Linearity and Uniqueness: An Entente Cordiale

Daniel Marshall[1], Michael Vollmer[1], and Dominic Orchard[1]

University of Kent, Canterbury, UK

Abstract. Substructural type systems, which restrict the use of weakening and contraction rules from intuitionistic logic, are growing in popularity because they allow for a resourceful interpretation of data which can be used to rule out various software bugs. Indeed, substructurality is finally taking hold in modern programming; Haskell now has linear types roughly based on Girard's linear logic but integrated via graded function arrows, Clean has uniqueness types designed to ensure that values have at most a single reference to them, and Rust has an intricate ownership system for guaranteeing memory safety. But despite this broad range of resourceful type systems, there is comparatively little understanding of their relative strengths and weaknesses or whether their underlying frameworks can be unified. In particular, are linearity and uniqueness essentially the same thing, are they somehow 'dual' to one another, or is the truth somewhere in between? This paper formalises the relationship between these two well-studied but rarely contrasted ideas, building on two distinct bodies of literature, showing that it is possible and advantageous to have both linear and unique types in the same type system. We study the guarantees of the resulting system and provide a practical implementation in the graded modal setting of the Granule language, adding a third kind of modality alongside coeffect and effect modalities. We then demonstrate via a benchmark that our implementation benefits from expected efficiency gains enabled by adding uniqueness to a language that already has a linear basis.

Keywords: linear types · uniqueness types · substructural logic

1 Introduction

Reading the literature one might get the impression that linear types and uniqueness types are two names for the same concept, perhaps separated only by some minor implementational details. Indeed, the section on substructural type systems in *Advanced Topics in Types and Programming Languages* [54] describes uniqueness types as "*a variant of linear types*". This framing is corroborated by various papers which, for example, make reference to "*a form of linearity (called uniqueness)*" [29] or other such statements of similarity or equality.

But reading a different set of papers might give the contrasting impression that linearity and uniqueness are in some sense "dual" to one another, and so have importantly different behaviour for at least some applications. A recent publication on linear types for Haskell [6] describes the two concepts as being

"*at their core, dual*", though this is later clarified to be only a "*weak duality*". The impression that these systems do not work in the same way is backed up by much of the theoretical work on uniqueness types, with one paper stating that "*although both linear logic and uniqueness typing are substructural logics, there are important differences*" [48] – closely followed by a tantalising mention of the fact that "*some systems based on linear logic are much closer to uniqueness typing than to linear logic*".

It is clear, at least, that both linear types and uniqueness types are *substructural* type systems, in that they both restrict the application of the structural rules (contraction, weakening, and sometimes exchange) found in type systems that are the Curry-Howard counterparts to regular intuitionistic logic. This captures the well-known maxim that "*not all things in life are free*" [53]; many kinds of data behave *resourcefully*, and are subject to constraints on their usage. Sensitive data should not be infinitely duplicated and passed around freely, file handles should not be arbitarily discarded without being properly closed, and communication channels should not be unconstrained and able to be used without restriction, to name a few!

Thanks to these clear benefits, notions of substructurality are slowly but surely making their way into the programming ecosystem, with languages such as Haskell [6], Idris [9], Clean [41], Rust [23], ATS [56], and Granule [32] all having type systems that behave substructurally in some way. What is not clear, however, is what exactly the relationship is between these varying systems. One important instance of this comes when we try to relate linearity and uniqueness. Linear types, though they themselves come in various forms, are in general based on the linear logic of Girard [15], and in the strictest sense they treat values as resources which must be used exactly once and never again. On the other hand, uniqueness types are named as such because they aim to ensure that values are guaranteed to have at most one reference to them [41, 35, 48, 47, 40, 42], with a view towards allowing them to be safely updated in-place. Do these two requirements always coincide, or are there cases where they diverge?

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

- In Section 2 we discuss the differing widespread understandings (and misunderstandings) of the relationship between linearity and uniqueness, and draw together aspects of these varying viewpoints in order to properly describe the link between these concepts.
- In Section 3 we formalise these notions by developing a unified calculus and type system that incorporates linear, unique, and Cartesian (unrestricted or non-unique/non-linear) types all at once, building on the linear λ-calculus. We give an operational model via a heap semantics which allows us to prove various key operational guarantees for both linearity and uniqueness.

– In Section 4, as a proof of concept, we implement uniqueness types into the language Granule which already has a linear basis, introducing a third flavour of modality alongside the graded comonadic (coeffectful) and graded monadic (effectful) modalities already present in the language. The implementation enables the classic primary use of uniqueness: access to safe in-place update in a functional language without reverting to a monadic style.
– In Section 4.2 we confirm the performance benefits of uniqueness types by benchmarking the performance of arrays which allow for in-place update. We do this by generating impure Haskell code from our Granule implementation, and demonstrating that further efficiency can be gained via adding uniqueness types even when your language is already linear at its core.

2 Key Ideas

It is clear by this point that linear types and uniqueness types are both obtained by restricting the substructural rules of intuitionistic logic, and that what remains unclear is the exact relationship between these two concepts. This section will discuss in more depth two widespread understandings of this relationship, both of which are accurate in some respects but fail to capture some important similarities or differences; we will then combine aspects of both of these viewpoints to properly relate linearity and uniqueness once and for all.

Misconception 1. *Linearity and uniqueness are two different words for the same concept / linearity and uniqueness are essentially the same.*

Perhaps the most well known substructural types are **linear** types, which have been studied for decades in the literature [49, 54] as the Curry-Howard counterpart of linear logic [15]. Several languages have implemented linear type systems over the years, including ATS [56], Alms [46] and Quill [28], and they are steadily making their way into the mainstream via extensions to languages like Haskell [6]. Examples of linearity in this paper will focus on the functional language Granule [32], since values in Granule are linear by default making the examples less complex, and also because Granule will later be the foundation upon which we build our unified calculus.

Strictly, linear types treat values as resources which must be used once and then never again. For instance, we can type the identity function, since it binds a single variable and then uses it, but the K combinator (which discards one of its arguments) is not linearly typed. Thus linearity is a claim about the consumption of a resource: a linear type is a contract, which says that we must consume a value that we are given exactly once. Consider the following classic example of a function which cannot be represented using linear types, assuming an interface where `eat : Cake → Happy` and `have : Cake → Cake`:

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

Ill-typed Granule

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

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

One of the most familiar languages featuring uniqueness types is Clean [41], which uses uniqueness for mutable state and input/output, in contrast to languages such as Haskell which use monads for similar purposes. We shall use Clean for our uniqueness examples for the moment, before we introduce our own implementation uniqueness in Section 4. Consider this function in Clean:

```
1 impossible :: *Coffee -> (*Awake, *Coffee)
2 impossible coffee = (drink coffee, have coffee)
```

Ill-typed Clean

Here we use coffee instead of cake, to differentiate our unique values from our linear ones, but notice that this function has exactly the same structure as the previous example in Granule. Here the annotation $*$ denotes a unique type, since in Clean unrestricted values are the default. Similarly to before, when presented with this function Clean offers a uniqueness error; the value of type `*Coffee` passed to the function must be duplicated, and so we can no longer guarantee there is only one reference to it. So far, it seems as though the concepts of linearity and uniqueness are very similar after all as is regularly claimed.[1]

However neither of these examples uses any *unrestricted* values; we only see values that are linearly typed or uniquely typed. In fact, in a setting where all values must be linear, we can also guarantee that every value is unique, and vice versa! Intuitively, if it is never possible to duplicate a value, then it is not possible for said value to ever have multiple references. It is when we offer the capability to apply or remove restrictions on the substructural rules that differences between linearity and uniqueness begin to arise, as we will see shortly.

Much of the classic literature on linear types makes mention of the idea that linearity can be used for tracking whether a value has only one reference, though we know by now that this more accurately describes *uniqueness*; indeed, one of the oldest such papers by Wadler, which has been (rightly) hugely influential, states that "*values of linear type have exactly one reference to them, and so require no garbage collection*" [49, p.2]. However, the linear type system discussed in this paper and other similar systems [4, 17] crucially separate values into two completely distinct and disconnected linear and non-linear worlds. In this context, a linear value can never have been used in an unrestricted manner and

[1] Though the aphorism "you can't have your coffee and drink it too" is somewhat less familiar!

thus also obeys the conditions required for uniqueness. Therefore, it is correct to say that a value of linear type has exactly one reference in such a system.

This issue is further discussed in a later article by Wadler [50], though uniqueness types had yet to be invented and so the concept is never referred to by this name. Linear types based on linear logic are defined in Section 3 of said article, for which linearity behaves as we understand it: a value being of linear type guarantees that it will not be duplicated or discarded in the future but makes no claims about what happened to it in the past. As the paper states, "*dereliction means we cannot guarantee a priori that a variable of linear type has exactly one pointer to it. But if we know this by other means, then linearity guarantees that the pointer will not be duplicated or discarded.*" Indeed, later in Section 7, Wadler defines *steadfast* types where dereliction and promotion are restricted as in the earlier paper [49] to recover the uniqueness guarantee in addition to the linearity restriction. The concept of steadfastness coincides with the notion of "necessarily unique" used in languages such as Clean, where a necessarily unique value is one that is unique and also can never be made non-unique [41].

Since never being able to duplicate or discard any value is a somewhat restrictive view of data, preventing many valid uses of various kinds of information, linear type systems generally provide an operator akin to the ! modality from linear logic (also called the *exponential* modality), which allows for the representation of non-linear or unrestricted values. A Granule example making use of this modality would work as follows:

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

Granule

This is no longer ill-typed; we can think of the value of type `!Cake` as representing an infinite amount of cake, which is made available once we eliminate the modality (via the `let`) to get an unrestricted variable `cake`. Note, importantly, that if we have an unrestricted value we can produce a linear value from it, so we can impose the restriction of linearity whenever we like, but it is not possible to produce an unrestricted value from a linear one.

This notion means that linear types are useful for representing resources such as file handles, as in the following example.

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

Granule

Here, we open a file handle, read two characters from it, and then close it. The linearity of the handle `h` ensures that once we have created it, we must close

it properly, and also that we cannot duplicate it along the way. But linearity is less useful in other circumstances. As an example, we consider the case of mutable arrays. Discarding an array will not cause any problems[2], so a linear array would be too restrictive and not allow for some valid use cases. But in order to be able to mutate an array we need to be able to guarantee that no other references to it exist, and in this sense linearity is not strong enough; any linear value could have previously been a non-linear one that was duplicated any number of times before being specialised to a linear type. For representing mutable arrays, we are better served by considering uniqueness types.

Uniqueness behaves slightly differently in that if we have an unrestricted value, we certainly cannot produce a unique one from it, as this would violate the guarantee of uniqueness; we cannot claim that a value has only one reference to it when it could have been duplicated and manipulated elsewhere. But conversely, if we have a unique value, there is no harm in discarding this guarantee and producing an unrestricted one; a non-unique value does not need to make any promises about how many references may exist. Thus, in Clean we can write the following program:

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

Clean

Here, we know that the value of type `*Coffee` passed into the function is unique, but in order for the function not to be ill-typed we can no longer claim that it is unique once it reaches the other side, as it has been duplicated along the way. Notice that the information is flowing in the opposite direction for uniqueness as opposed to linearity; the `possible` function in Clean would be ill-typed in Granule, and the `possible` function from Granule would be ill-typed in Clean (if we could swap the role of linearity and uniqueness). This directionality allows us to represent mutable arrays much more easily with uniqueness. For example, the following destructively fills a real-valued array:

```
fill :: *{Real} Int -> *{Real}
fill a1 0 = a1
fill a1 i
    # f = toReal i
    # a2 = {a1 & [i - 1] = f}
    = fill a2 (i - 1)
```

Clean

Here, we take in a unique array of floating point numbers and some unrestricted integer value, and fill the first cells of the array with the numbers up

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

to that value. Here we know that it is safe to write to the array because it is unique, so no other references to it can exist elsewhere; once we are finished with the array later on, however, it is fine to discard it, as with an array in any other functional programming language, which would not be possible if our array was linearly typed. This however means that uniqueness types are not appropriate for the earlier example of file handles – we cannot ensure that a unique file handle is closed, as it can be discarded at any time.

These complementary, but distinct, use cases make it clear that it would be valuable to have both linear and unique values together in a single programming language, but this has previously not been possible. Our main contribution is a core calculus that allows linearity and uniqueness to coexist and interact, demonstrated also via an implementation in the Granule language. Next we consider though how to formally describe how linearity and uniqueness differ.

Misconception 2. *Linearity and uniqueness are "dual" to one another.*

It is common folklore in the literature to say that linearity and uniqueness are somehow "dual" to one another, which does seem to line up with this framing, but rigorous versions of this statement are found much more rarely. The earliest formalisation of uniqueness comes from Harrington's paper on 'uniqueness logic' [19], and it is this formalisation which forms a foundation for much of the following. Harrington constructs a logic which is on the surface much like linear logic, but instead of the ! modality for non-linearity it includes a new $\circ$ modality for non-uniqueness which differs to non-linearity in its introduction rule.

In linear logic, ! in linear logic acts as a comonad, such that the introduction of ! on the right of a sequent means that all formulae on the left of a sequent must also have ! applied, whilst introduction of ! on the left is unrestricted:

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

(also known as *storage* and *dereliction* respectively [16]). In contrast, the non-uniqueness modality $\circ$ of Harrington acts as a monad, meaning that introduction of $\circ$ on the right is unrestricted but introduction of $\circ$ on the left of a sequent means that all formulae on the right of the sequent must also have $\circ$ applied:

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

Non-uniqueness then has the following structural rules for contraction and weakening which are conspicuously identical to those for non-linearity !:

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

One might be tempted to think that because the introduction rules for $\circ$ behave dually to those of !, the modalities are simply dual to one another, and uniqueness is dual to linearity as we hoped for. But since the contraction and weakening

rules for $\circ$ are the same as those for !, this is not quite the case; $\circ$ behaves dually to ! in some ways but not in others. Formally, $\circ$ is a monad while ! is a comonad, but both are comonoidal, whereas the dual modality to ! (called ? in linear logic) would necessarily be monoidal.

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

Another idea we can make more precise at this stage is the notion that linear types provide a restriction on what can be done with a value in the future whilst uniqueness types provide a guarantee about what has been done with a value in the past. The key point to consider here is substitutions which are generated by beta reductions. These look the same whether we are working with linear logic or uniqueness logic, as the rules for functions are identical, but the difference arises when thinking about what it is possible to know about a value in one logic compared to the other.

If we have a linear value, we know that substituting this value forwards into an expression will preserve linearity, as there is no way to transform a linear value into a non-linear one. Conversely, if we have a unique expression then we know that any values substituted in will not affect the uniqueness guarantee, as there is no way to transform a non-unique value into a unique one. This is how we can understand the idea of future vs past: 'future' refers to outgoing substitutions, while 'past' refers to incoming substitutions.

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

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

In other words, internally the non-linear and non-unique modalities are both comonoidal, so they allow for the same behaviour of contraction and weakening for values that are wrapped inside them. But the duality arises upon considering how we can map into and out of these modalities; we can map out of the non-linear modality and retrieve a linear value, but we can never map into it, giving the modality its familiar comonadic structure. Conversely, we can map a unique value into the non-unique modality to allow for substructural behaviour, but we can never map out of it, giving a more monadic flavour.

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

3 The Linear-Cartesian-Unique Calculus

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

- The first reason is pragmatic; we later implement our approach for Granule, which already has linear values as the default. Therefore, including uniqueness as an additional modality in the system will require far fewer changes to the language, since a unique base would most likely require a redesign. Moreover, languages with uniqueness types like Clean generally have non-unique values as their default, with uniqueness having to be specifically annotated; a system with a uniqueness modality will also map more closely onto these languages than one where uniqueness is the basis.
- The second reason is that developing a sound calculus with a unique base is more complex. Consider such a hypothetical calculus with a modality $\circ$ representing unrestricted values and a modality $\bullet$ representing linear values. If we construct a product of linear values $(a^\bullet, b^\bullet)$, then this product is unique (rather than linear), so we can promote to an unrestricted product $(a^\bullet, b^\bullet)^\circ$ and freely duplicate the product, though the values contained within are linear. A linear base avoids this problem (among others) as linear-by-default products means their usage is maximally restricted, so there is no circumventing either a uniqueness guarantee via their construction or a linearity restriction via their duplication.[4]

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

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

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

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

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

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

3.1 Typing

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

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

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

Typing contexts are defined as follows:

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

which are either empty, or contexts extended with a linear assignment $x : A$ or contexts extended with a non-linear assignment denoted $x : [A]$. This separation of assumptions into linear/non-linear assumptions traces back to Terui [44] as a way to guarantee substitution is admissible (avoiding, for example, issues pointed out by Wadler where substitution is not well-typed if care is not taken [51, 52], an issue noted also by Prawitz in 1965 in the context of S4 modal logic [37]).

We introduce the key typing rules inline. Figure 1 collects the full set of rules. The linear-λ calculus core is typed by the following three rules:

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs} \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \textsc{app}$$

In the case of (var), a linear variable is used but the rest of the context must be marked as non-linear, denoted by $[\Gamma]$ which marks all assumptions as non-linear.

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

In the case of APP, the two subterms are typed in different contexts which are then combined via *context addition*.

Definition 2 (Context addition). *For all Γ_1, Γ_2, context addition is defined as follows by ordered cases matching inductively on the structure of Γ_2:*

$$\Gamma_1 + \Gamma_2 = \begin{cases} \Gamma_1 & \Gamma_2 = \emptyset \\ (\Gamma_1 + \Gamma_2'), x : A & \Gamma_2 = \Gamma_2', x : A\ \wedge\ x : A \notin \Gamma_1 \\ (\Gamma_1 + \Gamma_2'), x : [A] & \Gamma_2 = \Gamma_2', x : [A] \end{cases}$$

Context addition is undefined if Γ_1 and Γ_2 overlap in linear assumptions.

The non-linear modality has the following introduction and elimination rules and related "dereliction" rule:

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

The left-most rule captures the idea that a computation t of value A can be used non-linearly, by "promoting" it to $!A$ as long as all its inputs are also non-linear, denoted by $[\Gamma]$ in the context. The middle rule eliminates a non-linear modality (a capability to use an A value non-linearly) by composing it with a variable x which is non-linear in t_2. These rules are accompanied by the "dereliction" rule that says linear variables can be treated as non-linear variables.

So far everything here is well-known from other definitions of linear types. We now move to our key uniqueness modality which has two syntactic constructs: *borrow* and *copy* typed respectively as:

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \&t : !A}\ \text{BORROW} \qquad \frac{\Gamma_1 \vdash t_1 : !A \quad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{copy}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \text{COPY}$$

The borrow rule maps a unique A value to a non-linear value, allowing a guarantee of uniqueness to be forgotten. The copy rule says that a non-linear A can be copied to produce a unique A which is used by t_2; the input is required to be non-linear so that we cannot circumvent a linearity restriction by copying a linear value, and the output is required to be non-unique so that we cannot leverage this temporary uniqueness to smuggle out a value which pretends to be truly unique. These rules in turn are accompanied by the "necessitation" rule that says that values can be unique by default as long as they have no dependencies:

$$\frac{\emptyset \vdash t : A}{[\Gamma] \vdash *t : *A}\ \text{NEC}$$

The borrow and copy rules in this logic suggest a monad-like relationship between the ! and $*$ modalities, with the borrow rule representing the 'return' of the monad and the copy rule likewise acting as the 'bind'. The $*$ modality is not in itself a monad (or indeed, a comonad like !); rather, it acts a functor over which the ! modality becomes a *relative monad* [3]. A relative monad comprises a functor J and an object mapping T along with an operation $\eta : JX \to TX$ and a mapping from $JX \to TY$ arrows to $TX \to TY$ with axioms analogous to the monad axioms. Thus, here J is the uniqueness modality $*$ and T the non-linearity modality !. If one imagines the dual version of this logic where the basis is unique, the hypothetical linearity modality would act as a functor making the non-uniqueness modality into a *relative comonad* [1, 33] in much the same way.

3.2 Equational theory

One way of understanding the meaning of the LCU calculus is to see its equational theory (which is later proved sound against its operational model). The

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \text{VAR} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \text{ABS} \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \text{APP}$$

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

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

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

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \&t : !A}\ \text{BORROW} \qquad \frac{\Gamma_1 \vdash t_1 : !A \quad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{copy}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \text{COPY} \qquad \frac{\emptyset \vdash t : A}{[\Gamma] \vdash *t : *A}\ \text{NEC}$$

Fig. 1: Collected typing rules for LCU calculus

calculus has the standard $\beta\eta$-equalities for the multiplicative linear-λ calculus fragment, which includes the following $\beta\eta$ rules for !:

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

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

along with the following equalities on the uniqueness fragment:

$$\mathsf{copy}\ t\ \mathsf{as}\ x\ \mathsf{in}\ \&x \equiv t \qquad (\text{unitR})$$

$$\mathsf{copy}\ \&t\ \mathsf{as}\ x\ \mathsf{in}\ t' \equiv [t/x]t' \qquad (\text{unitL})$$

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

The first axiom states that copying a non-linear t into a unique value x and immediately borrowing it to be non-linear is equivalent to just t. The second axiom states that borrowing a unique t and copying it to a unique x in the scope of t' is the same as just substituting in that t for x. The last axiom gives associativity of copying under the side condition that x is free in t_3. These equations are exactly the relative monad axioms [3].

The typability of these axioms relies on the admissibility of linear and non-linear substitution shown next in Section 3.5 on the metatheory of the calculus.

3.3 Exploiting uniqueness for mutation

A key use for ensuring uniqueness of reference is that it allows mutation to be used safely—the original pun behind Wadler's "Linear Types Can Change the World" [49]. To illustrate this idea, and consider its soundness in the next section, we extend the LCU calculus with a primitive type of arrays:

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

where $\mathbb{Z}$ are integers used for sizes and indices and $\mathbb{F}$ floating-point values. The calculus is also extended with operations for floating-point arrays, typed:

$$\begin{aligned}
\mathsf{newArray} &: \mathbb{Z} \to *(\mathsf{Array}\ \mathbb{F}) \\
\mathsf{readArray} &: *(\mathsf{Array}\ \mathbb{F}) \to \mathbb{Z} \to \mathbb{F} \otimes *(\mathsf{Array}\ \mathbb{F}) \\
\mathsf{writeArray} &: *(\mathsf{Array}\ \mathbb{F}) \to \mathbb{Z} \to \mathbb{F} \to *(\mathsf{Array}\ \mathbb{F}) \\
\mathsf{deleteArray} &: *(\mathsf{Array}\ \mathbb{F}) \to 1
\end{aligned}$$

These operations provide the interface for exploiting unique array references, where writeArray can perform mutation since the type system guarantees that uniquely typed values have not been duplicated in the past (Section 3.5). We ignore out-of-bounds exceptions as this is an orthogonal issue easily solved with indexed types. We elide rules for typing integer and floating terms here.

Our implementation in Section 4 replays these ideas in a practical setting. The next section gives the operational heap model for the calculus, where the semantics of the above with respect to mutation is made concrete.

3.4 Operational Heap Model

We define an operational model for the LCU calculus to make the meaning of uniqueness and linearity more concrete, and to prove that our type system enforces the desired properties. The semantics is call-by-name and resembles a small-step operational semantics but instead uses a notion of *heaps* both to capture the idea of a memory reference to the primitive arrays, as well as to give a way to track resource usage on program variables. We adapt the model of Choudhury et al. [11], which was used to track resource usage in a pure graded type system. Our model applies this idea to a non-graded setting, extended to include reference counting for uniqueness. To prove that linearity and uniqueness are respected (soundness), the heap semantics incorporates some typing information in order to ease the theorem statement and proofs as shown in Section 3.5.

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

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

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

$$\Delta ::= \emptyset \mid \Delta, x : r \quad \text{(usage contexts)} \qquad r ::= 1 \mid \omega \quad \text{(usage/reference counter)}$$

where r is a usage marker that says a variable was used either once (denoted with 1), or used more than once (denoted with ω).

Heaps are defined as follows akin to a context but containing two kinds of 'allocations' for variables x and for array references a (an additional value form):

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

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

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

$$\frac{}{H \vdash t \;\Rightarrow\; H \vdash t \mid \emptyset \mid \emptyset}\ \textsc{refl} \qquad \frac{\begin{array}{c} H \vdash t_1 \rightsquigarrow H' \vdash t_2; \Gamma_1; \Delta_1 \\ H' \vdash t_2 \;\Rightarrow\; H'' \vdash t_3 \mid \Gamma_2 \mid \Delta_2 \end{array}}{H \vdash t_1 \;\Rightarrow\; H'' \vdash t_3 \mid \Gamma_1, \Gamma_2 \mid \Delta_1 + \Delta_2}\ \textsc{ext}$$

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

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

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

Heap model The reduction rules for the heap model are collected in the appendix, but we explain the core rules of the single-reduction relation here. Unlike a normal small-step semantics, variables have a reduction, with two possibilities:

$$\frac{}{H, x{\mapsto_1}t \vdash x \rightsquigarrow H \vdash t; \emptyset; x : 1}\ {\rightsquigarrow}_{\textsc{var1}} \qquad \frac{}{H, x{\mapsto_\omega}t \vdash x \rightsquigarrow H, x{\mapsto_\omega}t \vdash t; \emptyset; x : 1}\ {\rightsquigarrow}_{\textsc{var}\omega}$$

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

β-reduction is then given as follows:

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

Rather than using a substitution, the body term is the result under a heap extended with x assigned to the (typed) argument term t'. Note that this is

given a resource capability of 1 since functions are linear. In the output, we remember that a linear binding has been opened up in the scope of the term. An inductive rule allows an application to reduce on the left:

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

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

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

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

$$\frac{H \vdash t \rightsquigarrow H' \vdash t'; \Gamma; \Delta}{H \vdash \&t \rightsquigarrow H' \vdash \&t'; \Gamma; \Delta}\ \rightsquigarrow_{\&} \qquad \frac{\mathsf{dom}(H) \equiv \mathsf{arrRefs}(v)}{H, H' \vdash \&(*v) \rightsquigarrow ([H]_\omega), H' \vdash !v; \emptyset; \emptyset}\ \rightsquigarrow_{\&*}$$

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

$$H', a \mapsto_1 \mathbf{arr} \vdash \& * a \rightsquigarrow H', a \mapsto_\omega \mathbf{arr} \vdash !a; \emptyset; \emptyset$$

Therefore, borrowing enacts the idea that a reference is no longer unique and may be used many times (and hence now is a non-linear value). Copying then has three rules; a congruence (elided), a rule which forces evaluation under the non-linear modality and then a reduction to enact the copy as a unique value and binding:

$$\frac{H \vdash t \rightsquigarrow H' \vdash t'; \Gamma; \Delta}{H \vdash \mathsf{copy}\ !t\ \mathsf{as}\ x\ \mathsf{in}\ t_2 \rightsquigarrow H' \vdash \mathsf{copy}\ !t'\ \mathsf{as}\ x\ \mathsf{in}\ t_2; \Gamma; \Delta}\ \rightsquigarrow_{\mathsf{copy!}}$$

$$\frac{\Gamma \vdash v : A \qquad \mathsf{dom}(H') \equiv \mathsf{arrRefs}(v) \qquad (H'', \theta) \equiv \mathsf{copy}(H')}{H, H' \vdash \mathsf{copy}\ !v\ \mathsf{as}\ x\ \mathsf{in}\ t_2 \rightsquigarrow H, H', H'', x \mapsto_1 (\Gamma \vdash *(\theta(v)) : *A) \vdash t_2; x : *A; \emptyset}\ \rightsquigarrow_{\mathsf{copy}\beta}$$

Thus, copying allows a non-linear term to be bound uniquely, but crucially forces evaluation of said term to a value so that it is dependency free and cannot be used in the resulting body t_2 in a way that would violate uniqueness.

Lastly, the semantics of the four array primitives use an array representation on the heap where **arr** is some array object and $\mathbf{arr}[i] = v$ indicates that the i^{th}

element is bound to the value v:

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

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

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

$$\frac{}{H, a{\mapsto_r}\mathbf{arr} \vdash \mathsf{deleteArray}\ a \rightsquigarrow H \vdash \mathsf{unit}; \emptyset; \emptyset}\ {\rightsquigarrow}_{\mathsf{deleteArray}}$$

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

3.5 Metatheory

The heap model allows us to establish the key properties of well-typed programs respecting linearity and uniqueness restrictions.

We first establish when a heap is *compatible* with a typing context:

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

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

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

Thus, a heap compatible with $\Gamma_1, x : A$ contains either an assignment for x marked with r which can be either 1 for linear use or ω representing the idea that non-linear values can flow to linear uses, as captured by dereliction (DER typing rule). However, a non-linear assumption must have a heap assignment marked with ω (CompatExt!), where the dependencies of the assigned term t must be all non-linear in the remaining compatibility judgment on the rest of the heap. 4 From a heap we can also extract a usage information context, and likewise from a typing context. This is useful for focusing on just resource usage information in the forthcoming results:

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

$$\overline{\emptyset} = \emptyset \qquad \overline{(\Gamma, x : [A])} = \overline{\Gamma}, x : \omega \qquad \overline{(\Gamma, x : A)} = \overline{\Gamma}, x : 1$$

$$\overline{\emptyset} = \emptyset \qquad \overline{(H, x{\mapsto_r}(\Gamma \vdash t : A))} = \overline{H}, x : r \qquad \overline{(H, x{\mapsto_r} t)} = \overline{H}$$

We now give the two main theorems about our calculus which give us the properties that linearity is respected (called *conservation*, Theorem 1) and that uniqueness is respected (Theorem 2).

Theorem 1 (Conservation). *For a well-typed term $\Gamma \vdash t : A$ and all Γ_0 and H such that $H \bowtie (\Gamma_0 + \Gamma)$ and a reduction $H \vdash t \rightsquigarrow H' \vdash t'; \Gamma_1; \Delta$ then:*

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

The first two conjuncts gives regular type preservation. The last conjunct expresses the core of conservation: that resource usage accrued in this reduction, given by Δ plus remaining resources given in the heap H' are approximated by the original resources given in the heap H plus the specification of the resources for any new heap assignments Γ_1.

The context Γ_0 expresses that the heap contains other bindings to those implied by Γ which is key to the inductive proof of this result.

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

Theorem 2 (Uniqueness). *For a well-typed term $\Gamma \vdash t : *A$ and all Γ_0 and H such that $H \bowtie (\Gamma_0 + \Gamma)$ and given a multi-reduction to a value $H \vdash t \Rightarrow H' \vdash *v \mid \Gamma' \mid \Delta$ then for all $a \in \mathsf{arrRefs}(v)$ (array references in v) we have:*

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

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

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

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

A sub-result of conservation is type preservation which is complemented by a separate progress result in Lemma 1 to give syntactic type safety:

Lemma 1 (Progress). *Given $\Gamma \vdash t : A$ then t is either a value v, given by the sub grammar of terms:*

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

or given $H \bowtie \Gamma_0 + \Gamma$ *then there exists heap* H'*, term* t'*, usage context* Δ*, and context* Γ' *such that* $H \vdash t \rightsquigarrow H' \vdash t'; \Gamma'; \Delta$.

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

Theorem 3 (Soundness wrt. equational theory). *For all* t_1, t_2 *such that* $\Gamma \vdash t_1 : A$ *and* $\Gamma \vdash t_2 : A$ *and* $t_1 \equiv t_2$ *and given* H *such that* $H \bowtie \Gamma$ *then there exists a value (irreducible term)* v *and* $\Gamma_1, \Gamma_2, \Delta_1, \Delta_2$ *such that there are full-*β *reductions to the same value*

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

Lastly, we conclude with some admissibility theorems that show the coherence of our approach.

Lemma 2 (Weakening is admissible). *If* $\Gamma \vdash t : A$ *then* $\Gamma, [\Gamma'] \vdash t : A$.

Lemma 3 (Linear substitution). *If* $\Gamma' \vdash t' : S$ *and* $\Gamma, s : S \vdash t : T$ *then* $\Gamma' + \Gamma \vdash [t'/s]t : T$.

Lemma 4 (Non-linear substitution). *If* $[\Gamma'] \vdash t' : S$ *and* $\Gamma, s : [S] \vdash t : T$ *then* $[\Gamma'] + \Gamma \vdash [t'/s]t : T$.

4 Implementation

4.1 Frontend

The implementation of uniqueness types in Granule follows much the same pattern as the logic defined earlier. Granule already possesses a *semiring graded necessity modality*, where for a pre-ordered semiring $(\mathcal{R}, *, 1, +, 0, \sqsubseteq)$, there is a family of types $\{\Box A_r\}_{r \in \mathcal{R}}$. We represent the ! from linear logic (and our extended calculus) via the pre-ordered semiring $\{0, 1, \omega\}$ (none-one-tons) with $!A = \Box A_\omega$.[5]

The semiring is defined with $r + s = r$ if $s = 0$, $r + s = s$ if $r = 0$ and otherwise ω, and $r * 0 = 0 * r = 0$, $r * \omega = \omega * r = \omega$ (for $r \neq 0$), and $r * 1 = 1 * r = r$ with ordering $0 \sqsubseteq \omega$ and $1 \sqsubseteq \omega$. This semiring allows us to represent both linear and non-linear use: variables graded with 1 must be used linearly, with 0 must be discarded, and a grade of ω permits unconstrained use à la linear logic's !.

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

As in LCU, uniqueness is a new kind of modality, which we call $*$ to match the calculus (and so that the syntax of programs involving uniqueness types will be familiar to users of Clean). The uniqueness modality wraps a value that

[5] It may not seem obvious that such a graded modality does exactly represent the behaviour of linear logic's !, and in fact capturing the precise behaviour of ! does require some additional structure on the semiring which is present in the latest version of Granule; this is discussed further in [21].

behaves in a 'linear' fashion (and so cannot be duplicated or discarded), with the key difference being that ! does not act as a comonad over values under a $*$ as it does with linear values; we provide primitive functions which allow ! to act instead as a relative monad. The primitives have the following type signatures:

```
uniqueReturn : ∀ {a : Type} . *a → !a                  -- borrow
uniqueBind : ∀ {a b : Type} . (*a → !b) → !a → !b      -- copy
```
Granule

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

We provide syntactic sugar for both of these primitives for convenience, with syntax designed to evoke the rules from the LCU calculus; `&x` is equivalent to writing `uniqueReturn x`, while `clone t1 as x in t2` is equivalent to writing `uniqueBind (λx → t2) t1`.[6] A simple example of uniqueness types in action is given below, to demonstrate the idea.

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

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

We also provide a library for arrays of floating point numbers in Granule, matching with the interface for arrays of floats that was introduced as an extension to the LCU calculus in Section 3.3. The type signatures for the primitives we provide for working with arrays of floats are as follows.

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

Note that it is possible for `writeFloatArray` to update an array destructively in place since we have a guarantee that no other references exist to the array which has been passed in. In the next section we will use this set of primitives to evaluate the performance of our implementation, by measuring the performance gains from allowing for in-place updates in this fashion.

[6] In the implementation we use 'clone' rather than 'copy', as the name 'copy' is often used elsewhere in Granule, e.g. for the non-linear function which duplicates its input.

Figure 2 illustrates the relationship between uniqueness, non-linearity, linearity and other common forms of substructural typing in the resulting system.

4.2 Compilation and Evaluation

As part of our implementation of uniqueness types in Granule, as described in Section 4.1, we also implemented a simple compiler that translates programs into Haskell. This compiler preserves the value types, but erases all of Granule's substructural types (linear, unique, graded, etc.). As a result, we can take advantage of both Granule's flexible type system and Haskell's libraries and optimizing compiler. For this paper, all performance results were measured by compiling Granule programs to Haskell, and compiling the resulting Haskell with GHC 9.0.1. The measurements were collected on an ordinary Macbook with a 2 GHz quad-core Intel i5 and 16 GB of RAM.

As mentioned in Section 1, one motivation for using uniqueness types is to do the kind of in-place mutation necessary for efficient programming with arrays. As a sanity check, we evaluated our implementation using an array processing benchmark. The benchmark recursively allocates and sums up lists of arrays of various sizes, with the goal of demonstrating the benefits of uniqueness types for arrays in functional programming. Each iteration of the benchmark allocates a list of a thousand arrays, populates the arrays with values, then traverses the list to sum them up. We prepared two versions of this benchmark: one with functional in-place updates and manual (safe) deletion of unique arrays, and one with non-unique, immutable, garbage collected arrays and updates via copying. The overall performance of these two benchmarks is shown in Figure 3a, with lower bars/numbers representing better performance. The results, while not surprising, do confirm that array-handling is generally more efficient when in-place mutation is allowed. Additionally, in Figure 3b, we compared the time spent in garbage collection between the two versions of the benchmark. Because our implementation allocates unique data outside of GHC's heap, and uniqueness types allow programmers to directly de-allocate objects in memory, the unique version of the benchmark spends significantly less time in garbage collection. For this benchmark, the unique arrays are outside of the garbage collected heap and

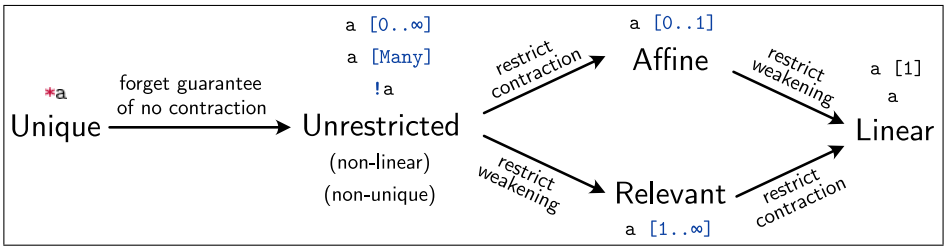

Fig. 2: Relationship between various flavours of substructural type, and demonstrating how they can all be represented using Granule's expressive modalities.

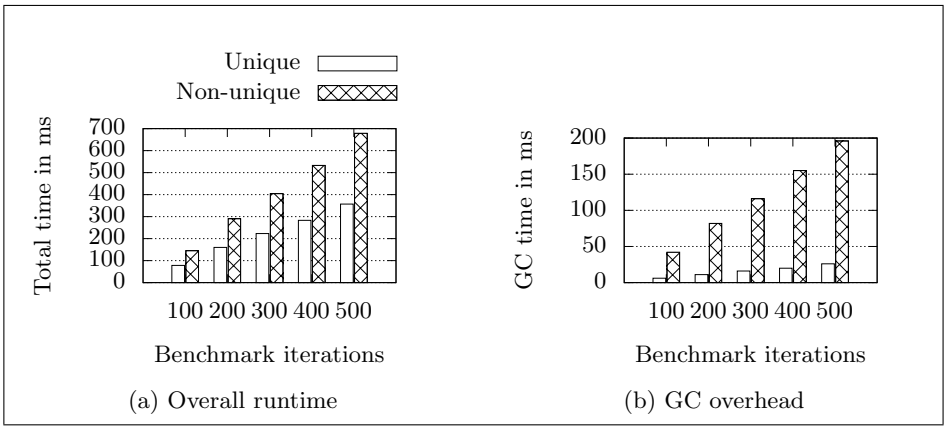

Fig. 3: Comparing performance of array programming on mutable and immutable arrays in Granule. Lower is better.

directly de-allocated, while other incidental objects (closures, lists, and so on) are still handled by the garbage collector.

Of course, this is a somewhat contrived benchmark. Real-world Haskell libraries, for example, typically provide functional high-level interfaces for array manipulation while using *unsafe* code to mutate arrays internally. The popular `vector` library[7] is one example, and `repa` [24] is another. Additionally, there is significant prior research on improving the efficiency of functional programs operating on arrays (for example, using combinators like `map` and `fold` in combination with aggressive fusion [12, 26, 24]), which we will not dwell on here. The main point is that, at some point in the process, arrays must be mutated. Rather than having this happen in unsafe code blocks, or in external C or Fortran code, uniqueness types give us a way to do that mutation directly in our functional language, efficiently and safely.

Crucially, in these comparisons, all versions of the programs are implemented in the same language: Granule. With our extensions, the language is expressive enough to encompass a variety programming approaches. Functional programmers may freely mix and match from a variety of options for data management and manipulation. Object lifetimes may be either manually or automatically managed, and object contents may allow in-place mutation or be immutable.

5 Related work

Uniqueness types are most well known for their appearance in the Clean language [41, 35], where they are used in lieu of monadic computation and for the efficiency gains offered by in-place update. In Clean, computation is based on graph rewriting and reduction; constants such as numbers are graphs, and functions are graph rewriting formulas. This gives the type system a rather different

[7] https://hackage.haskell.org/package/vector

feel to those offered by modern functional programming languages. Some theoretical groundwork for Clean's uniqueness types has been developed over the years [47, 13]; these papers aim to clarify the distinction between Clean's type system and systems based on the lambda calculus, and also solve some problems relating to the treatment of curried functions applied to unique arguments (briefly mentioned earlier). Other languages (old and new) featuring uniqueness types include Single-Assignment C [40], Mercury [42] and Cogent [31].

Ownership was first developed as a framework for understanding aliasing in object-oriented languages [30], and is intended to give a high-level structural view of objects and references in much the same way that powerful type systems give a high-level structural view of data. Ownership is now most familiar due to being pervasive in the Rust programming language, for which multiple formalisations have been attempted; RustBelt [23] gives a lower level encoding of Rust's intended for formal verification while Oxide [55] is a higher level encoding designed for more theoretical work, among others [34]. Extending these ideas to other languages is an active area of research; RefinedC [39] is one example.

Regions have been used over the years in the context of effect systems [22, 25]. One of the primary motivations of research into region types was their application in *region-based memory management* [45], which aimed to bring some of the benefits of traditional stack-based memory management to higher-order functional languages. Regions divide values based on their lifetimes, so a system with region types can safely allocate and de-allocate memory for values based on region type information, eliminating the need for garbage collection. Early on, regions were restricted to have LIFO (last-in, first-out) lifetimes which followed the block structure of a language, but later work relaxed this constraint using uniqueness [20]; a unique reference to a region ensures there are no aliases to the region, and that it can therefore be promptly de-allocated. Additionally, regions themselves act as a way to control aliasing, and can be thought of as equivalence classes for a "may alias" relation—in other words, values which do not share a region may not alias with one another, and so if a value does not share a region with anything else then it may be safely mutated in place. Work on the Cyclone programming language [14] demonstrated the relationship between regions and unique pointers, observing that "unique pointers are essentially lightweight, dynamic regions that hold exactly one object." Beyond that, Rust's lifetimes are heavily based on regions, and there exists an extension of ML called Affe [38] which aims to support both linearity and Rust-style borrowing using regions.

Capabilities are tokens that a function must possess in order to be able to access a particular location in memory. Capabilities are linear, and cannot be duplicated or discarded, in order to prevent them from being forged. Implementations exist for various object-oriented languages such as Java [2] and Scala [18]; more functional languages taking inspiration from the idea of capabilities also exist [29, 36]. Recent work on linear constraints for Haskell [43], which hopes to allow for something similar to borrowing within the framework of linear Haskell, also descends from work on capabilities.

Fractional permissions originated from the seminal paper by Boyland [7]. Their purpose was to allow resource tracking type systems to allow multiple readers to access the same resource without losing the ability to later gain unique write access. A "permission" could be split up, allowing multiple consumers read-only access, and later combined to form a unique permission (ensuring no other permissions existed). Later, fractional permissions were further developed to allow for nesting in order to better model ownership [8].

6 Future work

6.1 Ownership via fractional permissions

Though Granule can now represent values with both linear and unique types, the language allows for much more fine-grained analysis of the resourceful behaviour of programs via *grading*. For instance, we can replay our earlier simple non-linearity example but with some extra information in the types:

```
accurate : Cake [2] → (Happy, Cake)
accurate [cake] = let extra = have cake in (eat cake, extra)
```

Granule

Instead of an infinite amount of cake we specify that we have exactly two cakes; the cake on the right-hand side must be linear as we only have one usage remaining. If we used the input three times we would receive a type error.

Given that we can move beyond the simple binary view of linear vs. unrestricted, one might suspect that we could track the quantity of existing references to a value in a more accurate way than just whether or not there are more than one. One possible avenue to doing this could take inspiration from Boyland's notion of fractional permissions [7].

Let us hypothesise that $*_1\ P$ is a 'complete' unique value that we can do anything we like with, and that we can split this up arbitrarily into 'fractionally' unique values $*_n\ P$ where $0 < n \leq 1$, as follows.

$$*_n\ P \otimes *_m\ P \longleftrightarrow *_{n+m}\ P$$

Fractional values must only be used for behaviour that does not involve mutation, as whilst a value is only fractionally unique we cannot guarantee that other references do not exist. We should only regain the ability to do these and also to discard our uniqueness guarantee for an unrestricted value if we recombine them into a complete $*_1\ P$.

This model closely resembles ownership as in Rust [23] – we can think of a value of type $*_n\ P$ for $n < 1$ as being equivalent to a Rust-style $\&P$ which is a borrowed value that we cannot mutate. When a value has been borrowed the original reference cannot be touched until we are finished with the borrows, much like we would need to collect all the fractionally unique values back together to get back to our original unique $*_1\ P$. Being able to more closely model Rust's powerful ownership system would make this a fruitful avenue for future research.

6.2 Adjoint models

Benton's linear/non-linear (LNL) logic [5] consists of two fragments: intuitionistic (non-linear) logic $\Phi \vdash_{\mathcal{I}} X$ and a mixed fragment of intuitionistic linear logic with non-linear hypotheses $\Phi, \Gamma \vdash_{\mathcal{L}} A$. These two fragments are connected by a pair of modalities $\mathrm{Lin}(X)$ and $\mathrm{Mny}(A)$, which form an adjunction. The former takes a non-linear formula, X, and brings it into the linear fragment, while $\mathrm{Mny}(A)$ brings a linear formula, A, into the non-linear fragment. The ! modality can be recovered by $!A = \mathrm{Lin}(\mathrm{Mny}(A))$.

Breaking the ! modality into two modalities and allowing linear logic to be mixed with non-linear logic has of course been a valuable endeavour, and so a natural question is whether it is possible to build an LNL-style adjoint model for our unified LCU calculus.

It seems plausible that building an adjoint model for uniqueness logic would not be too difficult; this would be very similar to the LNL model but with the adjunction moving in the opposite direction, and the monadic modality $\circ$ from uniqueness logic could be represented in such a model in much the same way that the comonadic ! can be recovered in linear/non-linear logic. An adjoint model for the full LCU calculus would be more interesting. This would most likely involve three fragments, two of which would be symmetric monoidal categories (representing unique and linear values) and one of which would be a cartesian closed category (representing unrestricted values), and two adjunctions allowing values to flow from unique to unrestricted to linear as we would hope for.

6.3 Ordered and dependent types

As expressive as Granule's type system may be, there are opportunities for expressing and enforcing stronger properties on programs elsewhere in the landscape of type theories.

One possibility is that in addition to restricting the structural rules of contraction and weakening, it is also possible to restrict *exchange*, giving ordered type theories which correspond to noncommutative logic. Such systems can be used to model stack-based memory allocation (as opposed to heap-based), since without exchange an object may only be used when it is at the top of the modelled stack [54, 10]. But much like linearity, these systems restrict the use of exchange in the *future*; is there an equivalent of uniqueness for ordered types which provides a guarantee that exchange has never been applied in the *past*, and would such a guarantee be useful for tracking references on the stack?

Another possibility is to bring uniqueness into the more powerful realm of dependent types. Recent work on graded modal dependent type theory (GrTT) [27] allows for the possibility of capturing requirements on how each variable is used at both the type-level and computation-level; grades are written in pairs, where the first component is the computation-level grading and the second component is the type-level grading. As such it is certainly possible to represent linearity at the type-level, though strictly linear usage in types is rare – but how could we interpret uniqueness at the type-level, and is there value in being able to represent such behaviour?

6.4 Linear Haskell

Granule's linear basis and assortment of modalities allows for a particularly natural embedding of the LCU calculus, but this does not preclude the theory of this paper from being applied in other contexts. One particularly valuable setting to consider would be Haskell, which as of GHC 9 already has linear types based on an underlying graded system called $\lambda^{q}_{\rightarrow}$.

Haskell's graded representation of linearity involves function types `(a %r -> b)` which have a multiplicity annotation `r`; at present, this can be either `'One` or `'Many`, corresponding to linear and unrestricted use of the argument, respectively. But $\lambda^{q}_{\rightarrow}$ is designed to be extensible, and the possibility of introducing additional multiplicities is welcomed [6, 43].

The original paper on linear Haskell [6] mentions that "*linear types are conceptually simpler than uniqueness type systems, giving a clearer path to implementation in GHC*", and also that "*functional languages have more use for fusion than in-place update*". Our clarification of the relationship between linearity and uniqueness demonstrates that not only are uniqueness types no more complex conceptually than linear ones, they can comfortably sit alongside one another in a single calculus; our evaluation demonstrates that while linearity is certainly useful, there are still further practical benefits to be gained from introducing uniqueness into a language with linear types. Perhaps these contributions will begin to forge a path towards a future for Haskell where linear types and uniqueness types can both be leveraged for their respective strengths.

7 Conclusion

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

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

Bibliography

[1] Ahrens, B., Spadotti, R.: Terminal semantics for codata types in intensional Martin-Löf type theory (2014)

[2] Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. SIGPLAN Not. **37**(11), 311–330 (Nov 2002). https://doi.org/10.1145/583854.582448, https://doi.org/10.1145/583854.582448

[3] Altenkirch, T., Chapman, J., Uustalu, T.: Monads need not be endofunctors. Logical Methods in Computer Science **11**(1) (Mar 2015). https://doi.org/10.2168/lmcs-11(1:3)2015, http://dx.doi.org/10.2168/LMCS-11(1:3)2015

[4] Baker, H.G.: Lively linear lisp: "look ma, no garbage!". SIGPLAN Not. **27**(8), 89–98 (Aug 1992). https://doi.org/10.1145/142137.142162, https://doi.org/10.1145/142137.142162

[5] Benton, N.: A mixed linear and non-linear logic: Proofs, terms and models. pp. 121–135 (09 1994). https://doi.org/10.1007/BFb0022251

[6] Bernardy, J.P., Boespflug, M., Newton, R.R., Peyton Jones, S., Spiwack, A.: Linear Haskell: Practical linearity in a higher-order polymorphic language. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158093, https://doi.org/10.1145/3158093

[7] Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) Static Analysis. pp. 55–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)

[8] Boyland, J.T.: Semantics of fractional permissions with nesting. ACM Trans. Program. Lang. Syst. **32**(6) (Aug 2010). https://doi.org/10.1145/1749608.1749611, https://doi.org/10.1145/1749608.1749611

[9] Brady, E.: Idris 2: Quantitative Type Theory in practice (2021)

[10] Bryant, A.B., Eades, H.D.: The graded lambek calculus (2020)

[11] Choudhury, P., Eades III, H., Eisenberg, R.A., Weirich, S.: A graded dependent type system with a usage-aware semantics. Proceedings of the ACM on Programming Languages **5**(POPL), 1–32 (2021)

[12] Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: From lists to streams to nothing at all. SIGPLAN Not. **42**(9), 315–326 (Oct 2007). https://doi.org/10.1145/1291220.1291199, https://doi.org/10.1145/1291220.1291199

[13] De Vries, E., Plasmeijer, R., Abrahamson, D.M.: Uniqueness Typing Simplified. In: Symposium on Implementation and Application of Functional Languages. pp. 201–218. Springer (2007). https://doi.org/10.1007/978-3-540-85373-2_12

[14] Fluet, M., Morrisett, G., Ahmed, A.: Linear regions are all you need. In: Sestoft, P. (ed.) Programming Languages and Systems. pp. 7–21. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

[15] Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1 – 101 (1987). https://doi.org/https://doi.org/10.1016/0304-3975(87)90045-4, http://www.sciencedirect.com/science/article/pii/0304397587900454

[16] Girard, J.Y., Scedrov, A., Scott, P.J.: Bounded linear logic: a modular approach to polynomial-time computability. Theoretical computer science **97**(1), 1–66 (1992)

[17] Guzman, J., Hudak, P.: Single-threaded polymorphic lambda calculus. In: [1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science. pp. 333–343 (1990). https://doi.org/10.1109/LICS.1990.113759

[18] Haller, P., Odersky, M.: Capabilities for uniqueness and borrowing. In: D'Hondt, T. (ed.) ECOOP 2010 – Object-Oriented Programming. pp. 354–378. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

[19] Harrington, D.: Uniqueness logic. Theoretical Computer Science **354**(1), 24–41 (2006)

[20] Hicks, M., Morrisett, G., Grossman, D., Jim, T.: Experience with safe manual memory-management in cyclone. In: Proceedings of the 4th International Symposium on Memory Management. p. 73–84. ISMM '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1029873.1029883, https://doi.org/10.1145/1029873.1029883

[21] Hughes, J., Marshall, D., Wood, J., Orchard, D.: Linear Exponentials as Graded Modal Types. In: 5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021). Rome (virtual), Italy (Jun 2021), https://hal-lirmm.ccsd.cnrs.fr/lirmm-03271465

[22] Jouvelot, P., Gifford, D.: Algebraic reconstruction of types and effects. In: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 303–310. POPL '91, Association for Computing Machinery, New York, NY, USA (1991). https://doi.org/10.1145/99583.99623, https://doi.org/10.1145/99583.99623

[23] Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: Securing the foundations of the Rust programming language. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158154, https://doi.org/10.1145/3158154

[24] Keller, G., Chakravarty, M.M., Leshchinskiy, R., Peyton Jones, S., Lippmeier, B.: Regular, shape-polymorphic, parallel arrays in haskell. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. p. 261–272. ICFP '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1863543.1863582, https://doi.org/10.1145/1863543.1863582

[25] Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 47–57. POPL '88, Association for Computing Machinery, New York, NY, USA (1988). https://doi.org/10.1145/73560.73564, https://doi.org/10.1145/73560.73564

[26] Mainland, G., Leshchinskiy, R., Peyton Jones, S.: Exploiting vector instructions with generalized stream fusio. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. p. 37–48. ICFP '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2500365.2500601, https://doi.org/10.1145/2500365.2500601

[27] Moon, B., Eades III, H., Orchard, D.: Graded modal dependent type theory. In: Yoshida, N. (ed.) Programming Languages and Systems. pp. 462–490. Springer International Publishing, Cham (2021)

[28] Morris, J.G.: The best of both worlds: Linear functional programming without compromise. SIGPLAN Not. **51**(9), 448–461 (Sep 2016). https://doi.org/10.1145/3022670.2951925, https://doi.org/10.1145/3022670.2951925

[29] Morrisett, G., Ahmed, A., Fluet, M.: L3: A linear language with locations. In: Urzyczyn, P. (ed.) Typed Lambda Calculi and Applications. pp. 293–307. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

[30] Mycroft, A., Voigt, J.: Notions of aliasing and ownership, pp. 59–83. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[31] O'Connor, L., Chen, Z., Rizkallah, C., Jackson, V., Amani, S., Klein, G., Murray, T., Sewell, T., Keller, G.: Cogent: Uniqueness types and certified compilation. J. Funct. Program. (2021)

[32] Orchard, D., Liepelt, V.B., Eades III, H.: Quantitative program reasoning with graded modal types. Proc. ACM Program. Lang. **3**(ICFP) (Jul 2019). https://doi.org/10.1145/3341714, https://doi.org/10.1145/3341714

[33] Orchard, D., Mycroft, A.: Categorical programming for data types with restricted parametricity. Unpublished note. Available online at http://www. cl. cam. ac. uk/˜ dao29 (2012)

[34] Pearce, D.J.: A lightweight formalism for reference lifetimes and borrowing in Rust. ACM Trans. Program. Lang. Syst. **43**(1) (Apr 2021). https://doi.org/10.1145/3443420, https://doi.org/10.1145/3443420

[35] Plasmeijer, R., van Eekelen, M., van Groningen, J.: Clean version 2.2 language report (2011)

[36] Pottier, F., Protzenko, J.: Programming with permissions in Mezzo. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. p. 173–184. ICFP '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2500365.2500598, https://doi.org/10.1145/2500365.2500598

[37] Prawitz, D.: Natural deduction: A proof-theoretical study. Stockholm Studies in Philosophy. Almqvist & Wiksell, Stockholm **3** (1965)

[38] Radanne, G., Saffrich, H., Thiemann, P.: Kindly bent to free us. Proc. ACM Program. Lang. **4**(ICFP) (Aug 2020). https://doi.org/10.1145/3408985, https://doi.org/10.1145/3408985

[39] Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., Garg, D.: RefinedC: Automating the foundational verification of C code with refined ownership types. Proc. ACM Program. Lang. (Jun 2021)

[40] Scholz, S.B.: Single Assignment C – Entwurf und Implementierung einer Funktionalen C-variante mit Spezieller Unterstützung Shape-invarianter Array-operationen. Ph.D. thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, Kiel, Germany (1996), sac-design-sbs-phd-96.pdf, shaker Verlag, Aachen, 1997

[41] Smetsers, S., Barendsen, E., van Eekelen, M., Plasmeijer, R.: Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In: Schneider, H.J., Ehrig, H. (eds.) Graph Transformations in Computer Science. pp. 358–379. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)

[42] Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. The Journal of Logic Programming **29**(1), 17–64 (1996). https://doi.org/10.1016/S0743-1066(96)00068-4, https://www.sciencedirect.com/science/article/pii/S0743106696000684, high-Performance Implementations of Logic Programming Systems

[43] Spiwack, A., Kiss, C., Bernardy, J.P., Wu, N., Eisenberg, R.: Linear constraints (2021)

[44] Terui, K.: Light affine lambda calculus and polytime strong normalization. In: LICS '01. pp. 209–220. IEEE Computer Society (2001)

[45] Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N.: A Retrospective on Region-Based Memory Management. Higher-Order and Symbolic Computation **17**(3), 245–265 (Sep 2004). https://doi.org/10.1023/B:LISP.0000029446.78563.a4, https://doi.org/10.1023/B:LISP.0000029446.78563.a4

[46] Tov, J.A., Pucella, R.: Practical affine types. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 447–458. POPL '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926436, https://doi.org/10.1145/1926385.1926436

[47] de Vries, E., Plasmeijer, R., Abrahamson, D.M.: Uniqueness typing redefined. In: Proceedings of the 18th International Conference on Implementation and Application of Functional Languages. p. 181–198. IFL'06, Springer-Verlag, Berlin, Heidelberg (2006)

[48] de Vries, E., Plasmeijer, R., Abrahamson, D.M.: Uniqueness Typing Simplified. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) Implementation and Application of Functional Languages. pp. 201–218. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)

[49] Wadler, P.: Linear Types Can Change the World! In: Programming Concepts and Methods. vol. 3, p. 5. Citeseer (1990)

[50] Wadler, P.: Is There a Use for Linear Logic? SIGPLAN Not. **26**(9), 255–273 (May 1991). https://doi.org/10.1145/115866.115894, https://doi.org/10.1145/115866.115894

[51] Wadler, P.: There's no substitute for linear logic. In: 8th International Workshop on the Mathematical Foundations of Programming Semantics (1992)

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

[53] Wadler, P.: A taste of linear logic. In: Borzyszkowski, A.M., Sokołowski, S. (eds.) Mathematical Foundations of Computer Science 1993. pp. 185–210. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)

[54] Walker, D.: Substructural type systems. Advanced Topics in Types and Programming Languages pp. 3–44 (2005)

[55] Weiss, A., Gierczak, O., Patterson, D., Matsakis, N.D., Ahmed, A.: Oxide: The essence of Rust (2020)

[56] Zhu, D., Xi, H.: Safe programming with pointers through stateful views. In: Hermenegildo, M.V., Cabeza, D. (eds.) Practical Aspects of Declarative Languages. pp. 83–97. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)