsf{ref}_2 \mapsto_{\frac{p}{2}} \mathsf{id}, \mathsf{id} \mapsto \mathsf{v} \vdash (*\mathsf{ref}_1, *\mathsf{ref}_2)} \rightsquigarrow_{\text{SPLITREF}}$$

$$\frac{\mathsf{ref}\,\#H}{H, \mathsf{ref}_1 \mapsto_p \mathsf{id}, \mathsf{ref}_2 \mapsto_q \mathsf{id}, \mathsf{id} \mapsto \mathsf{v} \vdash \textbf{join}\ (*\mathsf{ref}_1)\ (*\mathsf{ref}_2) \rightsquigarrow_s H, \mathsf{ref} \mapsto_{(p+q)} \mathsf{id}, \mathsf{id} \mapsto \mathsf{v} \vdash *\mathsf{ref}} \rightsquigarrow_{\text{JOINREF}}$$

$$\frac{\begin{array}{c} H \vdash \textbf{split}\ (*\mathsf{v}) \rightsquigarrow_s H' \vdash (*\mathsf{v}_1, *\mathsf{v}_2) \\ H' \vdash \textbf{split}\ (*\mathsf{w}) \rightsquigarrow_s H'' \vdash (*\mathsf{w}_1, *\mathsf{w}_2) \end{array}}{H \vdash \textbf{split}\ (*(\mathsf{v}, \mathsf{w})) \rightsquigarrow_s H'' \vdash (*(\mathsf{v}_1, \mathsf{w}_1), *(\mathsf{v}_2, \mathsf{w}_2))} \rightsquigarrow_{\text{SPLIT}\otimes} \qquad \frac{\begin{array}{c} H \vdash \textbf{join}\ (*\mathsf{v}_1)\ (*\mathsf{v}_2) \rightsquigarrow_s H' \vdash *\mathsf{v} \\ H' \vdash \textbf{join}\ (*\mathsf{w}_1)\ (*\mathsf{w}_2) \rightsquigarrow_s H'' \vdash *\mathsf{w} \end{array}}{H \vdash \textbf{join}\ (*(\mathsf{v}_1, \mathsf{w}_1))\ (*(\mathsf{v}_2, \mathsf{w}_2)) \rightsquigarrow_s H'' \vdash *(\mathsf{v}, \mathsf{w})} \rightsquigarrow_{\text{JOIN}\otimes}$$

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

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

Join and split also have congruence rules which are straightforward and elided.

Multi-step reductions. Lastly, we define a relation that composes single-step reductions into a sequence of reductions, called a multi-reduction:

$$\frac{}{H \vdash \mathsf{t} \ \Rightarrow_s \ H \vdash \mathsf{t}}\ \text{REFL} \qquad \frac{H \vdash \mathsf{t}_1 \ \rightsquigarrow_s \ H' \vdash \mathsf{t}_2 \qquad H' \vdash \mathsf{t}_2 \ \Rightarrow_s \ H'' \vdash \mathsf{t}_3}{H \vdash \mathsf{t}_1 \ \Rightarrow_s \ H'' \vdash \mathsf{t}_3}\ \text{EXT}$$

6.3 Theorems

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

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

Lemma 6.2 (Linear substitution is admissible, extending [Orchard et al. 2019]). *If* $\Gamma_1 \vdash \mathsf{t}_1 : \mathsf{A}$ *and* $\Gamma_2, \mathsf{x} : \mathsf{A} \vdash \mathsf{t}_2 : \mathsf{B}$ *then* $\Gamma_2 + \Gamma_1 \vdash \mathsf{t}_2[\mathsf{t}_1/\mathsf{x}] : \mathsf{B}$.

Lemma 6.3 (Graded substitution is admissible, extending [Orchard et al. 2019]). *If* $[\Gamma_1] \vdash \mathsf{t}_1 : \mathsf{A}$ *and* $\Gamma_2, \mathsf{x} : [\mathsf{A}]_r \vdash \mathsf{t}_2 : \mathsf{B}$ *(where* $[\Gamma_1]$ *represents a context* Γ_1 *containing only graded assumptions) and* $\neg\mathsf{resourceAllocator}(\mathsf{t}_1)$ *then* $\Gamma_2 + r \cdot \Gamma_1 \vdash \mathsf{t}_2[\mathsf{t}_1/\mathsf{x}] : \mathsf{B}$.

Type safety. Key to ensuring type safety is the notion of *heap compatibility with a typing context.*

Definition 6.4 (Heap compatibility). A heap H is compatible with free variable context Γ, denoted $H \ \square\ \Gamma$, if the grades in the heap match those of the context, and any values stored in the heap have their resources accounted for in the rest of the heap. The relation is defined inductively over the syntax of heaps and contexts:

$$\frac{}{\emptyset \ \square\ \emptyset}\ \text{EMPTY} \qquad \frac{H, \mathsf{id} \mapsto v_r \ \square\ \Gamma + \gamma \qquad \gamma \vdash v_r : \mathsf{Res}_{\mathsf{id}}\ \mathsf{A}}{H, \mathsf{ref} \mapsto_p \mathsf{id}, \mathsf{id} \mapsto v_r \ \square\ (\Gamma, \mathsf{ref} : \mathsf{Res}_{\mathsf{id}}\ \mathsf{A})}\ \text{EXTRES} \qquad \frac{H \ \square\ \emptyset}{H, \mathsf{ref} \mapsto_p \mathsf{id} \ \square\ \emptyset}\ \text{GCARR}$$

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

$$\frac{H \ \square\ \Gamma + \Gamma' \qquad \mathsf{x} \ \square\ \mathsf{dom}(H) \qquad \Gamma' \vdash \mathsf{v} : \mathsf{A} \qquad \exists r'.\ 1 + r' \equiv r}{(H, \mathsf{x} \mapsto_r \mathsf{v}) \ \square\ (\Gamma, \mathsf{x} : \mathsf{A})}\ \text{EXTLIN}$$

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

The (ext) rule says that a context with graded assumption $\mathsf{x} : [\mathsf{A}]_s$ is compatible with a heap as long as the heap contains a binding for x to some value v and heap grade r that can accommodate the usage of s (via the constraint $\exists r'.\ s + r' \equiv r$) and as long as the free variables Γ' of v are also compatible with the heap (scaled by s to reflect the usage of x). The (extLin) rule is similar to (ext), but *effectively* where $s = 1$; the variable x is used in a linear fashion.

Example 6.5. As an illustrative example, the context $\mathsf{x} : [\mathsf{A}]_1, \mathsf{y} : [\mathsf{B}]_2$ is compatible with the heap $\mathsf{x} \mapsto_7 \mathsf{v}_1, \mathsf{y} \mapsto_2 \mathsf{v}_2$ assuming the typing $\mathsf{x} : [\mathsf{A}]_3 \vdash \mathsf{v}_2 : \mathsf{B}$, with heap compatibility derivation:

$$\frac{\dfrac{\dfrac{}{\emptyset \ \square\ \emptyset}\ \text{EMPTY}}{\emptyset, \mathsf{x} \mapsto_7 \mathsf{v} \ \square\ \mathsf{x} : [\mathsf{A}]_7}\ \text{EXT} \qquad \mathsf{x} : [\mathsf{A}]_3 \vdash \mathsf{v}_2 : \mathsf{B}}{(\emptyset, \mathsf{x} \mapsto_7 \mathsf{v}_1, \mathsf{y} \mapsto_2 \mathsf{v}_2) \ \square\ (\mathsf{x} : [\mathsf{A}]_1, \mathsf{y} : [\mathsf{B}]_2)}\ \text{EXT}$$

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

Theorem 6.6 (Progress). *Given* $\Gamma \vdash \mathsf{t} : \mathsf{A}$*, then* t *is either a value, or for all grades* s *and contexts* Γ_0 *then if* $H \Box \Gamma_0 + s \cdot \Gamma$ *there exists a heap* H' *and term* t' *such that* $H \vdash \mathsf{t} \rightsquigarrow_s H' \vdash \mathsf{t}'$*.*

Theorem 6.7 (Type Preservation). *For a well-typed term* $\Gamma \vdash \mathsf{t} : \mathsf{A}$*, under a restriction that polymorphic reference resources are restricted to non-function types, and for all* s*,* Γ_0*, and* H *such that* $H \Box (\Gamma_0 + s \cdot \Gamma)$ *and a reduction* $H \vdash \mathsf{t} \rightsquigarrow_s H' \vdash \mathsf{t}'$*, then we have:*

$$\exists \Gamma', H'.\ \ \Gamma' \vdash \mathsf{t}' : \mathsf{A} \ \ \wedge \ \ H' \Box (\Gamma_0 + s \cdot \Gamma')$$

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

In addition to preservation, Marshall et al. [2022] also proved a *conservation* property, ensuring that resource usage is respected. In particular, they show resource usage accrued in a given reduction plus remaining resources in the resulting heap are approximated by the resources in the original heap plus the specified resource usage from any variable bindings encountered along the way. The semantics of Marshall et al. [2022] was call-by-name, which is more natural for conservation; checking that a modified form of conservation still holds in our setting is future work, though various other graded type systems with similar properties are also call-by-value [Bianchini et al. 2022; Orchard et al. 2019]. Our inclusion of semiring grades in the heap here goes towards being able to extend the semantics to full accounting in future work, although they could be elided here and the above results would still be provable. The tracking of permission grades is however crucial to the next results.

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

Lemma 6.8 (Borrow safety). *For a well-typed term* $\Gamma \vdash \mathsf{t} : \mathsf{A}$ *and all* Γ_0*,* s*, and heaps* H *such that* $H \Box (\Gamma_0 + s \cdot \Gamma)$*, and given a single-step reduction* $H \vdash \mathsf{t} \rightsquigarrow_s H' \vdash \mathsf{t}'$ *then for all* $\mathsf{id} \in \mathsf{dom}(H)$*:*

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

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

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

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

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

Theorem 6.9 (Multi-reduction borrow safety). *For a well-typed term* $\Gamma \vdash \mathsf{t} : \mathsf{A}$ *and all* Γ_0*,* s*, and* H *such that* $H \vDash (\Gamma_0 + s \cdot \Gamma)$*, and multi-step reduction* $H \vdash \mathsf{t} \Rightarrow_s H' \vdash \mathsf{v}$*, then for all* $\mathsf{id} \in \mathsf{dom}(H)$*:*

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

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

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

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

The uniqueness theorem presented by Marshall et al. [2022] now follows as a direct corollary of the multi-step borrow safety theorem we described above.

Corollary 6.10 (Uniqueness). *For a well-typed term* $\Gamma \vdash \mathsf{t} : {*}\mathsf{A}$ *and all* Γ_0*,* s*, and* H *such that* $H \vDash (\Gamma_0 + s \cdot \Gamma)$ *and multi-reduction to a value* $H \vdash \mathsf{t} \Rightarrow_s H' \vdash {*}\mathsf{v}$*, for all* $\mathsf{id} \in \mathsf{dom}(H)$ *then:*

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

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

Finally, we see that the operational semantics we have defined (extended to full β-reduction) supports the equational theory we have gradually developed in this work. Note that the proof includes only those equations defined by the ≡ relation presented throughout Sections 4 and 5; the remaining equations, presented in the appendix, are as standard for a graded linear calculus.

Theorem 6.11 (Soundness with respect to the equational theory). *For all* $\mathsf{t}_1, \mathsf{t}_2$ *such that* $\Gamma \vdash \mathsf{t}_1 : \mathsf{A}$ *and* $\Gamma \vdash \mathsf{t}_2 : \mathsf{A}$ *and* $\mathsf{t}_1 \equiv \mathsf{t}_2$ *and given* H *such that* $H \vDash \Gamma$*, there exist multi-reductions to values that are equal under full* β*-reduction and evaluating any references to the value they point to in the resulting heaps (written* $H'(\mathsf{v}_1)$ *and* $H'(\mathsf{v}_2)$*):*

$$H \vdash \mathsf{t}_1 \Rightarrow_1 H' \vdash \mathsf{v}_1 \quad \wedge \quad H \vdash \mathsf{t}_2 \Rightarrow_1 H'' \vdash \mathsf{v}_2 \quad \wedge \quad H'(\mathsf{v}_1) \equiv H''(\mathsf{v}_2)$$

Full proofs for the above theorems may be found in the appendices, provided as supplementary material, along with collected typing and reduction rules [Marshall and Orchard 2024a].

7 RELATED WORK

Linear and graded types. The notion of linearity originated with Girard's linear logic [Girard 1987], which treats information resourcefully by restricting the structural rules of intuitionistic logic. This was soon adopted by programming language researchers who were interested in resourceful behaviour of data, rapidly developing into the concept of *linear types* [Wadler 1990, 1993], but it took substantial time for linearity to emerge into the realm of practical programming. Lately, linear types have seen a renaissance due to their adoption as an extension to Haskell [Bernardy et al. 2017]; other languages incorporating linearity (or affinity, which is similar but allows weakening) include ATS [Zhu and Xi 2005], Alms [Tov and Pucella 2011] and Mezzo [Balabonski et al. 2016].

This binary view (linear vs. non-linear) was later refined by the notion of *bounded* linear logic [Girard et al. 1992], which introduced a family of operators indexed by a polynomial giving an upper bound on the usage of a resource. Further generalisations of this idea to track a broader range of increasingly fine-grained properties are what led to the introduction of *graded* types, which allow for annotating values with precise information about how they interact with their environment.

Granule's particular approach to graded modalities draws heavily from literature on *coeffects*, which describe how programs *depend* on their context [Petricek et al. 2014], though it also incorporates ideas from effect systems which we focus on less here. Coeffect systems developed concurrently with other work which approached the same target from a different angle—through attempting to find generalisations of bounded linear logic [Brunel et al. 2014; Ghica and Smith 2014]. Other instances of graded types include Quantitative Type Theory (QTT) [Atkey 2018; McBride 2016] upon which the type system for Idris 2 is built [Brady 2021], the core calculus underlying Linear Haskell [Bernardy et al. 2017; Spiwack et al. 2022], and others [Abel and Bernardy 2020; Abel et al. 2023; Gaboardi et al. 2016; Wood and Atkey 2022].

Uniqueness. Uniqueness types originated in the Clean language [Smetsers et al. 1994], where they are used in lieu of monadic computation and for efficiency gains offered by in-place update. In Clean, computation is based on graph rewriting and reduction; constants are graphs, and functions are graph rewriting formulas. This gives the type system a different feel to those of more recent functional languages. Recent work has attempted to capture benefits of uniqueness while allowing a more modern programming style, such as Cogent [O'Connor et al. 2021], Mercury [Somogyi et al. 1996] and the prior iteration of uniqueness in Granule [Marshall et al. 2022].

Theoretical work on understanding uniqueness began with Harrington's uniqueness logic [Harrington 2006]; this was followed by a substantial amount of theoretical groundwork on Clean's particular strategy for uniqueness [de Vries et al. 2008]. This paper clarified the distinction between Clean's type system and systems based on the λ-calculus. Further work made headway on distinguishing uniqueness from other substructural systems, allowing for applications such as polymorphic programming and concurrency [de Vries 2013; de Vries et al. 2009]. This laid a foundation for Granule's orthogonal approach to uniqueness, allowing for uniqueness and linearity to be integrated in a single system, which lead to the extensions developed in the present work.

Region-based memory management. Regions [Tofte et al. 2004] were conceived to bring benefits of traditional stack-based memory management into higher-order functional languages. Regions divide values using lifetimes; as with modern ownership systems, they eliminate the need for garbage collection by using region information to allow safe allocation and deallocation. Historically, regions have often been used for effect systems [Jouvelot and Gifford 1991; Lucassen and Gifford 1988].

Later work on regions extends this stack-based foundation by making use of uniqueness information [Walker et al. 2000]; a unique reference ensures that a region has no aliases, so it can be deallocated efficiently. Similarly to lifetimes in ownership systems, regions act as equivalence classes for a "may alias" relation; values which do not share a region may not alias with one another, so if a value does not share a region with any other then it can be safely mutated.

Work on Cyclone (a 'safe dialect' of C [Hicks et al. 2004]) clarified the relationship between references and regions, observing that "unique pointers are essentially lightweight, dynamic regions that hold exactly one object [Fluet et al. 2006]." Rust's lifetimes were inspired by regions. One extension of ML supports both linearity and borrowing using regions [Radanne et al. 2020].

Ownership and borrowing. Ownership was developed as a framework for understanding aliasing in object-oriented languages [Clarke et al. 1998; Mycroft and Voigt 2013], with related work introducing notions like 'islands' [Hogg 1991], 'balloon types' [Almeida 1997] and external uniqueness [Clarke and Wrigstad 2003]. The intent of ownership is to give a high-level structural view of objects and references, akin to type systems which allow for a high-level structural view of data.

More recently, ownership is pervasively used in Rust in order to help ensure memory safety. Multiple formalisations for Rust's ownership model have been attempted; RustBelt [Jung et al. 2017] gives a lower-level encoding of Rust intended for formal verification while Oxide [Weiss et al. 2021] is a higher-level encoding designed for theoretical work, among others [Jung et al. 2019; Pearce 2021]. Rust is not the only modern language to make use of ownership; other languages like Swift incorporate similar ideas, and work on introducing ownership into languages with manual memory management is ongoing [Sammler et al. 2021]. Lorenzen et al. [2023] describe how to use related ideas to determine which functional programs can be executed in-place without allocation, and recent work on *reachability types* [Bao et al. 2021; Wei et al. 2024] scales reasoning about lifetimes and sharing into higher-order and polymorphic settings, taking inspiration from separation logic.

Rust also provides an *unsafe* mode (a superset of the safe portion of Rust). While in Rust memory is managed through ownership and borrowing by default with memory for unsafe code needing to be managed manually, in languages like Granule managing memory automatically through a garbage collector is the default. Using our extension to obviate the need for garbage collection is a special case applying to some subset of a program. If the user does not wish for ownership to be taken into consideration, they can use Granule's pre-existing type system as before.

8 CONCLUSIONS AND FUTURE WORK

Performance improvements. This paper has generalised uniqueness and linearity in order to develop a unified framework for reasoning about ownership and grading. While we have an implementation of this built upon the Granule compiler, the practical benefits it can offer for resourceful programming have not been fully explored. We hope to evaluate **performance improvements** that introducing precise ownership tracking into a functional language can offer through obviating garbage collection for a subset of a program, by collecting examples involving ownership and borrowing and benchmarking them against equivalent functional code.

Guarantees. Linearity is only one property that can be tracked via *coeffects*, the flavour of grading we consider; many others have been described in prior work [Petricek et al. 2014]. If uniqueness dualises linearity, translating the same relationship onto other coeffects may result in other interesting properties that can be tracked in a similar manner; for example, it has been noted that in the realm of information flow tracking for security, *confidentiality* can be understood as a coeffect with *integrity* as its dual [Marshall and Orchard 2022a]. It would be valuable to find an analogy for the ownership generalisation developed here in the context of security, and also to go further and develop a *general* algebraic theory for **guarantees** about global program behaviour.

Counting permissions. Fractional permissions and the general notion of dividing a single mutable borrow into many immutable borrows is not the only model for representing borrowing that we considered. One interesting path for future research would be to explore an alternative strategy for graded uniqueness that is less symmetrical but instead privileges the original owner, allowing for a more exact count of other extant references. This model, based on **counting permissions** [Bornat et al. 2005], involves having many references with only read permissions, and a designated owner that keeps a count of how many such references exist, so that uniqueness can still be recovered.

Non-lexical lifetimes. The model of ownership developed here was *not* intended to be a complete model of every aspect of Rust's ownership system, instead aiming to extract essential features and develop a framework where they coexist with more traditional linear and graded types. It would still be of interest to pursue this further and capture Rust's more advanced features—in particular, incorporating notions such as **non-lexical lifetimes** would increase the power of our system. This could eventually allow for an encoding study between Granule's calculus and a more faithful representation of Rust's type system such as λ_{Rust} [Jung et al. 2017]. It would also be useful to uncover general principles for extending our type system with further resource interfaces, beyond the particular examples introduced in this work.

Adjoint decomposition and categorical models. Finally, alongside the operational heap model presented in this paper it would be interesting to explore a **categorical model** based on **adjunctions**. Benton's linear/non-linear (LNL) logic [Benton 1995] is well-known, and progress has been made on using similar tools to understand more advanced substructural systems involving graded types (much like Granule's core calculus as extended in this paper) but with an adjoint presentation [Hanukaev and III 2023; III and Orchard 2020; Vollmer et al. 2024]. Despite the close theoretical relationship between linearity and uniqueness, the categorical background of uniqueness has been only briefly explored [Harrington 2006], and for more complex ownership and borrowing systems even less so. This would be a fruitful pathway for further research.

Conclusion. Graded type systems and ownership with borrowing are both ways of carefully and precisely managing the usage of data built upon the foundation of linear logic, but these two approaches have developed through very different pathways on the road to being incorporated in modern-day programming languages. In this work, we have developed a core calculus that captures many of the key concepts for ownership tracking in a graded setting, connecting these fine-grained substructural notions with prior work on simpler systems such as linear and uniqueness types which sit closer to the theory. This has allowed us to not only better understand the relationship between these disparate approaches to resourceful reasoning but also to examine how they interact.

By developing a framework through which both ownership properties and precise grades for reasoning about data can be tracked within the setting of the Granule language, we demonstrate that careful management of both resource and memory usage are not only compatible but complementary in a functional context. This paper represents a piece of two larger puzzles—one aiming to develop an in-depth theoretical understanding of Rust's comprehensive approach to memory management, and one aiming to expand the range of properties about programs that can be represented explicitly through graded types—and we look forward to seeing more ideas being shared (or, indeed, borrowed) across the boundary between these two closely related worlds in the future.

ACKNOWLEDGEMENTS

Thanks to comments and feedback from the anonymous reviewers as well as input from Michael Vollmer. This work was supported in part by an EPSRC Doctoral Training Award (Marshall) and EPSRC grant EP/T013516/1 (Orchard). Orchard is also supported in part by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Sciences program.

DATA-AVAILABILITY STATEMENT

All of the examples presented throughout this work, as well as an installation of Granule with the ownership and borrowing extensions that we describe, are included in the artefact for this paper. Full definitions as well as proofs of all theorems are presented in the appendices. Both the artefact and the full appendices are available on Zenodo [Marshall and Orchard 2024a,b].

REFERENCES

Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. *Proc. ACM Program. Lang.* 4, ICFP (2020), 90:1–90:28. https://doi.org/10.1145/3408972

Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proc. ACM Program. Lang.* 7, ICFP (2023), 920–954. https://doi.org/10.1145/3607862

Paulo Sérgio Almeida. 1997. Balloon Types: Controlling Sharing of State in Data Types. In *ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11*. Springer, 32–59. https://doi.org/10.1007/BFb0053373

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 56–65. https://doi.org/10.1145/3209108.3209189

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The Design and Formalization of Mezzo, a Permission-Based Programming Language. *ACM Trans. Program. Lang. Syst.* 38, 4, Article 14 (Aug 2016), 94 pages. https://doi.org/10.1145/2837022

Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 139 (Oct 2021), 32 pages. https://doi.org/10.1145/3485516

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science* 6, 6 (1996), 579–612. https://doi.org/10.1017/S0960129500070109

P Nick Benton. 1995. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models. In *Computer Science Logic: 8th Workshop, CSL'94 Kazimierz, Poland, September 25–30, 1994 Selected Papers 8*. Springer, 121–135. https://doi.org/10.1007/BFb0022251

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29. https://doi.org/10.1145/3158093

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2023a. Multi-Graded Featherweight Java. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:27. https://doi.org/10.4230/LIPIcs.ECOOP.2023.3

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2023b. Resource-Aware Soundness for Big-Step Semantics. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 267 (Oct 2023), 29 pages. https://doi.org/10.1145/3622843

Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, and Marco Servetto. 2022. Coeffects for Sharing and Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 156 (Oct 2022), 29 pages. https://doi.org/10.1145/3563319

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 259–270. https://doi.org/10.1145/1040305.1040327

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis: 10th International Symposium, SAS 2003 San Diego, CA, USA, June 11–13, 2003 Proceedings*. Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. , 9:1–9:26 pages. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *ESOP*, Vol. 8410. Springer, 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

Pritam Choudhury, Harley Eades III, Richard A Eisenberg, and Stephanie Weirich. 2021. A Graded Dependent Type System with a Usage-Aware Semantics. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434331

Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness is Unique Enough. In *ECOOP 2003–Object-Oriented Programming: 17th European Conference, Darmstadt, Germany, July 21-25, 2003. Proceedings 17*. Springer, 176–200. https://doi.org/10.1007/978-3-540-45070-2_9

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

Edsko de Vries. 2013. Modelling Unique and Affine Typing Using Polymorphism. In *Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday on The Beauty of Functional Code - Volume 8106*. Springer-Verlag, Berlin, Heidelberg, 181–192. https://doi.org/10.1007/978-3-642-40355-2{_}13

Edsko de Vries, Adrian Francalanza, and Matthew Hennessy. 2009. Uniqueness Typing for Resource Management in Message-Passing Concurrency. In *Proceedings First International Workshop on Linearity, LINEARITY 2009, Coimbra, Portugal, 12th September 2009 (EPTCS, Vol. 22)*, Mário Florido and Ian Mackie (Eds.). 26–37. https://doi.org/10.4204/EPTCS.22.3

Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. 2008. Uniqueness Typing Simplified. In *Implementation and Application of Functional Languages*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 201–218. https://doi.org/10.1007/978-3-540-85373-2_12

Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 7–21. https://doi.org/10.1007/11693024_2

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining Effects and Coeffects via Grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 476–489. https://doi.org/10.1145/2951913.2951939

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

Jean-Yves Girard, Andre Scedrov, and Philip J Scott. 1992. Bounded Linear Logic: A Modular Approach to Polynomial-Time Computability. *Theoretical Computer Science* 97, 1 (1992), 1–66. https://doi.org/10.1016/0304-3975(92)90386-T

Peter Hanukaev and Harley Eades III. 2023. Combining Dependency, Grades, and Adjoint Logic. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe 2023, Seattle, WA, USA, 4 September 2023*, Youyou Cong and Pierre-Évariste Dagand (Eds.). ACM, 58–70. https://doi.org/10.1145/3609027.3609408

Dana Harrington. 2006. Uniqueness Logic. *Theoretical Computer Science* 354, 1 (2006), 24–41. https://doi.org/10.1016/j.tcs.2005.11.006

Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with Safe Manual Memory-Management in Cyclone. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/1029873.1029883

John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. *SIGPLAN Not.* 26, 11 (Nov 1991), 271–285. https://doi.org/10.1145/118014.117975

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

Jack Hughes, Michael Vollmer, and Dominic Orchard. 2021b. Deriving Distributive Laws for Graded Linear Types. *Electronic Proceedings in Theoretical Computer Science* 353 (Dec 2021), 109–131. https://doi.org/10.4204/eptcs.353.6

Harley Eades III and Dominic Orchard. 2020. Grading Adjoint Logic. arXiv:2006.08854 [cs.LO]

Pierre Jouvelot and David Gifford. 1991. Algebraic Reconstruction of Types and Effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Orlando, Florida, USA) *(POPL '91)*. Association for Computing Machinery, New York, NY, USA, 303–310. https://doi.org/10.1145/99583.99623

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32. https://doi.org/10.1145/3371109

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34. https://doi.org/10.1145/3158154

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. In *ICFP'23*. ACM SIGPLAN. https://doi.org/10.1145/3607840 preprint.

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

Dhruv C Makwana and Neel Krishnaswami. 2019. NumLin: Linear Types for Linear Algebra. (2019). https://doi.org/10.4230/LIPICS.ECOOP.2019.14

Daniel Marshall and Dominic Orchard. 2022a. Graded Modal Types for Integrity and Confidentiality. In *17th Workshop on Programming Languages and Analysis for Security (PLAS 2022).* arXiv:2309.04324 [cs.PL]

Daniel Marshall and Dominic Orchard. 2022b. How to Take the Inverse of a Type. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:27. https://doi.org/10.4230/LIPIcs.ECOOP.2022.5

Daniel Marshall and Dominic Orchard. 2022c. Replicate, Reuse, Repeat: Capturing Non-Linear Communication via Session Types and Graded Modal Types. *Electronic Proceedings in Theoretical Computer Science* 356 (March 2022), 1–11. https://doi.org/10.4204/eptcs.356.1

Daniel Marshall and Dominic Orchard. 2024a. Functional Ownership through Fractional Uniqueness (Appendix). https://doi.org/10.5281/zenodo.10799026

Daniel Marshall and Dominic Orchard. 2024b. Functional Ownership through Fractional Uniqueness (Artefact). https://doi.org/10.5281/zenodo.10797791

Daniel Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings.* Springer International Publishing Cham, 346–375. https://doi.org/10.1007/978-3-030-99336-8_13

Conor McBride. 2001. The Derivative of a Regular Type is its Type of One-Hole Contexts. *Unpublished manuscript* (2001), 74–88. http://strictlypositive.org/diff.pdf

Conor McBride. 2016. I Got Plenty o' Nuttin'. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday* (2016), 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. *Programming Languages and Systems* 12648 (2021), 462. https://doi.org/10.1007/978-3-030-72019-3_17

Alan Mycroft and Janina Voigt. 2013. Notions of Aliasing and Ownership. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 59–83. https://doi.org/10.1007/978-3-642-36946-9_4

Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: Uniqueness Types and Certified Compilation. *J. Funct. Program.* (2021). https://doi.org/10.1017/S095679682100023X

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–30. https://doi.org/10.1145/3342537

David J Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 1 (2021), 1–73. https://doi.org/10.1145/3443420

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 123–135. https://doi.org/10.1145/2628136.2628160

Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. 2020. Kindly Bent to Free Us. *Proc. ACM Program. Lang.* 4, ICFP, Article 103 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408985

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 158–174. https://doi.org/10.1145/3453483.3454036

Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. 1994. Guaranteeing Safe Destructive Updates Through a Type System with Uniqueness Information for Graphs. In *Graph Transformations in Computer Science*, Hans Jürgen Schneider and Hartmut Ehrig (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 358–379. https://doi.org/10.1007/3-540-57787-4_23

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *The Journal of Logic Programming* 29, 1 (1996), 17–64. https://doi.org/10.1016/S0743-1066(96)00068-4 High-Performance Implementations of Logic Programming Systems.

Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. 2022. Linearly Qualified Types: Generic Inference for Capabilities and Uniqueness. *Proc. ACM Program. Lang.* 6, ICFP, Article 95 (Aug 2022), 28 pages. https://doi.org/10.1145/3547626

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17, 3 (Sept. 2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. *SIGPLAN Not.* 46, 1 (Jan. 2011), 447–458. https://doi.org/10.1145/1925844.1926436

Victoria Vollmer, Daniel Marshall, Harley Eades III, and Dominic Orchard. 2024. A Mixed Linear and Graded Logic: Proofs, Terms, and Models. arXiv:2401.17199 [cs.LO]

Philip Wadler. 1990. Linear Types can Change the World!. In *Programming Concepts and Methods*, Vol. 3. Citeseer, 5. https://homepages.inf.ed.ac.uk/wadler/papers/linear/linear.dvi

Philip Wadler. 1993. A Syntax for Linear Logic. In *Mathematical Foundations of Programming Semantics, 9th International Conference, New Orleans, LA, USA, April 7-10, 1993, Proceedings*. 513–529. https://doi.org/10.1007/3-540-58027-1_24

David Walker, Karl Crary, and Greg Morrisett. 2000. Typed Memory Management via Static Capabilities. *ACM Trans. Program. Lang. Syst.* 22, 4 (Jul 2000), 701–771. https://doi.org/10.1145/363911.363923

Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 14 (Jan 2024), 32 pages. https://doi.org/10.1145/3632856

Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2021. Oxide: The Essence of Rust. arXiv:1903.00982 [cs.PL]

James Wood and Robert Atkey. 2022. A Framework for Substructural Type Systems. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 376–402. https://doi.org/10.1007/978-3-030-99336-8_14

Dengping Zhu and Hongwei Xi. 2005. Safe Programming with Pointers Through Stateful Views. In *Practical Aspects of Declarative Languages*, Manuel V. Hermenegildo and Daniel Cabeza (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 83–97. https://doi.org/10.1007/978-3-540-30557-6_8

Received 21-OCT-2023; accepted 2024-02-24