CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Reasoning about Mutability in Graded Modal Type Theory

Formalization of a graded modal typed λ-calculus with array primitives

Master's thesis in Computer science and engineering

JULIUS MAROZAS

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Master's thesis 2024

Reasoning about Mutability in Graded Modal Type Theory

Formalization of a graded modal typed λ-calculus with array primitives

JULIUS MAROZAS

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Reasoning about Mutability in Graded Modal Type Theory
Formalization of a graded modal typed λ-calculus with array primitives
JULIUS MAROZAS

Supervisor: Nils Anders Danielsson, Department of Computer Science and Engineering
Examiner: Christian Sattler, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Reasoning about Mutability in Graded Modal Type Theory
Formalization of a graded modal typed λ-calculus with array primitives
JULIUS MAROZAS
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Pure functional programming enables easier maintainability, parallelism, and reasoning about programs. However, mutable state has historically been at odds with the functional paradigm. Linear types provide a way to safely integrate mutable state into functional programming languages. This thesis explores the intersection of functional programming and mutable state, focusing on the challenges and innovations surrounding mutable arrays in languages with linear or uniqueness types. We present a partial formalization of a graded lambda calculus with array primitives. Graded modal types are used in an attempt to show that efficient mutable operations are safe. We tried to prove bisimilarity between copying and mutable operational semantics, but the proof is not complete.

Keywords: type theory, uniqueness types, linear types, graded type theory, Agda formalization.

Acknowledgements

First and foremost, I would like to thank my supervisor, Nils Anders Danielsson, for his guidance and support throughout the project. His feedback and advice have been invaluable in shaping the thesis. I would also like to thank Oskar Eriksson for explaining his work related to heap semantics for linear types and providing me with some of his code.

Julius Marozas, Gothenburg, 2024

Contents

1

Introduction

The focus of pure functional programming is *compositionality*, leading to increased code readability, expressiveness, modularity, and correctness, as highlighted by Hughes [1]. However, pure functions often operate on *immutable state* to maintain their purity. While immutable data structures work well in many programs and can lower the cognitive load for the programmer, there can be significant performance penalties arising from worse operational complexity and memory usage due to additional (de)allocations compared to an imperative approach.

There are several ways to mitigate these performance penalties for immutable structures. Fusion [2]–[4], for instance, eliminates intermediate structures, which are consumed immediately after being produced. Consider, the following Haskell function:

$$sumTo\ n = sum\ (map\ square\ [1\,..\,n])$$

Running without optimizations, the program would create two intermediate lists: $[1, 2, \ldots, n]$ (that is passed to *map*), and $[1, 4, \ldots, n^2]$ (that is passed to *sum*). With fusion, both lists can be eliminated:

$$\begin{aligned} &sumTo'\ n = h\ 0\ 1\ n \\ &\quad \textbf{where} \\ &\quad\quad h\ a\ i\ n \mid i > n \quad\ = acc \\ &\quad\quad\quad\quad\ \mid otherwise = h\ (acc + square\ i)\ (i + 1)\ n \end{aligned}$$

Now, $sumTo'$ does not allocate any lists and is thus more efficient in terms of both space and time as compared to *sumTo*. Fusion is applied when a set of specific conditions are met, including:

- producers ($[1\,..\,n]$) are directly composed with consumers (*sum* and *map*),
- function definitions (*sum* and *map*, in this case) can be inlined.

The above conditions are brittle in practise as small changes to the code might stop them from holding, for instance, definitions might not be subject for inlining when they are defined in separate modules or when they are recursive. The fact that fusion is done automatically, makes reasoning about the performance of programs difficult. Simply moving a function to a separate module might severly change the performance characteristics of the program.

Another solution is to add in-place updates to a language directly while maintaining referential transparency. This can be done by ensuring purity through the type system.

For example, in Haskell mutable state is commonly used via monads like IO or ST. This way, mutation is tracked explicitly and performance can be ensured with a higher degree of certainty. However, monadic approaches suffer from making programs overly sequential which prevents parallelization and compiler optimizations (such as reordering instructions). Furthermore, some operations like *freeze*, which casts a mutable array to an immutable one, or *free*, which deallocates an array, are *unsafe*, meaning that the type system cannot ensure the safety of the program. Recently, Haskell has been extended with *linear types* through the Linear Haskell [5] language extension. One of the core motivations of Linear Haskell is modelling safe mutable state by enforcing that mutable references are used exactly once. This is an exciting development, as it forgoes the limitations of monadic-based approach, e.g., *freeze* is safe, as well as being arguably easier to use. For instance, parallelism is much easier to achieve with linearly-encoded mutable arrays [6].

Dependently-typed languages are also experimenting with substructural type theory in the interest of allowing reasoning about resources. Idris 1 [7] and ATS [8] support uniqueness types. Idris 2 [9] moved to a generalization of linear types instead, which is based on Quantified Type Theory (QTT) [10], [11]. The new version of Idris also came with a complete rewrite of the compiler, and Idris is now implemented in Idris itself [12]. This has proven beneficial in ensuring that terms are always well-scoped (terms are indexed by the number of free variables in their context). Additionally, to achieve good performance, mutable arrays are used to represent the typing context inside the compiler[13].

Meanwhile, there are also discussions [14] to add (some form of) linear types to Agda as well. However, naïvely adding linear types (or QTT) to Agda is complicated by the presence of the Cubical Agda extension: the syntactic nature of linearity conflicts with the extensional nature of Cubical Type Theory [15]. The interest of introducing linear types to Agda in a sound way has motivated further research [16].

Noticing the need for mutable arrays in pure functional (as well as dependently-typed) languages, we explore the design space of safe mutable array primitives and provide a formalization of a language with array primitives. For simplicity, we chose to work with a graded simply-typed lambda calculus, but with the possibility of extending the language to a dependently-typed setting in the future. The existing formalization by Abel *et al.* [16] was used as a starting point for the project.

1.1 Goals

The primary goals of the thesis are as follows:

1. Investigate how languages with linear/uniqueness types deal with mutable arrays.
2. Formalize a graded lambda calculus with array primitives in Agda.
3. Formulate and prove correctness properties of the language.

1.2 Outline

The structure of the thesis is as follows:

1. Chapter 2 provides an overview of different ways to model safe mutable state, focusing specifically on mutable arrays. The deficiencies of common approaches in Haskell, like the ST monad, are presented. Type systems based on substructural type theories (uniqueness and linear types) are presented as an alternative. Finally, graded modalities are motivated as a generalization of linear types.
2. Chapter 3 presents the main part of the thesis: the formalization. After presenting the syntax and rules of the language, the semantics are defined in terms of heaps. Finally, the bisimilarity theorems are stated.
3. Chapter 4 investigates possible approaches of extending the formalization, such as, adding an untyped language for a separate optimization step, moving to a dependently-typed setting.
4. Chapter 5 summarizes the results of the study and outlines future work.

Disclaimer. *The formalization presented in Chapter 3 is a work in progress and is not yet complete. Hence, the presented results (e.g. the reduction rules) may look different if the formalization is finished.*

2

Background

In this section, we will look at how introducing mutable state into a language can lead to impurity and the loss of referential transparency. We will then discuss how monads can be used to encapsulate mutable state in a pure functional language, and the limitations of this approach. We will then introduce substructural type systems, such as linear and uniqueness types, and show how they can be used to model safe mutable state APIs. Finally, we will discuss graded modal types, a generalization of linear types, and how they can be used to model more fine-grained properties of mutable state.

2.1 Naïve mutable state

Let's take a very naïve interface for mutable arrays, where the write operation performs an in-place update.

$$
\begin{aligned}
&\textbf{type } \mathit{MArray}\ a \\
&\mathit{newArray} \ :: \mathit{Int} \to a \to \mathit{MArray}\ a \\
&\mathit{readArray} \ :: \mathit{MArray}\ a \to \mathit{Int} \to a \\
&\mathit{writeArray} :: \mathit{MArray}\ a \to \mathit{Int} \to a \to ()
\end{aligned}
$$

Now, consider the following program that simply counts the values passed to it:

$$
\begin{aligned}
&\mathit{frequencyCounter} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
&\mathit{frequencyCounter}\ n = \\
&\quad \textbf{let } \mathit{freqs} = \mathit{newArray}\ n\ 0 \\
&\quad \textbf{in } \lambda i \to \\
&\qquad \textbf{let } \mathit{count} = \mathit{readArray}\ \mathit{freqs}\ i + 1 \\
&\qquad\quad\ \ !\,() \quad = \mathit{writeArray}\ \mathit{freqs}\ i\ \mathit{count} \\
&\qquad \textbf{in } \mathit{count}
\end{aligned}
$$

frequencyCounter n creates n counters for values from 0 to $n - 1$, and returns a function that increments the counter at a given index each time it is called, returning the new count as a result. In the context of pure functional programming, this presents a problem—on the outside (looking at the type) the function looks as though it is pure, but its behaviour is anything but: calling the function performs a side-effect (mutating the array in its closure), which makes multiple calls with the same argument produce different results.

We would like to resolve this discreptancy by making the function's type reflect its impurity.

Doing this directly, would yield something like the following:

```
type State = MArray Int
frequencyCounter :: Int → (State, (State, Int) → (State, Int))
frequencyCounter n = (newArray n 0, count) where
  count (freqs, i) =
    let count = readArray freqs i + 1
        ! ()  = writeArray freqs i count
    in (freqs, count)
```

In this version, we track the internal state of the counter explicitly: the state of the counter is returned when it is first created, while for updates, the previous state is passed as an argument and the updated state is returned. Now, the function's signature is a bit more truthful about its behaviour, but for it to be called pure, the user still has multiple obligations to fulfill:

- the state should not be inspected or modified by other parts of the program,
- the state should not be used multiple times.

The presence of such obligations makes *frequencyCounter unsafe*, we would like to use the type system to get rid of them. The first obligation could be solved by making the state opaque (i.e., defining a new type and making the constructor private), but solving the latter one is more complicated and we will look at some methods to do so in the following sections.

2.2 Mutable state with monads

One possible way to ensure that mutable state is carefully threaded (without being duplicated) through the program is to use *monads*. A common example is the *ST*[1] monad [17]. *ST* provides a safe way to embed an imperative algorithm that uses in-place mutable state inside a pure functional program. Looking at its simplified definition (Figure 2.1), *ST* is a state-passing function that takes an initial state and returns a new state along with a result.

```
newtype ST s a = MkST (s → (s, a))
runST :: (∀s.ST s a) → a
instance Monad (ST s)
```

Figure 2.1: The ST monad in Haskell (simplified). Note that the constructor *MkST* is private.

[1] ST stands for *state transformer*, or *state thread*

ST can be classified as a *region-based* approach for ensuring safe mutable state. The region-based approach is based on the idea that mutable state is only valid within a certain region of the program. In the case of *ST*, the region can be seen as the scope of the *s* type variable introduced by the explicit forall in the *runST* function. This way, *runST* ensures that the state is not leaked outside the monadic context so that all references and arrays (which are also annotated with an *s*) stay local as they cannot be used in separate *runST* calls.

The API for mutable *ST* arrays is presented below.

```
type STArray s a
newArray  :: Int → a → ST s (STArray s a)
readArray  :: STArray s a → Int → ST s a
writeArray :: STArray s a → Int → a → ST s ()
```

Notice that the return type of each primitive is wrapped in a monadic context *ST s*. Intuitively, *STArray s a* is simply a pointer to an array in memory, and array primitives represent actions (allocating, reading, mutating) on the array, which are only executed in the *ST s* context. Rewriting *frequencyCounter* with *ST* primitives, forces us to change the type of our function:

```
frequencyCounter :: Int → ST s (Int → ST s Int)
frequencyCounter n = do
  freqs ← newArray n 0
  return $ λi → do
    prev ← readArray freqs i
    let count = prev + 1
    () ← writeArray freqs i count
    return count
```

Now, *frequencyCounter* returns a function that, when executed in the *ST* context, will return a new counter. The function *incr* is also monadic, as it performs actions on the array *freqs*.

2.2.1 Limitations of the monadic approach

However, there a several problems when it comes to programming with *ST*. The core issue lies in the definition *ST*—the state token is *globally* threaded through the whole *ST* program.

Sequencing prevents compiler optimizations. In pure non-monadic code, the compiler is free to reorder or even automatically parallelize independent operations. However, in *ST* (and *IO*) programs, the state token constrains the order of operations by adding an artificial dependency between them. For example, when a program contains multiple arrays, some operations might be independent of each other and could be prone to optimizations, but the compiler will have a hard time applying them.

Harder to parallelize. It is difficult to parallelize an *ST* program without losing the ability to return to the "pure world" (which can be done with *ST* but not *IO*). In his book *Parallel and Concurrent Programming in Haskell*, Marlow [18] writes: "...it is typically difficult to use parallelism within the ST monad, and in that case probably the only solution is to drop down to concurrency..."

Limited safe primitives. Some operations are inherently unsafe in the *ST* monad due to the lack of guarantees it can provide. For instance, the operation to freeze[2] an array is one such example:

$$unsafeFreeze :: STArray\ s\ a \to ST\ s\ (Array\ a)$$

Since nothing prevents the programmer from mutating the array through the $STArray\ s\ a$ variable, *unsafeFreeze* is not safe as its usage can break the immutability invariant that the type $Array\ a$ should ensure.

2.3 Substructural type systems

In order to solve the inadequacies of the monadic approach, we could look at the problem of modelling mutable state in a pure way from another viewpoint. Instead of tracking mutability as a side effect of the returned value, we can view it as an effect on the context—the consumption of the array variable. Effects on the context are better known as *coeffects* [19], and substructural type theory provides a way to model them. A type system is called *substructural* [20] when at least one of the following rules are not admissible:

$$\frac{\Gamma \vdash t : B}{\Gamma, x : A \vdash t : B}\ \textsc{Weakening} \qquad \frac{\Gamma, x_1 : A, x_2 : A \vdash t : B}{\Gamma, x : A \vdash t[x_1 := x][x_2 := x] : B}\ \textsc{Contraction}$$

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

Weakening serves as a way to discard a variable (by introducing an unnecessary assumption in the context), *contraction* allows a variable to be used more than once (by combining two identical assumptions into one), and *exchange* enables variables to be reordered in the context.

As previously noted (Section 2.1), we want to ensure that mutable array variables are not duplicated, that is, we want to disallow contraction.

Contraction can be restricted through *uniqueness types* and *linear types*. Linearity and uniqueness are two common flavours of *substructural* type systems: both restrict contraction and weakening. They have been implemented in several languages, e.g., Clean [21], [22], ATS [8], Linear Haskell [5], Granule [23] and Idris [9]. Linearity and uniqueness are

[2] Freezing is a coercion of a mutable array to an immutable one without copying the underlying data in memory.

closely related concepts, and as a result, they are often confused between each other, but they are in fact dual from the perspective of time. Marshall *et al.* [24] explain the duality on a more intuitive level:

"*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**".

We will cover both systems and their applications to mutable arrays in the following sections.

2.3.1 Uniqueness types in Clean

Uniqueness types were introduced by Smetsers *et al.* [25] as a way to embed safe destructive updates in a pure functional language. Clean [26] is largely similar to Haskell[3] but one of its main innovations is support for *uniqueness types*. Values of a unique type (types annotated with a star $*$) are guaranteed to have no other references. This is achieved by changing the uniquenes of a variables[4] according to its usage:

- if a variable is used once, the uniqueness at the usage site can be preserved,
- otherwise (using a variable more than once), makes all references of the variable non-unique.

The loss of uniqueness (from the latter point), can be seen in the ill-typed *cannotShare* and well-typed *shareWithoutUniqueness* examples (Figure 2.2).

Remark. Note that uniqueness is propagated outwards, i.e., for any type constructor T and type A, $T\ (*A)$ is the same type as $*(T\ (*A))$.

$$
\begin{aligned}
&\mathit{cannotIntroUniqueness} :: a \to *a \\
&\mathit{cannotIntroUniqueness}\ x = x \\
&\mathit{cannotShare} :: *a \to (*a, *a) \\
&\mathit{cannotShare}\ x = (x, x)
\end{aligned}
$$

(a) Ill-typed programs in Clean.

$$
\begin{aligned}
&\mathit{discard} :: *a \to () \\
&\mathit{discard}\ x = () \\
&\mathit{forgetUniqueness} :: *a \to a \\
&\mathit{forgetUniqueness}\ x = x \\
&\mathit{shareWithoutUniqueness} :: *a \to (a, a) \\
&\mathit{shareWithoutUniqueness}\ x = (x, x)
\end{aligned}
$$

(b) Well-typed programs in Clean.

Figure 2.2: Example programs in Clean.

One of the core motivations of uniqueness types was the ability to perform destructive updates on unique arrays. As an example, the program below doubles the elements of a unique array (starting from a given index).

$$
\begin{aligned}
&\mathit{double} :: *\{\mathit{Real}\}\ \mathit{Int} \to *\{\mathit{Real}\} \\
&\mathit{double}\ \mathit{arr}\ 0 = \mathit{arr}
\end{aligned}
$$

[3] In terms of syntax, lazy semantics.

[4] By uniqueness of a variable, we mean whether the type of the variable is unique ($*A$) or not (A).

$$
\begin{aligned}
&\mathit{double}\ \mathit{arr}\ i \\
&\quad \#\ (x, \mathit{arr}) = \mathit{arr}\ ![\,i\,] \\
&\quad \#\ \mathit{arr} = \{\mathit{arr}\ \&\ [\,i\,] = 2.0 * x\} \\
&\quad = \mathit{double}\ \mathit{arr}\ (i - 1)
\end{aligned}
$$

The function *double* takes two arguments, a unique array and an index i, and returns a unique array. If the index is zero, the array is returned immediately. Otherwise, the element x at index i is looked up in the array through the indexing operator $-![-]$ which consumes the unique array and returns a "new" unique array (but which represents the same location in memory). The array is then updated at the index in-place and the function continues recursively until i reaches zero.

While arrays and their operations have dedicated syntax in Clean, we can write out the array interface explicitly (see Figure 2.3).

$$
\begin{aligned}
&\mathit{newArray} &&:: \mathit{Int} \to a \to *\{a\} \\
&\mathit{readArray} &&:: *\{a\} \to \mathit{Int} \to (a, *\{a\}) \\
&\mathit{writeArray} &&:: *\{a\} \to \mathit{Int} \to a \to *\{a\} \\
&\mathit{freezeArray} &&:: *\{a\} \to \{a\}
\end{aligned}
$$

Figure 2.3: Signatures for array primitives in Clean.

Note that *freezeArray* can be implemented safely in Clean. It is simply the identity function, since a unique value can be implicitly converted to a non-unique one by forgetting the guarantee of uniqueness.

2.3.2 Necessarily unique values

There is a caveat with uniqueness types, which complicates the type system: when a unique variable is captured in a function's closure, the function may not be shared (called multiple times), since the captured variable must be unique. Thus, functions with unique variables in their closures can neither be unique nor non-unique as both options would permit sharing. Instead, Clean makes such functions *necessarily unique* which have to remain unique and cannot be sutyped into a non-unique function.

As an example, consider the function for appending an element into a unique array (reallocating the array with a larger size):

$$
\mathit{append} :: *\{a\} \to a \to *\{a\}
$$

Partially applying the function (given an unique array $\mathit{arr} :: *\{a\}$), we get a necessarily unique function $\mathit{append\ arr} :: a \to *a$. The resulting function can only be used (called) once—it cannot be implicitly converted to a non-unique function, contrary to unique values. Without necessary uniqueness (if *append arr* was simply unique), we would lose safety—the first call would reallocate the array and subsequent calls would use memory that has been freed (use-after-free).

Another way [27] to understand the problem is to think of a function capturing variables as a *container*, and variables in the closure as their elements. In terms of uniqueness, containers (e.g., arrays) have a property that uniqueness is propagated outwards—as soon as a value inside a containers is marked unique, the whole container becomes unique[5]. Or, in other words, if we want to extract a unique value from a container, the container itself has to be unique. Hence, a function that captures a unique variable (and uses it uniquely in its body) has to extract a unique value when it is called, and, thus, has to be unique. Crucially, a container cannot be shared without also sharing its elements. But since it might not be possible to share a captured variable (the body might require it to be unique), a function capturing a unique variable must remain unique.

2.3.3 Linear types

Linear types originate from Girard's linear logic [28]. In contrast to uniqueness types, which originally appeared out of practical needs for safe destructive updates [25], linearity has been first explored in a more theoretical setting [29] and it appeared in mainstream programming languages relatively recently (Linear Haskell [5]).

At the core of linear types, we have the idea of *linearity*—a guarantee that a value will be used exactly once[6]. Following Wadler [29], the function space is represented with a linear function arrow, denoted by $A \multimap B$, which indicates that the argument (of type A) will be used exactly once in the body of the function. The rules for the core λ-calculus are as follows:

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

The variable rule contains only necessary assumptions, and in the application rule Γ, Δ indicates that t and u have distinct sets of free variables. As a consequence of the above rules, weakening and contraction do not hold, in general.

Instead, weakening and contraction only hold for non-linear values, which are behind a new type constructor !. This also allows us to define the usual non-linear function space: $A \to B = !A \multimap B$.

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

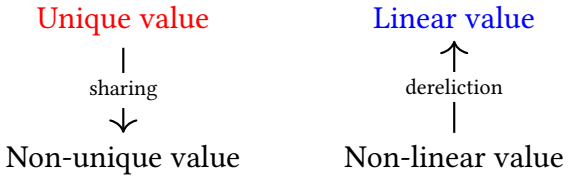
$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : !A \vdash t : B}\,\textsc{Dereliction} \qquad \frac{\Gamma, x : !A, y : !A \vdash t : B}{\Gamma, x : !A \vdash t : B}\,\textsc{Contraction}$$

In the rules above, promotion is a !-introduction rule, the rest are !-elimination rules. The three elimination rules correspond to possible uses of a non-linear value: used once (dereliction), none at all (weakening), or many times (contraction). $!\Gamma$ means that all variables in the context Γ are non-linear.

[5] In Clean, this means that writing $\{*a\}$ is the same as $*\{*a\}$.

[6] Uniqueness types, for contrast, guarantee that a value *has* not been shared.

Uniqueness vs linearity. Clean's uniqueness types can be slightly reformulated such that the relationship with linear types becomes more apparent, as shown by Marshall *et al.* [24]. This can be done by introducing an explicit *share* $:: *A \to A$ operation and removing the rules for implicitly changing the uniqueness of a variable (when it is used more than once). Additionally, if we remove the weakening rule for unique values (we can still recover it by using *share*), we get a system, that is still equivalent to that of Clean, but is almost the same in presentation to linear types. More concretely, the unique fragment (in uniqueness types) behaves exactly the same as the linear fragment (in linear types). But the difference lies in how the restricted types interact with their unrestricted counterparts: converting to (or from) non-linear and non-unique values. Namely, a unique value can be converted to a non-unique one (through sharing which forgets the guarantee of uniqueness); but for linear values the opposite is true—a non-linear value can be converted to a linear one (through *dereliction*). And importantly, going back (from non-unique to unique, or linear to non-linear) is disallowed, in general.



The fact that discarding a linear value in general is not permitted allows for reasoning about resources such as file handles or network sockets, since we often want to ensure that they are eventually closed. This is not possible with unique types, since they cannot provide a guarantee "in the future", e.g., that a resource will eventually be closed.

2.3.4 Unique vs linear arrays

At first glance, it does seem that the notion of uniqueness is more fitting for modelling state with destructive updates. A unique array guarantees that it has not been shared, which forces mutations to be local and thus retains referential transparency. On the other hand, a linear array only guarantees that it will be used exactly once, its type does not guarantee that it has not been shared before. So, it is perhaps a little less obvious how to exploit the linearity property for modelling mutable state. With linear types, the responsibility of ensuring safety falls on the design of the mutable array interface, instead of the type system. A linear array interface has to be defined in such a way that any value that is capable of mutating an array is linear. Otherwise, mutations could be done non-locally, which would break referential transparency.

There are also some subtle differences in regards to how an immutable array type should be defined in such systems. With uniqueness types, it is enough to have a single array type, where the mutable variant is the unique version and the immutable one is non-unique. But in the case of linearity, this is not possible due to the fact that the dereliction rule allows for converting a non-linear value to a linear one. In our case, dereliction would correspond to casting an immutable array to a mutable one without any way to ensure that the immutable array is not used somewhere else. A common way to fix this is to have completely separate types for mutable and immutable arrays.

2.3.5 Linear arrays in Linear Haskell

Linear Haskell [5] is a language extension for Haskell, which adds support for linear functions. By enabling the extension, functions can be annotated by a multiplicity: either 1 (linear) or ω (unrestricted). In other words, the ordinary function type $a \to b$, representing a function that can use its argument any number of times, is now defined to be $a \,\%\omega \to b$; and a new linear function type $a \,\%1 \to b$ is introduced which guarantees that the argument will be used exactly once.

Additionally, it's possible to represent unrestricted values (not just function arguments). This can be done by defining a new type *Ur* (unrestricted) which corresponds to ! in linear logic:

```
data Ur a where
  MkUr :: a %ω → Ur a
```

The constructor *MkUr* demands that the argument is non-linear, allowing *Ur a* to be passed linearly which, after unpacking to *a*, becomes non-linear. This can be seen in action in the mutable array API (Figure 2.4).

```
type MArray a
type Array a
new    :: a → Int → (MArray a ⊸ Ur b) ⊸ Ur b
readArray  :: MArray a ⊸ Int → (MArray a, Ur a)
writeArray :: MArray a ⊸ (Int, a) → MArray a
freeze :: MArray a ⊸ Ur (Array a)
```

Figure 2.4: Signatures for linear array primitives [5]

The array is allocated by *new* which takes a linear continuation with a linear mutable array as its argument. The continuation is expected to return an unrestricted (non-linear) value *Ur b*, preventing the array itself from being returned. The final result is also wrapped with *Ur* to allow the caller to use the value as many times as needed. The same is the case for *readArray* which returns the original linear array and an unrestricted element, as well as *freeze* which consumes the mutable linear array and returns an immutable non-linear array (without copying).

2.3.6 Linearly – source of uniqueness

Spiwack *et al.* [30] identified the following shortcomings of the above interface:

- *readArray* and *writeArray* consume the input array, and produce a new one, creating unwanted boilerplate for the programmer,
- nested array allocations are cumbersome to handle, as the linear array cannot be returned from the inner continuation.

To address the first issue, the authors change the API so that instead of the array being linear, it is the *capabilities* (access to read or write the array) that are linear. The second point is addressed through a new *Linearly* constraint:

$$
\begin{aligned}
&\mathit{linearly} \;\; :: (\mathit{Linearly}\ \%1 \Rightarrow \mathit{Ur}\ a) \multimap \mathit{Ur}\ a \\
&\mathit{consume} \;\; :: \mathit{Linearly}\ \%1 \Rightarrow a \to () \\
&\mathit{duplicate} :: \mathit{Linearly}\ \%1 \Rightarrow a \to () \wedge \mathit{Linearly} \wedge \mathit{Linearly} \\
&\textbf{data}\ a \wedge (c :: \mathit{Constraint})\ \textbf{where} \\
&\quad \mathit{Wrap} :: c\ \%1 \Rightarrow a\ \%1 \to a \wedge c
\end{aligned}
$$

The data type $a \wedge c$, defined above, allows linear contraints to be packed together with some value, in order for them to be returned. *Linearly*, introduced with *linearly*, acts as a source of uniqueness, permitting allocation of resources such as arrays with *new* (from Figure 2.5). Additionally, *Linearly* forms a comonoid: it can be consumed or duplicated[7], but only a finite number of times, i.e., one cannot satisfy a constraint *Linearly* at multiplicity ω. Furthermore, the paper models read/write permissions with linear constraints as well, which allows for more fine-grained control over the array operations.

$$
\begin{aligned}
&\textbf{class}\ \mathit{Read} \\
&\textbf{class}\ \mathit{Write} \\
&\textbf{type}\ \mathit{RW}\ n = (\mathit{Read}\ n, \mathit{Write}\ n) \\
&\mathit{newArray} \;\; :: \mathit{Linearly}\ \%1 \Rightarrow a \to \mathit{Int} \to \exists n.\mathit{UArray}\ a\ n \wedge \mathit{RW}\ n \\
&\mathit{writeArray} :: \mathit{RW}\ n \quad \%1 \Rightarrow \mathit{UArray}\ a\ n \to \mathit{Int} \to a \to () \wedge \mathit{RW}\ n \\
&\mathit{readArray} \;\; :: \mathit{Read}\ n \quad \%1 \Rightarrow \mathit{UArray}\ a\ n \to \mathit{Int} \to \mathit{Ur}\ a \wedge \mathit{Read}\ n \\
&\mathit{freeArray} \;\; :: \mathit{RW}\ n \quad \%1 \Rightarrow \mathit{UArray}\ a\ n \to ()
\end{aligned}
$$

Figure 2.5: Signatures for array primitives with linear constraints

Due to the linear constraint *Linearly* on *newArray*, the resulting capability-constraint *RWn* will be linear. Furthermore, due to *Linearly* being a comonoid, *newArray* can be easily called multiple times in a single continuation (of *linearly*), thus making the API less cumbersome to use.

2.4 Graded modal types

Linearity in type theories can be viewed as a *modality*—a qualifier that modifies a type, just like in modal logic where modalities qualify statements. This perspective has been explored and generalized by Abel and Bernardy [31]. The authors introduce a general way to define graded modal typed lambda calculi by using a semiring structure to represent *modalities* (multiplicities). The following definitions are based on Abel and Bernardy [31], Abel *et al.* [16], and McBride [10].

[7] Although presented here explicitly, these operations are implicit and are able to be inferred by the compiler [30].

Definition 2.4.1. A modality semiring is a 6-tuple $(\mathcal{M}, +, \cdot, \wedge, 0, 1)$ consisting of a set $\mathcal{M}$, three binary operations: addition $+$, multiplication $\cdot$, and meet $\wedge$; and two elements zero ($0 \in \mathcal{M}$) and unit ($1 \in \mathcal{M}$); with the following structure:

- $(\mathcal{M}, +, \cdot, 0, 1)$ forms a semiring:
  - $(\mathcal{M}, +, 0)$ forms a commutative monoid: addition is associative and commutative, with 0 as identity element.
  - $(\mathcal{M}, \cdot, 1)$ forms a monoid: multiplication is associative, with 1 as identity element.
  - 0 is an absorbing element for multiplication: $p \cdot 0 = 0 \cdot p = 0$.
  - Multiplication distributes over addition: $p(q + r) = pq + pr$ and $(p + q)r = pr + qr$.
- $(\mathcal{M}, \wedge)$ forms a semilattice: meet is associative, commutative and idempotent.
- Addition distributes over meet: $(p \wedge q) + r = (p + r) \wedge (q + r)$.
- Multiplication distributes over meet (like for addition).

Definition 2.4.2. A partial ordering on multiplicities, defined as $(p \leq q) \stackrel{\text{def}}{=} (p = p \wedge q)$, which is standard for semi-lattices.

Definition 2.4.3. A modality context (denoted γ, δ, ζ) is a map from variables to modalities. Addition, meet and scaling by q is lifted to act pointwise on modality contexts:

$$
\begin{aligned}
(\gamma + \delta)(x) &= \gamma(x) + \delta(x) \\
(\gamma \wedge \delta)(x) &= \gamma(x) \wedge \delta(x) \\
(q \cdot \delta)(x) &= q \cdot \delta(x)
\end{aligned}
$$

Additionally, a modality context forms a left semimodule over $\mathcal{M}$, satisfying the following laws:

$$
\begin{array}{lcc}
1 \cdot p = p & (p + q) \cdot \gamma = p \cdot \gamma + q \cdot \gamma & p \cdot \mathbf{0} = \mathbf{0} \\
0 \cdot p = 0 & (pq) \cdot \gamma = p(q \cdot \gamma) & p \cdot (\gamma + \delta) = p \cdot \gamma + p \cdot \delta \\
 & (p \wedge q) \cdot \gamma = p \cdot \gamma + q \cdot \gamma & p \cdot (\gamma \wedge \delta) = p \cdot \gamma \wedge p \cdot \delta
\end{array}
$$

Here $\mathbf{0}$ denotes a modality context mapping all variables to 0.

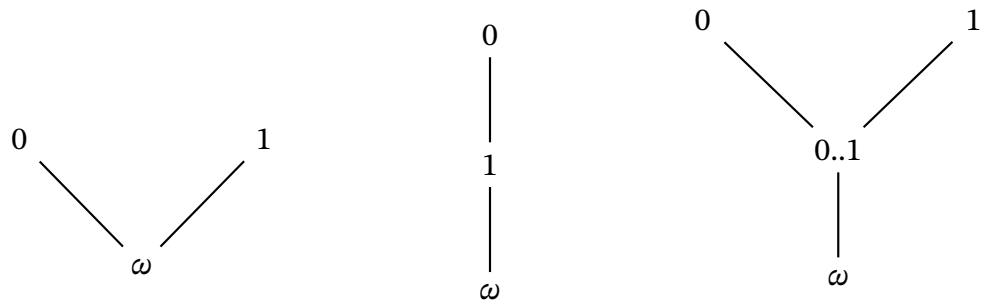
2.4.1 Examples

As an example we can model linearity with a modality semiring $(\{0, 1, \omega\}, +, \cdot, \wedge, 0, 1)$ where $0 \geq \omega \leq 1$ with addition and multiplication operate according to Table 2.1 (which simply comes out from the laws). Such modality corresponds to the same system as in Linear Haskell but with an added 0 multiplicity for erased values.

$(+)$	0	1	ω
0	0	1	ω
1	1	ω	ω
ω	ω	ω	ω

$(\cdot)$	0	1	ω
0	0	0	0
1	0	1	ω
ω	0	ω	ω

Table 2.1: Addition and multiplication for linearity modality.



(a) Linearity modality (b) Affine modality (c) Combined modality

Figure 2.6: Hasse diagrams of some modality semirings.

Other examples include affinity, where the unit modality (1) represents "0 or 1 uses", effectively permitting weakening. The lattice structure of linearity, affinity, as well combined modality, are visualized Figure 2.6.

In the rest of the thesis, we will mostly focus on linear types (and graded modal types). This is mainly due to the original motivation, which was to add mutable state in a dependently-typed setting. Combining dependent types with linear types has been explored much more in the literature [10], [11] compared to doing the same but with uniqueness types. Furthermore, graded modal type theory is a generalization of linear types, and as such, it is more natural to start with linear types as a foundation.

3

Formalization

Motivated by the fact that mutable array APIs with linear types (by [5], and its later modification by [30]) has not been formalized before[1](to the best of our knowledge), we present an Agda formalization to verify the correctness of such APIs. More specifically, we will focus on defining a linearly-style [30] mutable array API (Section 2.3.6) with graded modal types. For simplification, we will not model read/write access capabilities for arrays, but these could be added in future work.

The Agda formalization[2] consists of a λ-calculus language with primitive array operations, and graded arrows over a modality semiring. We rely on previous work by Abel *et al.* [16] in formalizing modality semirings and contexts. We will first define the syntax, then give the typing rules, and, in the style of Abel *et al.* [16], give the usage rules (that count the resources consumed by a term). After that, the semantics of the language will be defined in terms of heaps which will allow us to explore properties such as bisimulation.

In the formalization, the syntax is defined in an *intrinsically-typed* manner, meaning that the syntax and the typing rules are merged together. This ensures that all expressible terms are also well-typed without doing any extra work[3]. In this section, however, we will present the syntax and typing rules separately for clarity. Furthermore, the formalization uses de Bruijn indices to represent variables.

Disclaimer. *The formalization is a work in progress and is not yet complete. The syntax and semantics are fully defined, but the proof of bisimilarity and some other things have not been completed.*

[1]Bernardy *et al.* [5] do include proofs of their results, but they only appear in the unpublished appendix and are quite brief.

[2]"Formalization of graded modal type theory extended with mutable arrays." (2024), [Online]. Available: https://github.com/jul1u5/graded-type-theory/tree/9ff712147e99a11449fdbb83cd95df505ef2b28d (visited on 09/23/2024).

[3]An alternative would be to first define the raw terms of the language (the syntax), and then, separately, define the typing relation. However, this requires us to prove properties like *type-preservation*, requiring more code. Wadler [33] estimates that this approach uses about 1.6 as many lines of Agda code as the intrinsically-typed one. One possible complication with the intrinsically-typed approach is that it is harder to come up with a way to define the terms (e.g., encoding intrinsically-typed dependently-typed terms is an active area of research [34]), but in our case this does not complicate things.

The language consists of the following terms:

$$
\begin{aligned}
t ::= {} & x \mid \lambda^p t \mid t\,{}^p u \\
& \mid \mathbf{zero} \mid \mathbf{suc}\ t \mid \mathbf{natcase}_p\ t\ t\ t \\
& \mid \star \mid \mathbf{let}\ \star = t\ \mathbf{in}\ t \\
& \mid [^p t] \mid \mathbf{let}\ [^p] = t\ \mathbf{in}\ t \\
& \mid (t, t) \mid \mathbf{let}\ (,) = t\ \mathbf{in}\ t \\
& \mid \mathbf{linearly}\ t \mid \mathbf{consume}\ t \mid \mathbf{duplicate}\ t \\
& \mid \mathbf{new}\ t\ t \mid \mathbf{read}\ t\ t \mid \mathbf{write}\ t\ t\ t \mid \mathbf{free}\ t
\end{aligned}
\tag{terms}
$$

Let bindings are defined as follows:

$$\mathbf{let}_p\ t\ \mathbf{in}\ u \stackrel{\text{def}}{=} (\lambda^p u)\,{}^p t$$

We will explain them in the following section with the typing rules.

Disclaimer. *In the formalization, only ω-specialized boxes ($[^\omega t]$) are included,* **natcase** *is also currently missing.*

3.1 Typing rules

The typing judgement $\Gamma \vdash t : A$ shows that under context Γ a term t is of type A.

First, we will cover the types:

$$A\ B ::= A \Rightarrow_p B \mid \mathbb{N} \mid \mathsf{Unit} \mid \Box_p A \mid A \otimes B \mid \mathsf{Lin} \mid \mathsf{Arr} \tag{types}$$

Compared to the simply-typed λ-calculus, the function arrow is replaced with a graded one, annotated with a multiplicity grade p representing the grade of the argument. Besides the natural numbers, and the unit type, there is: a box type containing a value with a given modality, a linear product type where both elements have to be consumed linearly, the Lin type corresponds to the linear constraint seen in Section 2.3.6, and Arr type represents arrays of arbitrary size consisting of naturals numbers[4].

As an example, the linearity modality ("none-one-tons" [10], see Figure 2.6a) can be used for the modality semiring. But the language is parameterized over a modality semiring which allows us to use other semirings as well such as the affinity modality. In general, we require that there is some element ω with $0 \geq \omega \leq 1$ in the semilattice structure. Later, when introducing the semantics, we will add some additional restrictions.

To be more concise, we will denote $A \Rightarrow_1 B$ by $A \multimap B$, and $\Box_\omega A$ by $!A$ (which is also the same as *Ur* in Linear Haskell), following Girard's [28] notation for linear logic.

The contexts are defined as follows:

$$\Gamma ::= \varepsilon \mid \Gamma, A \tag{contexts}$$

[4]We restrict to natural numbers instead of arbitrary types to simplify the formalization.

As usual, a context can either be empty or be a context extension with a variable (Γ, A).

The judgement $x : A \in \Gamma$ checks that variable x is inside the context Γ with type A, and is defined below:

$$\frac{}{0 : A \in \Gamma, A} \qquad \frac{x : A \in \Gamma}{x + 1 : A \in \Gamma, B}$$

At the core of the language are the following λ-calculus typing rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\,\textsc{Var} \qquad \frac{\Gamma, A \vdash t : B}{\Gamma \vdash \lambda^p t : A \Rightarrow_p B}\,\textsc{Abs} \qquad \frac{\Gamma \vdash t : A \Rightarrow_p B \quad \Gamma \vdash u : A}{\Gamma \vdash t^p u : B}\,\textsc{App}$$

Lambda abstractions and applications are annotated by the grade of the arrow, following Abel *et al.* [16].

Next, we have constructors for the natural numbers, together with its eliminator:

$$\frac{}{\Gamma \vdash \mathbf{zero} : \mathbb{N}}\,\textsc{Nat-Z} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathbf{suc}\ n : \mathbb{N}}\,\textsc{Nat-S}$$

$$\frac{\Gamma \vdash z : A \quad \Gamma \vdash s : \mathbb{N} \Rightarrow_p A \quad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathbf{natcase}_p\ z\ s\ n : A}\,\textsc{Nat-Case}$$

The introduction and elimination rules for the unit type, boxes, and products are straightforward:

$$\frac{}{\Gamma \vdash \star : \mathsf{Unit}}\,\textsc{Unit-I} \qquad \frac{\Gamma \vdash t : \mathsf{Unit} \quad \Gamma \vdash u : A}{\Gamma \vdash \mathbf{let}\ \star = t\ \mathbf{in}\ u : A}\,\textsc{Unit-E}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash [^p t] : \Box_p A}\,\textsc{Box-I} \qquad \frac{\Gamma \vdash t : \Box_p A \quad \Gamma, A \vdash u : B}{\Gamma \vdash \mathbf{let}\ [^p] = t\ \mathbf{in}\ u : B}\,\textsc{Box-E}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash (t, u) : A \otimes B}\,\textsc{Prod-I} \qquad \frac{\Gamma \vdash t : A \otimes B \quad \Gamma, A, B \vdash u : C}{\Gamma \vdash \mathbf{let}\ (,) = t\ \mathbf{in}\ u : C}\,\textsc{Prod-E}$$

Note that the eliminators $\mathbf{let}\ [^p] = t\ \mathbf{in}\ u$ and $\mathbf{let}\ (,) = t\ \mathbf{in}\ u$ do not contain variable names for the newly bound variables because we are using de Bruijn indices.

Below are the rules for the Lin type, which correspond to the operations of Linearly constraint [30] from Section 2.3.6:

$$\frac{\Gamma, \mathsf{Lin} \vdash k : !A}{\Gamma \vdash \mathbf{linearly}\ k : !A} \qquad \frac{\Gamma \vdash l : \mathsf{Lin}}{\Gamma \vdash \mathbf{consume}\ l : \mathsf{Unit}} \qquad \frac{\Gamma \vdash l : \mathsf{Lin}}{\Gamma \vdash \mathbf{duplicate}\ l : \mathsf{Lin} \otimes \mathsf{Lin}}$$

For the Lin type, we have **linearly** k for introducing a Lin value inside a continuation k, and its comonoidal operations: **consume** l and **duplicate** l. Lin allows resources (arrays) to be introduced and tracked linearly.

$$\frac{\Gamma \vdash l : \mathsf{Lin} \quad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathbf{new}\ l\ n : \mathsf{Arr}} \qquad \frac{\Gamma \vdash a : \mathsf{Arr} \quad \Gamma \vdash i : \mathbb{N}}{\Gamma \vdash \mathbf{read}\ a\ i : \mathsf{Arr} \otimes\, !\mathbb{N}}$$

$$\frac{\Gamma \vdash a : \mathsf{Arr} \quad \Gamma \vdash i : \mathbb{N} \quad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathbf{write}\ a\ i\ n : \mathsf{Arr}} \qquad \frac{\Gamma \vdash a : \mathsf{Arr}}{\Gamma \vdash \mathbf{free}\ a : \mathsf{Unit}}$$

The array operations are **new** $l\ n$ for creating an array of size n containing zeros, the linear Lin argument l ensures that the array is linear. **read** $a\ i$ for reading from an array a at index i, **write** $a\ i\ v$ for writing a value v to an array a at index i, and **free** a for freeing an array. Compared to Figure 2.5, Lin is explicit, and we do not model read/write capabilities, although the formalization could be extended to include them.

3.2 Usage rules

The usage rules govern how resources are allowed to be used in our system. A term t is *well-resourced* iff it satisfies the relation $\gamma \blacktriangleright t$ where γ is a modality context (a map from variables to modalities, see Definition 2.4.3). The rules follow previous conventions from Abel and Bernardy [31] and Abel *et al.* [16].

Remark. Compared to the rules presented for linear types Section 2.3.3, the usage rules are defined separately from the typing rules, and the use of a modality semiring allows us to track usages of variables in a more general way.

Starting again with the core λ-calculus, we have:

$$\frac{}{\mathbf{e}_i \blacktriangleright x_i} \qquad \frac{\gamma, p \blacktriangleright t}{\gamma \blacktriangleright \lambda^p t} \qquad \frac{\gamma \blacktriangleright t \quad \delta \blacktriangleright u}{\gamma + p\delta \blacktriangleright t^p u}$$

Here $\mathbf{e}_i$ represents a unit modality context, where all variables are mapped to 0, except for the i-th variable which is mapped to 1. The annotation p on the lambda abstraction, requires the body t to use the argument with grade p. The application rule accounts for every use of the argument u in the body t of the lambda by multiplying the resources δ of the argument by p: $p\delta$.

Next, we have the naturals:

$$\frac{}{\mathbf{0} \blacktriangleright \mathbf{zero}} \qquad \frac{\gamma \blacktriangleright t}{\gamma \blacktriangleright \mathbf{suc}\ t} \qquad \frac{\gamma \blacktriangleright z \quad \delta, p \blacktriangleright s \quad \eta \blacktriangleright n}{(\gamma \wedge \delta) + p\eta \blacktriangleright \mathbf{natcase}_p\ z\ s\ n : A}$$

The constructors simply reuse the resources of their arguments (**zero** has none, so its usage is **0**). The eliminator uses the meet operation to account for the resources used in both branches, also adding the resources $p\eta$ used by the scrutinee n.

The rules for the unit type are straightforward.

$$\frac{}{\mathbf{0} \blacktriangleright \star} \qquad \frac{\gamma \blacktriangleright t \quad \delta \blacktriangleright u}{\gamma + \delta \blacktriangleright \mathbf{let}\ \star = t\ \mathbf{in}\ u}$$

The usage rules for the box allow us to capture a value with a specific modality.

$$\frac{\gamma \blacktriangleright t}{p\gamma \blacktriangleright [^p t]} \qquad \frac{\gamma \blacktriangleright t \quad \delta, p \blacktriangleright u}{\gamma + \delta \blacktriangleright \mathbf{let}\ [^p] = t\ \mathbf{in}\ u}$$

Introducing a box demands that there are enough resources to capture the value t with grade p. Meanwhile, the elimination rule allows the continuation u to use the captured value x with the same grade p. For instance, taking linearity as the modality semiring, if we need to construct a box $\square_\omega A$, this restricts us from using values of 1 or 0 grade. However, once we have the box, even if it is linear (its grade is 1), we can unpack it and use the inner value without any restrictions.

Crucially, the product type behaves differently to the ordinary product type in a non-linear/non-graded setting:

$$\frac{\gamma \blacktriangleright t \quad \delta \blacktriangleright u}{\gamma + \delta \blacktriangleright (t, u)} \qquad \frac{\gamma \blacktriangleright t \quad \delta, 1, 1 \blacktriangleright u}{\gamma + \delta \blacktriangleright \mathbf{let}\ (,) = t\ \mathbf{in}\ u}$$

Since the constructor consumes the resources of both elements, the elimination rules requires that both elements are also consumed. This prevents us from having projection functions that would allow us to use only one of the elements[5]. This is also why the tensor product notation ($\otimes$) is used instead of the product one ($\times$).

The rules for the Lin type and arrays are below.

$$\frac{\gamma, 1 \blacktriangleright t}{\gamma \blacktriangleright \mathbf{linearly}\ t} \qquad \frac{\gamma \blacktriangleright t}{\gamma \blacktriangleright \mathbf{consume}\ t} \qquad \frac{\gamma \blacktriangleright t}{\gamma \blacktriangleright \mathbf{duplicate}\ t} \qquad \frac{\gamma \blacktriangleright t \quad \delta \blacktriangleright u}{\gamma + \delta \blacktriangleright \mathbf{new}\ t\ u}$$

$$\frac{\gamma \blacktriangleright t \quad \delta \blacktriangleright u}{\gamma + \delta \blacktriangleright \mathbf{read}\ t\ u} \qquad \frac{\gamma \blacktriangleright t \quad \delta \blacktriangleright u \quad \eta \blacktriangleright v}{\gamma + \delta + \omega\eta \blacktriangleright \mathbf{write}\ t\ u\ v} \qquad \frac{\gamma \blacktriangleright t}{\gamma \blacktriangleright \mathbf{free}\ t}$$

In the rules for write, η is multiplied by ω since elements in the array have to be unrestricted—we allow multiple reads to the same element.

Last, but not least, we have the subsumption rule which allows us to approximate the resources of a term:

$$\frac{\gamma \blacktriangleright t \quad \delta \leq \gamma}{\delta \blacktriangleright t}$$

This corresponds to going down the semilattice structure of the modality semiring, as in Figure 2.6.

[5]Note that defining projections is possible when elements are wrapped with a !. In linear logic terms, this is a way to encode *strong* (&) products by the use of *weak* ($\otimes$) products.

3.3 Example

As an example, we can rewrite the counter program (presented in Section 2.1) in our language:

```
λ₁l. λ₁n.
let₁ count = λ₁ p.
let (arr, i) = p in
let (arr, c) = read arr i in
let [ω c] = c in
let₁ arr = write arr i (suc c) in
(arr, suc c)
in (new l n, count)
```

Here we used explicit names rather than de Bruijn indices for clarity. Writing this directly would result in the following:

```
λ₁ λ₁
let₁ λ₁
let (,) = 0 in
let (,) = read 1 0 in
let [ω] = 1 in
let₁ (write 2 3 (suc 0)) in
(0, suc 1)
in (new 2 1, 0)
```

Let's call the above term t. It is well-resourced ($\mathbf{0} \blacktriangleright t$), and well-typed:

$$\varepsilon \vdash t : \mathsf{Lin} \multimap \mathbb{N} \multimap \mathsf{Arr} \otimes (\mathsf{Arr} \otimes \mathbb{N} \multimap \mathsf{Arr} \otimes \mathbb{N}).$$

Since the internal state (the array) in the counter is always linear, we do not have the problem of the state being potentially used multiple times. Although in our language we do not have the ability to create private constructors (or any kind of custom data types, for that matter), and the state has to remain exposed.

3.4 Heap semantics

To reason about the semantics of our language, we define an operational model based on heaps. We adapt the model from Sestoft [35], which defines an abstract machine for a lazy language[6]. Furthermore, taking inspiration from Marshall *et al.* [24], we adapt the model to work with mutable array references.

We first define renamings and values.

[6]The Agda formalization of the heap model was based on the work done by Oskar Eriksson extending the formalization of Abel *et al.* [16].

Definition 3.4.1 (Renamings)**.** A renaming Ren Γ Δ is an injective map from variables in Δ to variables in Γ, defined as follows:

$$\frac{}{\varepsilon \in \mathsf{Ren}\ \Gamma\ \varepsilon} \qquad \frac{\rho \in \mathsf{Ren}\ \Gamma\ \Delta \quad x : A \in \Gamma \quad x \notin \rho}{\rho, x \in \mathsf{Ren}\ \Gamma\ (\Delta, A)}$$

where $x \notin \varepsilon$ and $x \notin (\rho, y) \iff x \neq y \wedge x \notin \rho$.

Importantly, the definition of renamings does not ensure that variables are renamed in an order-preserving way which is unlike other standard tools such as OPEs (order-preserving embeddings). This allows us to model arrays being allocated and freed in any order.

Definition 3.4.2 (Values)**.**

$$\frac{}{\mathsf{Value}\ (\lambda^p\ t)} \qquad \frac{}{\mathsf{Value}\ \mathbf{zero}} \qquad \frac{\mathsf{Value}\ t}{\mathsf{Value}\ (\mathbf{suc}\ t)} \qquad \frac{}{\mathsf{Value}\ \star} \qquad \frac{\mathsf{Value}\ t}{\mathsf{Value}\ [^p t]}$$

$$\frac{\mathsf{Value}\ t_1 \quad \mathsf{Value}\ t_2}{\mathsf{Value}\ (t_1, t_2)} \qquad \frac{x : \mathsf{Arr} \in \Gamma}{\mathsf{Value}\ x} \qquad \frac{x : \mathsf{Lin} \in \Gamma}{\mathsf{Value}\ x}$$

The value predicate is used to restrict terms that are saved in the heap.

Definition 3.4.3 (Heap)**.** A heap $H \in \mathsf{Heap}\ \Gamma$ is a mapping from variables to heap objects (defined below) and their associated grades.

$$\frac{}{\varepsilon \in \mathsf{Heap}\ \varepsilon} \qquad \frac{H \in \mathsf{Heap}\ \Gamma \quad o \in \mathsf{HeapObject}\ \Gamma\ A}{H, \mapsto_p o \in \mathsf{Heap}\ (\Gamma, A)}$$

Definition 3.4.4 (Heap object)**.** For a context Γ and type A, a heap object $o \in \mathsf{HeapObject}\ \Gamma\ A$ is defined as follows:

$$\frac{\Delta \vdash v : A \quad \mathsf{Value}\ v \quad E \in \mathsf{Ren}\ \Gamma\ \Delta}{\mathsf{value}\ v\ E \in \mathsf{HeapObject}\ \Gamma\ A}\ \text{H-Val}$$

$$\frac{}{[n_0, \ldots, n_{k-1}] \in \mathsf{HeapObject}\ \Gamma\ \mathsf{Arr}}\ \text{H-Arr} \qquad \frac{}{\lightning \in \mathsf{HeapObject}\ \Gamma\ A}\ \text{H-Erased}$$

$$\frac{}{\mathsf{lin} \in \mathsf{HeapObject}\ \Gamma\ \mathsf{Lin}}\ \text{H-Lin}$$

H-Val takes care of the values in the heap that additionally contain the environment (renaming to the heap context). H-Arr is used for arrays, where $n_0, \ldots, n_{k-1}$ are the elements of the array of size k. Since we do evaluate erased terms (a term applied to a function with grade 0), H-Erased is used to represent such terms. H-Lin represents the source of the linear constraint Lin, which is added to the heap when evaluating **linearly**.

Finally, we need to define what kind of heaps are well-resourced. This is realized with the following relation:

Definition 3.4.5 (Well-resourced heap)**.** A heap H is well-resourced if and only if $\gamma \blacktriangleright H$, which says that the heap H contains γ resources.

$$\frac{}{\mathbf{0} \blacktriangleright \varepsilon} \qquad \frac{\gamma + p\delta \blacktriangleright H \quad \delta \blacktriangleright_p o}{\gamma, p \blacktriangleright H, \mapsto_p o}$$

Obviously, an empty heap does not contain any resources. For a heap extension, we require that the object o is well-resourced under the preceding heap H. As a result, some resources of H may be consumed $p\delta$, leaving the rest γ together with a new resource p.

Definition 3.4.6 (Well-resourced heap objects)**.** For a modality context γ, modality p, and heap object o, the relation for well-resourced heap objects $\gamma \blacktriangleright_p o$ is defined as follows:

$$\frac{\gamma \blacktriangleright t}{\mathsf{ren}\ E\ \gamma \blacktriangleright_p \mathsf{value}\ t\ E} \qquad \frac{p \in \{0,1\}}{\mathbf{0} \blacktriangleright_p [n_0, \ldots, n_{k-1}]} \qquad \frac{p \in \{0,1\}}{\mathbf{0} \blacktriangleright_p \mathsf{lin}} \qquad \frac{}{\mathbf{0} \blacktriangleright_0 \lightning}$$

In the rule for values, the modality context has to be adapted to the context of the preceding heap by renaming it.

Additionally, there are a few operations that can be performed on heaps:

Definition 3.4.7 (Heap lookup)**.** Looking up a variable (or reference) x in the heap H, resulting in an object o and the heap H', is captured by the relation $H \vdash x \mapsto^p_q o \mid H'$, defined below.

$$\frac{p - q \equiv r}{H, \mapsto_p o \vdash 0 \mapsto^p_q (\uparrow o) \mid H, \mapsto_r o} \qquad \frac{H \vdash x \mapsto^p_q o \mid H'}{H, \mapsto_{p'} o' \vdash x + 1 \mapsto^p_q (\uparrow o) \mid H', \mapsto_{p'} o'}$$

Here p is the previous grade of the variable x in H, while q is the grade of the lookup's request (how much should be subtracted from p).

Since the context Γ of the heap $H \in \mathsf{Heap}\ \Gamma$ has to match the context of the looked up object $o \in \mathsf{HeapObject}\ \Gamma\ A$, we weaken the object each time with the $\uparrow$ operation.

Additionally, we introduce a few shorthand notations for the heap lookup:

$$\begin{aligned}
(H \vdash x \mapsto o) &\stackrel{\text{def}}{=} (\exists p.\ H \vdash x \mapsto^p_0 o \mid H) \\
(H \vdash x \mapsto^p) &\stackrel{\text{def}}{=} (\exists o.\ H \vdash x \mapsto^p_0 o \mid H) \\
(H \vdash x \mapsto^p o) &\stackrel{\text{def}}{=} (H \vdash x \mapsto^p_0 o \mid H) \\
(H \vdash x \mapsto_q o \mid H') &\stackrel{\text{def}}{=} (\exists p.\ H \vdash x \mapsto^p_q o \mid H')
\end{aligned}$$

The relation $p - q \equiv r$ is used to reduce the grade of resources in the heap. As the structure for modalities forms a semiring (instead of a *ring*), we do not have a general subtraction operation. Instead, we take the definition of subtraction[7] from the graded-type-theory [16] formalization project.

[7] Defined in `Graded.Modality.Properties.Subtraction`.

Definition 3.4.8 (Subtraction in a semiring)**.**

$$(p - q \equiv r) \stackrel{\text{def}}{=} (p - q \leq r) \wedge (\forall s.\ (p - q \leq s) \Rightarrow r \leq s)$$

where $(p - q \leq r) \stackrel{\text{def}}{=} (p \leq r + q)$.

We require that subtraction is supported in the chosen modality semiring, meaning that the following proposition holds for any p, q, r grades:

$$(p - q \leq r) \Rightarrow \exists r'.p - q \equiv r'.$$

As explained to me by Oskar Eriksson, the above definition can be explained from a quantitative perspective: $p - q \equiv r$ means that if we have p copies of some resource and use q copies, we will have r copies remaining (where r is chosen in such a way that leaves as many resources as possible). Choudhury *et al.* [36] also use subtraction-like operation to manage resources in a heap, however their definition is non-deterministic, contrary to the one presented above.

Note that both linearity and affine modalities support subtraction. For linearity, the operation behaves in the following way:

$(-)$	0	1	ω
0	0		
1	1	0	
ω	ω	ω	ω

Lemma 3.4.1 (Lookup modifies the grade)**.** *If* $p - q \equiv r$ *and* $H \vdash x \mapsto^{p}_{q} o \mid H'$*, then* $H \vdash x \mapsto^{r} o$*.*

Lemma 3.4.2 (Unrelated lookup preserves grades)**.** *If* $H \vdash x \mapsto^{p'}_{q} o \mid H'$ *and* $H \vdash y \mapsto^{p} o$*, but* $x \neq y$*, then* $H' \vdash y \mapsto^{p} o$*.*

To model mutable arrays, an operation to update an array in the heap is needed, which is defined below.

Definition 3.4.9 (Heap update for arrays)**.** Replacing the array under the reference a (in the heap H) with a new array xs, resulting in the heap H', is denoted by $H \vdash a := xs \mid H'$, with the following definition:

$$\frac{}{H, 0 \mapsto_1 [n_0, \ldots, n_k] \vdash a := [m_0, \ldots, m_k] \mid H, a \mapsto_1 [m_0, \ldots, m_k]}$$

$$\frac{H \vdash a := [m_0, \ldots, m_k] \mid H'}{H, \mapsto_r o \vdash a + 1 := [m_0, \ldots, m_k] \mid H', \mapsto_r o}$$

Remark. The relation could be further restricted to only allow updates of an element in an array (given the element's index and its new value).

Lemma 3.4.3 (Updated array is live)**.** *If* $H \vdash x := xs \mid H'$*, then* $H \vdash x \mapsto^{1} xs$*.*

Lemma 3.4.4 (Unrelated update preserves grades)**.** *If* $H \vdash x := xs \mid H'$ *and* $H \vdash y \mapsto^{p} o$*, but* $x \neq y$*, then* $H' \vdash y \mapsto^{p} o$*.*

Having defined the heap, we can now define the abstract call-by-value machine. The machine operates on states, containing the heap, the term being evaluated, and the surrounding evaluation context.

Remark. Call-by-value was chosen as a middle-ground option between call-by-name and call-by-need evaluation strategies. Call-by-name for heap semantics in graded modal type theory was explored before [31], however it does not work well in terms of mutability. Call-by-need would be obvious choice, given that Haskell, being the main focus in previous work [5], [30], uses call-by-need. However, call-by-value is much simpler, and does not conflict with our original motivation of modelling pure mutable arrays.

Definition 3.4.10 (State)**.** A state $(H, t, E, S) \in \mathsf{State}\ \Gamma\ \Delta\ A\ B$ consists of

- a heap $H \in \mathsf{Heap}\ \Gamma$ mapping variables to values and references to arrays,
- a head term $\Delta \vdash t : A$ (the subject for reduction),
- an environment $E \in \mathsf{Ren}\ \Gamma\ \Delta$ renaming variables (and references) from Δ to Γ, and
- a stack $S \in \mathsf{Stack}\ \Gamma\ A\ B$ of eliminators.

An eliminator $\mathsf{Elim}\ \Gamma\ A\ B$ is defined as a singular "term" constructor (in context Δ and of type B) containing a single hole (of type A), together with its environment $\mathsf{Ren}\ \Gamma\ \Delta$. The full definition can be found in the Agda formalization.

Definition 3.4.11 (Stack)**.** A stack $S \in \mathsf{Stack}\ \Gamma\ A\ B$ is a sequence of eliminators such that their types "match-up": for each eliminator in the sequence, its type is equal to the type of the hole of the preceding eliminator (if there is one).

$$\frac{}{\varepsilon \in \mathsf{Stack}\ \Gamma\ A\ A} \qquad \frac{S \in \mathsf{Stack}\ \Gamma\ A\ B \quad e \in \mathsf{Elim}\ \Gamma\ B\ C}{e, S \in \mathsf{Stack}\ \Gamma\ A\ C}$$

Now, we define when a state is well-resourced.

Definition 3.4.12 (Well-resourced state)**.** A state (H, t, E, S) is well-resourced if and only if the heap H has the enough resources needed for the term t and the stack S (which represents the surrounding evaluation context):

$$\frac{\gamma \blacktriangleright H \quad \delta \blacktriangleright t \quad \eta \blacktriangleright S \quad \gamma \leq |S| \cdot \mathsf{ren}\ E\ \delta + \eta}{\gamma\,;\delta\,;\eta \blacktriangleright (H, t, E, S)}$$

The relation for well-resourced stacks as well as the modality of a stack $|S|$ is defined below. The last premise, in the above rule, ensures that the resources of the heap H are enough to provide the resources needed in the term t and the stack S.

Definition 3.4.13 (Well-resourced stack)**.**

$$\frac{}{\mathbf{0} \blacktriangleright \varepsilon} \qquad \frac{\gamma \blacktriangleright S \quad |S| \cdot \delta \blacktriangleright e}{\gamma + \delta \blacktriangleright e, S}$$

The definition of well-resourced eliminators $\gamma \blacktriangleright^e e$ is not included, but it follows the definition of well-resourced terms, skipping the resources of the hole in the eliminator.

Definition 3.4.14 (Modality of an eliminator/stack)**.**

$$\begin{aligned} |\varepsilon| &= 1 & |t^p - | &= p \\ |\textbf{linearly} -, S| &= 1 & |[^p -]| &= p \\ |e, S| &= |e| \cdot |S| & |e| &= 1 \end{aligned}$$

For the eliminators, the dash $-$ marks the hole in the term. When evaluating an argument we put the eliminator $t^p -$ on the stack, and multiply the current context modality with the annotated modality p. The same applies for evaluating under the box $[^p -]$. The grade of the stack is reset upon encountering the **linearly** $-$ eliminator to make consume/duplicate/new rules go through. This works because the result of the continuation of linearly is unrestricted—the result is well-resourced under any grade even though it was originally produced in the 1-grade stack. Note that the grade of a well-resourced eliminator is never 0, i.e., $|t^p - |$ and $[^p -]$ are well-resourced only if $p \neq 0$. This gives rise to the following lemma:

Lemma 3.4.5 (No erased evaluation)**.** *If a stack S is well-resourced $\gamma \blacktriangleright S$, then $|S| \not\equiv 0$.*

The operational semantics have three variants:

1. **Ungraded** ($\rightsquigarrow_{\mathsf{u}}$): Basic semantics that do not modify the grades in the heap.
2. **Pure** ($\rightsquigarrow_{\mathsf{p}}$): Graded semantics where arrays are copied when writing to them.
3. **Mutable** ($\rightsquigarrow_{\mathsf{m}}$): Graded semantics where arrays are updated destructively in-place.

For variables, we have the following reductions:

$$\frac{H \vdash \mathsf{ren}\ E\ x \mapsto \mathsf{value}\ v\ E'}{(H, x, E, S) \rightsquigarrow_{\mathsf{u}} (H, v, E', S)}\ \textsc{U-Var} \qquad \frac{H \vdash \mathsf{ren}\ E\ x \mapsto_{|S|} \mathsf{value}\ v\ E' \mid H'}{(H, x, E, S) \rightsquigarrow_{\mathsf{p,m}} (H', v, E', S)}\ \textsc{G-Var}$$

The ungraded rule simply checks that the variable exists in the heap, while the graded rule (pure and mutable semantics) requires that the variable in the heap has enough resources.

The reduction rules for applications are a bit more involved:

$$\frac{}{(H, t^p u, E, S) \rightsquigarrow (H, t, E, (-^p u\ E, S))}\ \textsc{App}_1$$

$$\frac{}{(H, \lambda^0\ t, E', (-^0 u\ E, S)) \rightsquigarrow ((H, \mapsto_0 \lightning), t, \Uparrow E, \uparrow S)}\ \textsc{App}_{2e}$$

$$\frac{p \not\equiv 0}{(H, \lambda^p\ t, E', (-^p u\ E, S)) \rightsquigarrow (H, u, E, ((\lambda^p\ t)^p -\ E', S))}\ \textsc{App}_2$$

$$\frac{\mathsf{Value}\ u}{(H, u, E'', ((\lambda^p\ t)^p -\ E', S)) \rightsquigarrow ((H, \mapsto_{p|S|} \mathsf{value}\ u\ E''), t, \Uparrow E', \uparrow S)}\ \textsc{App}_3$$

The first rule $\textsc{App}_1$ pushes the argument (together with the current environment) on the stack, and the left side of the application is left as the head term. When it is reduced to a

lambda abstraction, there is a choice between two reductions—if the grade annotation is 0, then the argument is erased in the $\textsc{App}_{2e}$ rule. Otherwise, the $\textsc{App}_2$ rule is applied: the argument is popped from the stack, and after it is evaluated, the third rule $\textsc{App}_3$ is applied. The final rule adds the evaluated argument on the heap, and proceeds with the body of the lambda. $\Uparrow$ and $\uparrow$ denote the operations for lifting and weakening, respectively.

The rules for **linearly** are as follows:

$$\frac{}{(H, \textbf{linearly}\ k, E, S) \rightsquigarrow_{\mathsf{u}} ((H, \mapsto_1 \mathsf{lin}), k, \Uparrow E, (\textbf{linearly}\ 0, \uparrow S))}\ \textsc{Linearly}_1$$

$$\frac{H \vdash x \mapsto \mathsf{lin} \quad \mathsf{Value}\ k}{(H, k, E, (\textbf{linearly}\ x, S)) \rightsquigarrow_{\mathsf{u}} (H, k, E, S)}\ \textsc{Linearly}_2$$

The first rule adds the lin token to the heap with grade 1. Note that the rule does not mention the grade of the stack $|S|$, so that linearly can be called from unrestricted contexts, i.e. $!\,\textbf{linearly}\ k$ is fine since the continuation will be evaluated only once and the lin token cannot escape its scope. The final two rules remove the **linearly** x eliminator from the stack.

Remark. The last rule could be made more precise by checking that lin has grade 0, which in the context of linearity would mean that the token has been consumed. However, this would require a more complex definition of the heap, and the current definition is sufficient for the purposes of this work.

The full rules can be found in Appendix A, but as a final example, we can look at the last step of evaluating the term **write** $a\ i\ v$. The reduction evaluates the arguments from right to left, so in the last step we will have reduced a, and the final rule will apply the write operation to the array in the heap.

$$\frac{H \vdash \mathsf{ren}\ E\ a \mapsto xs \quad xs' = (xs[i] := v)}{(H, a, E, (\textbf{write}\ \_\ i\ v, S)) \rightsquigarrow_{\mathsf{u}} ((H, \mapsto_{\mathbf{1}} xs'), 0, E, S)}\ \text{U-}\textsc{Write}_4$$

$$\frac{H \vdash \mathsf{ren}\ E\ a \mapsto_1 xs \mid H' \quad xs' = (xs[i] := v)}{(H, a, E, (\textbf{write}\ \_\ i\ v, S)) \rightsquigarrow_{\mathsf{p}} ((H', \mapsto_{\mathbf{1}} xs'), 0, E, S)}\ \text{P-}\textsc{Write}_4$$

$$\frac{H \vdash \mathsf{ren}\ E\ a := xs' \mid H' \quad xs' = (xs[i] := v)}{(H, a, E, (\textbf{write}\ \_\ i\ v, S)) \rightsquigarrow_{\mathsf{m}} (H', a, E, S)}\ \text{M-}\textsc{Write}_4$$

The ungraded and pure semantics are copying, thus with every write operation a new array is allocated on the heap. However, in the case of pure semantics, the grade of the original array is checked and subtracted by 1, meaning that the array cannot be of grade 0. In the mutable semantics, the array is updated in-place, without additional allocations. $xs[i] := v$ sets the element in the array xs at index i to v, note that the semantics get stuck if i is out of bounds for the array xs.

3.5 Bisimilarity

The goal of the overall formalization is to show that the three (ungraded, pure, mutable) semantics are bisimilar to one another. In the context of our language, bisimilarity states that starting with related states, a reduction step in one semantics can be matched by a reduction step in the other semantics, with the resulting states staying related. This ensures that different semantic interpretations of our language are consistent with each other.

To state bisimilarity, we first need to establish a relation between the states of the two semantics. Since we have three reduction semantics, we will define two bisimilarity relations:

1. one between ungraded and pure states, and
2. one between pure and mutable states.

Proving bisimilarity for the two relations is enough to show that all semantics are bisimilar to one another, since we can compose the two relations (as well as their bisimilarity proofs) together.

Remark. Note that whenever we refer to states we assume that they are well-resourced.

3.5.1 Between ungraded and pure semantics

To relate ungraded and pure semantics we need to establish that reduction without considering grades can be matched by reduction with grades. We define a relation between the heaps of the two semantics, and then show that related states remain related after a reduction step.

Definition 3.5.1 (Heap equality up to grades)**.**

$$\frac{}{\varepsilon \sim \varepsilon} \qquad\qquad \frac{H \sim H'}{H, \mapsto_p o \sim H', x \mapsto_q o}$$

Definition 3.5.2 (State equality up to grades)**.** The relation $s_u \sim s_p$ between states s_u, s_p of the ungraded and pure semantics, respectively, simply relates the two heaps up to grades.

$$\frac{H \sim H'}{(H, t, E, S) \sim (H', t, E, S)}$$

Lemma 3.5.1 (Reflexivity)**.** $s \sim s$.

Theorem 3.5.2 (Bisimilarity between ungraded and pure semantics)**.** *For all* $s_u \sim s_p$,

- *given any* s'_u *with* $s_u \rightsquigarrow_u s'_u$, *there exists* s'_p *such that* $s_p \rightsquigarrow_p s'_p$ *and* $s'_u \sim s'_p$;
- *given any* s'_p *with* $s_p \rightsquigarrow_p s'_p$, *there exists* s'_u *such that* $s_u \rightsquigarrow_u s'_u$ *and* $s'_u \sim s'_p$.

3.5.2 Between pure and mutable semantics

Given contexts Γ_p, Δ_p (pure) and Γ_m, Δ_m (mutable), we define a relation $s_p \approx s_m$ between states $s_p \in \mathsf{State}\ \Gamma_p\ \Delta_p\ A\ B$ and $s_m \in \mathsf{State}\ \Gamma_m\ \Delta_m\ A\ B$ of the pure and mutable semantics, respectively.

Definition 3.5.3 (State equality up to renaming)**.**

$$\frac{H \approx_\rho H' \quad \mathsf{ren}\ E\ t \equiv \mathsf{ren}\ \rho\ (\mathsf{ren}\ E'\ t') \quad S \equiv \mathsf{ren}\ \rho\ S'}{(H, t, E, S) \approx (H', t', E', S')}$$

Lemma 3.5.3 (Reflexivity)**.** $s \approx_{id} s$.

For two heaps $H_p \in \mathsf{Heap}\ \Gamma_p$, $H_m \in \mathsf{Heap}\ \Gamma_m$, and a renaming $\rho \in \mathsf{Ren}\ \Gamma_p\ \Gamma_m$, the heaps are related under the renaming $H_p \approx_\rho H_m$ if and only if for all $x_p : A \in \Gamma_p$, x_p is either *dead* or *shared*.

A variable $x_p : \mathsf{Arr} \in \Gamma_p$ is *dead* if it has grade 0 in the heap ($H_p \vdash x_p \mapsto^0$) and there is no $x_m : Arr \in \Gamma_m$ such that $\mathsf{ren}\ \rho\ x_m = x_p$. Meanwhile, $x_p : A \in \Gamma_p$ is *shared* iff there is $x_m : A \in \Gamma_m$ with $\mathsf{ren}\ \rho\ x_m = x_p$ and objects $o_m \in \mathsf{HeapObject}\ \Gamma_m\ A$ with $o_p \equiv ren\ \rho\ o_m$ such that $H_m \vdash x_m \mapsto^p o_m$ and $H_p \vdash x_p \mapsto^p o_p$.

Now, we can state the bisimilarity on the above relation. Since the proof is not fully formalized yet, we mark it as a conjecture.

Conjecture 3.5.4 (Bisimilarity between pure and mutable semantics)**.** *For all $s_p \approx s_m$,*

- *given any s'_p with $s_p \rightsquigarrow_p s'_p$, then there exists s'_m such that $s_m \rightsquigarrow_m s'_m$ and $s'_p \approx s'_m$;*
- *and given any s'_m with $s_m \rightsquigarrow_m s'_m$, then there exists s'_p such that $s_p \rightsquigarrow_p s'_p$ and $s'_p \approx s'_m$.*

Remark. The main missing piece in the partially formalized proof is in the final write rule (P-Write$_4$/M-Write$_4$). We want to show that we can remap the array variable from the old location to the top of the context, in accordance to the copying semantics. For that to work, we need to prove that the objects in the heap and eliminators on the stack do not reference the array. This should not be difficult to formalize since Definition 3.4.12 ensures that if a linear resource (the array) is consumed in the head term of the state, then neither the heap nor the stack is allowed to consume it.

3.5.3 Combining the relations

Finally, to show that all semantics are bisimilar to one another, we combine them together:

Definition 3.5.4 (Equality between states of ungraded and mutable semantics)**.**

$$\frac{s_u \sim s_p \quad s_p \approx s_m}{s_u \approx' s_m}$$

Now, combining Theorem 3.5.2 with Conjecture 3.5.4, we get bisimilarity[8] between all three semantics:

[8] This is yet to be formalized.

Conjecture 3.5.5 (Bisimilarity)**.** *For all s_u, s_p, s_m with $s_u \sim s_p$ and $s_p \approx s_m$, the following statements are equivalent:*

- *given any s'_u with $s_u \rightsquigarrow_u s'_u$, then there exists s'_m such that $s_m \rightsquigarrow_m s'_m$ and $s'_u \approx' s'_m$;*
- *given any s'_m with $s_m \rightsquigarrow_m s'_m$, then there exists s'_u such that $s_u \rightsquigarrow_u s'_u$ and $s'_u \approx' s'_m$.*

Remark. Both directions follow from composing Theorem 3.5.2 with Conjecture 3.5.4. But since one component is marked as a conjecture, the combined bisimilarity is also a conjecture.

From the above conjecture, we can also conclude that evaluating a numeral in different semantics produces the same value:

Conjecture 3.5.6. *Given a well-resourced closed term $\varepsilon \vdash t : \mathbb{N}$, if t evaluates to a value in one semantics, then it evaluates to the same value in the other semantics. That is, the following statements are equivalent, and values v_u, v_p, v_m are all equal:*

- $(\varepsilon, t, \varepsilon, \varepsilon) \rightsquigarrow_u^* (H_u, v_u, E_u, \varepsilon)$ *with* Value v_u,
- $(\varepsilon, t, \varepsilon, \varepsilon) \rightsquigarrow_p^* (H_p, v_p, E_p, \varepsilon)$ *with* Value v_p,
- $(\varepsilon, t, \varepsilon, \varepsilon) \rightsquigarrow_m^* (H_m, v_m, E_m, \varepsilon)$ *with* Value v_m.

Remark. The above conjecture is not yet formalized.

The multi-step reduction $\rightsquigarrow^*$ used above is defined as follows:

Definition 3.5.5 (Transitive closure of the reduction relation)**.**

$$\frac{}{s \rightsquigarrow^* s} \qquad \frac{s \rightsquigarrow^* s' \quad s' \rightsquigarrow s''}{s \rightsquigarrow^* s''}$$

4

Discussion

In this chapter, we revisit the core concepts discussed throughout the thesis. We explore extensions, and alternatives found in related work.

4.1 Compilation step

Currently, the semantics of the presented language deals with primitives related to Lin: linearly, consume, and duplicate. However, these high-level primitives only exist to ensure safety of the source language, and, ideally, they would be optimized away in the machine code. To formalize this, we would need to define a compilation step that translates the high-level language to a low-level language with mutability that does not contain these primitives. This would involve defining another abstract machine (operating on the low-level language), and proving that the compilation step preserves the semantics of the high-level language.

Furthermore, one could add a translation step to verify that mutable array operations preserve resources (e.g. linearity) of the source language. This could be achieved by translating the array primitives to a pure implementation (with no mutability): arrays could be translated to nested pairs, then read and write operations would simply be projections and updates. Note that if the target language of the translation is the source language itself, the array type would have to be extended to include the size of the array and have some kind of recursion (see Section 4.3). Alternatively, having the target language be untyped would make things simpler, and we would get a guarantee that the types could be erased at runtime. In any case, to prove that the translation preserves the semantics we would need to use a weak bisimilarity allowing us to relate a single step on one side with multiple steps on the other.

4.2 Progress and other properties

The formalization lacks a proof of progress verifying that the semantics do not get stuck for well-resourced programs. Note that this should not be the case in general, since read and write operations fail if the index is out of bounds for the array. The property could be stated as follows:

If $\Gamma \vdash t : A$ and $(\varepsilon, t, \varepsilon, \varepsilon) \rightsquigarrow^* (H, u, E, S)$, then either there is another step to take from (H, u, E, S), or the term u is a value and the stack S is empty, or the top of the stack

contains the final eliminator for read or write where the index points outside the array.

4.3 Extending the language

The language presented in this thesis is a minimalistic core calculus with the ability to encode linearly-style mutable array API, originally presented by Spiwack *et al.* [30]. However, there are several ways in which the language could be extended to support additional features and programming paradigms. One obvious extension would be to add a fixpoint operator to enable recursive functions, which are essential for writing any kind of algorithms. Without recursion, we cannot even iterate through a dynamically-sized array. Furthermore, sum types could be introduced to allow for more expressive data structures. Both of these extensions would not be too difficult to implement, as they do not require any changes to the underlying type system and their reduction rules should be straightforward.

4.4 Call-by-Need

Previous work on defining an abstract machine for graded modal type theory uses a call-by-name evaluation strategy [31]. Mutability does not work well with call-by-name semantics, since an expression to create an array could be substituted in multiple places and potentially allocating multiple arrays, where only one is expected. Instead, call-by-value evaluation strategy was chosen for the abstract machine in the formalization. However, adapting the abstract machine to be call-by-need, would be more in line with the original work (Linear Haskell [5]), thus this poses as an interesting extension to the work presented in this thesis. In particular, this would require extending heaps to store arbitrary terms, not just values.

4.5 Exploring other modalities

The language presented formalization is parameterized by a modality semiring (with some restrictions). It has largely been explored in a context of linearity, but other modality semiring could also be used such affinity. Affinity ensures that resources are used *at most* once (whereas in linearity it is *exactly once*). Relaxing the restriction for the weakening rule allows the array to be silently dropped (perhaps leaving it to be garbage collected by the runtime of the language) which might be easier to use for the programmer while still retaining the necessary safety guarantees. Furthermore, Bernardy *et al.* [5] proposed introducing another grade β to the linearity modality for modelling borrowed types.

4.6 Extending to Dependent Types

Dependent types extend non-dependent type systems by allowing types to depend on values. This enables programs to encode a wide range of program properties directly within the type system. Dependent types can ensure stronger correctness guarantees and facilitate formal verification of program behaviour.

Integrating mutable arrays to a dependently typed language, such as Agda, would allow implementing fast and provably correct algorithms. For example, it could be possible to implement a type-safe matrix library, where the dimensions of matrices are encoded in their types. This would prevent runtime errors caused by mismatched dimensions and enable the compiler to optimize matrix operations based on their sizes. Currently, the only way to reason about correctness in such settings in Agda is through the use of linked list-like structure, which have a much worse complexity compared to arrays.

The presented semantics in Section 3.4 have been designed to be extended to dependent types. In particular, copying (ungraded, pure) variant of the semantics could be used at compile time, while the mutable variant could be used at runtime. This would be somewhat akin to how natural numbers are treated in Agda: at compile time, naturals are represented in a unary way (zero and suc n), but at runtime they become machine integers.

To give a quick example of how mutable arrays could look like in Agda, we can define a simply mutable array API, following the ideas from Linear Haskell and Spiwack *et al.* [30]:

```
linearly   : (Lin ⊸ Ur A) ⊸ Ur A
duplicate : Lin ⊸ Lin ⊗ Lin
consume  : Lin ⊸ ⊤

new    : Lin ⊸ (Vec A n ⊸ Arr A n)
read   : Arr A n ⊸ (Fin n ⊸ Ur A ⊗ Arr A n)
write  : Arr A n ⊸ (Fin n ⊸ (A → Arr A n))
freeze : Arr A n ⊸ Vec A n
```

Now we can equational rules for the API can be defined as follows:

```
linearly-consume : (e : Ur A)
  → linearly (λ l → case consume l of λ { tt → e })
  ≡ e

linearly-duplicate : (f : Lin ⊸ Lin ⊸ Ur A)
  → linearly (λ l  → case duplicate l of λ { ⟨ l₁ , l₂ ⟩ → f l₁ l₂ })
  ≡ linearly (λ l₁ → linearly (λ l₂ → f l₁ l₂))

write-new : (l : Lin) (xs : Vec A n) (i : Fin n) (x : A)
  → write (new l xs) i x
  ≡ new l (xs [ i ]:= x)

read-new : (l : Lin) (xs : Vec A n) (i : Fin n)
  → read (new l xs) i
  ≡ ⟨ mkUr (lookup xs i) , new l xs ⟩

freeze-new : (l : Lin) (xs : Vec A n)
  → freeze (new l xs)
  ≡ case consume l of λ { tt → xs }
```

With these rules, we could reason about the behavior of the program using mutable arrays at the type level and prove properties about them. For instance, we could implement an efficient in-place sorting algorithm and prove that it is correct, all in Agda. This presents another area to explore in the formalization—showing that the above equations hold.

However, the fact that *linearly* takes a continuation is a bit problematic since function extensionality has to be used (to apply rules under a binder), which is not provable in Agda (unless additional extensions such as Cubical Agda are enabled). As an alternative, uniqueness types do not exhibit this problem, since no continuation is necessary:

```
new    : Vec A n → *(Arr A n)
read   : *(Arr A n) → Fin n → A × *(Arr A n)
write  : *(Arr A n) → Fin n → A → *(Arr A n)
freeze : *(Arr A n) → Vec A n
```

The equational rules for the uniqueness-typed API are also simpler:

```
write-new : (xs : Vec A n) (i : Fin n) (x : A)
  → write (new xs) i x
  ≡ new (xs [ i ]:= x)

read-new : (xs : Vec A n) (i : Fin n)
  → read (new xs) i
  ≡ lookup xs i , new xs

freeze-new : (xs : Vec A n)
  → freeze (new xs)
  ≡ xs
```

However, there is still a problem with *necessarily unique functions* (functions that capture a unique variable, see Section 2.3.2) which can pose a challenge.

4.6.1 More flexible arrays

Embedding mutable arrays in a dependently typed language would enable more flexibility in terms of what kind of static guarantees could be tracked at compile-time. For instance, Arend supports[1] immutable dependent arrays, which in Agda's notation would look something like this:

```
record DArray {len : Nat} (A : Fin len → Set) : Set where
  field
    at : (j : Fin len) → A j
```

[1] Array in Arend's Prelude:
https://arend-lang.github.io/documentation/language-reference/prelude.html#array.

4.7 Generalized uniqueness types for arrays

In Section 2.3 we introduced the concept of substructural types, focusing on linearity and uniqueness. Graded modal types were presented as a generalization of linear types, allowing for more flexible resource management. In the context of arrays, uniqueness types can be used to enforce single ownership of an array, preventing aliasing and enabling efficient in-place updates. One question to ask is if we can generalize uniqueness types in the same vain, i.e., encode uniqueness in some modality semiring. This would allow us to experiment with uniqueness types in the same formalization by simply using a uniqueness semiring.

Marshall and Orchard [37] presented a generalization of uniqueness types by grading the uniqueness modality with a fraction, which represents a share of ownership. Fractional ownership can encode Rust's model of shared immutable or exclusive mutable access. Furthermore, a unique type can be split, shared among multiple parts of a program (with a shared immutable access), and later rejoined, which recovers uniqueness (and exclusive mutable access). This approach allows for more fine-grained control over array access and modification, enabling concurrent and parallel programming paradigms while maintaining safety and predictability.

Lorenzen *et al.* [38] present a different kind of generalization, where uniqueness together with affinity and locality are all combined into a single semiring structure—the elements of the semiring are triplets whose elements each represent different substructural axes. This presents potential for a more unified approach to modelling uniqueness, allowing to reuse extensive research devoted on modelling substructural types (like linearity) with a modality semiring structure.

5

Conclusion

In conclusion, this thesis has explored embedding mutable state in a pure functional language, particularly focusing on the challenges and innovations surrounding mutable arrays in programming languages with linear or uniqueness types. By investigating existing solutions like the ST monad in Haskell and recent advancements such as Linear Haskell [5], we highlighted some limitations of different approaches in safely integrating mutable state within a purely functional paradigm.

The presented formalization of a graded lambda calculus with array primitives was intended to demonstrate the correctness of a "linearly"-style [30] mutable array API. However, unfortunately the formalization is incomplete.

This research contributes to the ongoing discourse on incorporating substructural rules to functional programming languages enabling more robust ways of handling resources as well as practical performance benefits while maintaining functional purity.

5.1 Future Work

The presented formalization is a bare-bones core calculus that has just enough features to encode the linearly-style [30] mutable array API. There are several ways in which the formalization could be extended:

- Prove additional properties like progress for the semantics.
- Add a compilation step to translate the high-level language to a low-level language that does not contain linearly-style tokens.
- Add a fixpoint operator to enable recursive programs.
- Add sum types for encoding branching computations.
- Explore other modalities (e.g. affine) to allow for more flexible resource management.
- Investigate the integration of mutable arrays with dependently-types.

Lastly, as mentioned in the disclaimers, I have not proved all theorems that I present in the text. Future work should focus on completing the formalization.

Bibliography

[1] J. Hughes, "Why Functional Programming Matters," *The Computer Journal*, vol. 32, no. 2, pp. 98–107, Jan. 1989, ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.98. eprint: https://academic.oup.com/comjnl/article-pdf/32/2/98/1445644/320098.pdf. [Online]. Available: https://doi.org/10.1093/comjnl/32.2.98.

[2] P. Wadler, "Deforestation: Transforming programs to eliminate trees," *Theoretical Computer Science*, vol. 73, no. 2, pp. 231–248, 1990, ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(90)90147-A. [Online]. Available: https://www.sciencedirect.com/science/article/pii/030439759090147A.

[3] A. Gill, J. Launchbury, and S. L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '93, Copenhagen, Denmark: Association for Computing Machinery, 1993, pp. 223–232, ISBN: 089791595X. DOI: 10.1145/165180.165214. [Online]. Available: https://doi.org/10.1145/165180.165214.

[4] J. Svenningsson, "Shortcut fusion for accumulating parameters & zip-like functions," *SIGPLAN Not.*, vol. 37, no. 9, pp. 124–132, Sep. 2002, ISSN: 0362-1340. DOI: 10.1145/583852.581491. [Online]. Available: https://doi.org/10.1145/583852.581491.

[5] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack, "Linear Haskell: Practical linearity in a higher-order polymorphic language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. DOI: 10.1145/3158093. [Online]. Available: https://doi.org/10.1145/3158093.

[6] "Experience report: Linear Haskell enables pure, parallel, and in-place Fast Fourier Transformation." (2023), [Online]. Available: https://discourse.haskell.org/t/experience-report-linear-haskell-enables-pure-parallel-and-in-place-fast-fourier-transformation/8256 (visited on 07/21/2024).

[7] E. Brady, "Type-driven development of concurrent communicating systems," *Computer Science*, vol. 18, p. 219, Jul. 2017. DOI: 10.7494/csci.2017.18.3.1413.

[8] R. Shi and H. Xi, "A linear type system for multicore programming in ATS," *Science of Computer Programming*, vol. 78, no. 8, pp. 1176–1192, 2013, Special section on software evolution, adaptability, and maintenance & Special section on the Brazilian Symposium on Programming Languages, ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2012.09.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642312001700.

[9] E. Brady, "Idris 2: Quantitative Type Theory in practice," in *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, A. Møller and M. Sridharan, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 194,

Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 9:1–9:26, ISBN: 978-3-95977-190-0. DOI: 10.4230/LIPIcs.ECOOP.2021.9. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2021.9.

[10] C. McBride, "I got plenty o' nuttin'," in *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, S. Lindley, C. McBride, P. Trinder, and D. Sannella, Eds. Cham: Springer International Publishing, 2016, pp. 207–233, ISBN: 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_12. [Online]. Available: https://doi.org/10.1007/978-3-319-30936-1_12.

[11] R. Atkey, "Syntax and semantics of Quantitative Type Theory," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '18, Oxford, United Kingdom: Association for Computing Machinery, 2018, pp. 56–65, ISBN: 9781450355834. DOI: 10.1145/3209108.3209189. [Online]. Available: https://doi.org/10.1145/3209108.3209189.

[12] E. Brady. "Why is Idris 2 so much faster than Idris 1." (2020), [Online]. Available: https://www.type-driven.org.uk/edwinb/why-is-idris-2-so-much-faster-than-idris-1.html (visited on 07/21/2024).

[13] E. Brady. "Source code of idris 2 (`Core.Context`)." (), [Online]. Available: https://github.com/idris-lang/Idris2/blob/a65298e210870f004545a2af5d9c14bf5c1ce0f9/src/Core/Context/Context.idr%5C#L406C19-L406C22 (visited on 07/21/2024).

[14] "*Feature request: Linear types* (agda's issue tracker)." (), [Online]. Available: https://github.com/agda/agda/issues/4642 (visited on 07/21/2024).

[15] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, "Cubical Type Theory: A constructive interpretation of the univalence axiom," in *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, T. Uustalu, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 69, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 5:1–5:34, ISBN: 978-3-95977-030-9. DOI: 10.4230/LIPIcs.TYPES.2015.5. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TYPES.2015.5.

[16] A. Abel, N. A. Danielsson, and O. Eriksson, "A graded modal dependent type theory with a universe and erasure, formalized," *Proc. ACM Program. Lang.*, vol. 7, no. ICFP, Aug. 2023. DOI: 10.1145/3607862. [Online]. Available: https://doi.org/10.1145/3607862.

[17] J. Launchbury and S. L. Peyton Jones, "Lazy functional state threads," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94, Orlando, Florida, USA: Association for Computing Machinery, 1994, pp. 24–35, ISBN: 089791662X. DOI: 10.1145/178243.178246. [Online]. Available: https://doi.org/10.1145/178243.178246.

[18] S. Marlow, *Parallel and Concurrent Programming in Haskell, Techniques for Multicore and Multithreaded Programming*. O'Reilly Media, 2013.

[19] T. Petricek, D. Orchard, and A. Mycroft, "Coeffects: A calculus of context-dependent computation," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14, Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 123–135, ISBN: 9781450328739. DOI: 10.1145/2628136.2628160. [Online]. Available: https://doi.org/10.1145/2628136.2628160.

[20] B. C. Pierce, *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004, ISBN: 0262162288.

[21] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer, "Clean — a language for functional graph rewriting," in *Functional Programming Languages and Computer Architecture*, G. Kahn, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 364–384, ISBN: 978-3-540-47879-9.

[22] E. de Vries, R. Plasmeijer, and D. M. Abrahamson, "Uniqueness typing simplified," in *Implementation and Application of Functional Languages*, O. Chitil, Z. Horváth, and V. Zsók, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 201–218, ISBN: 978-3-540-85373-2.

[23] D. Orchard, V.-B. Liepelt, and H. Eades III, "Quantitative program reasoning with graded modal types," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jul. 2019. DOI: 10.1145/3341714. [Online]. Available: https://doi.org/10.1145/3341714.

[24] D. Marshall, M. Vollmer, and D. A. Orchard, "Linearity and uniqueness: An entente cordiale," in *European Symposium on Programming*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247858665.

[25] S. Smetsers, E. Barendsen, M. van Eekelen, and R. Plasmeijer, "Guaranteeing safe destructive updates through a type system with uniqueness information for graphs," in *Graph Transformations in Computer Science*, H. J. Schneider and H. Ehrig, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 358–379, ISBN: 978-3-540-48333-5.

[26] E. Barendsen and S. Smetsers, "Uniqueness typing for functional languages with graph rewriting semantics," *Mathematical Structures in Computer Science*, vol. 6, no. 6, pp. 579–612, 1996. DOI: 10.1017/S0960129500070109.

[27] E. de Vries, "Making uniqueness typing less unique," Ph.D. dissertation, Trinity College Dublin, Ireland, 2009. [Online]. Available: https://hdl.handle.net/2262/90081.

[28] J.-Y. Girard, "Linear logic," *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, 1987, ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(87)90045-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397587900454.

[29] P. Wadler, "Is there a use for linear logic?" *SIGPLAN Not.*, vol. 26, no. 9, pp. 255–273, May 1991, ISSN: 0362-1340. DOI: 10.1145/115866.115894. [Online]. Available: https://doi.org/10.1145/115866.115894.

[30] A. Spiwack, C. Kiss, J.-P. Bernardy, N. Wu, and R. A. Eisenberg, "Linearly qualified types: Generic inference for capabilities and uniqueness," *Proceedings of the ACM on Programming Languages*, vol. 6, no. ICFP, pp. 137–164, Aug. 2022, ISSN: 2475-1421. DOI: 10.1145/3547626. [Online]. Available: http://dx.doi.org/10.1145/3547626.

[31] A. Abel and J.-P. Bernardy, "A unified view of modalities in type systems," *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, Aug. 2020. DOI: 10.1145/3408972. [Online]. Available: https://doi.org/10.1145/3408972.

[32] "Formalization of graded modal type theory extended with mutable arrays." (2024), [Online]. Available: https://github.com/jul1u5/graded-type-theory/tree/9ff712147e99a11449fdbb83cd95df505ef2b28d (visited on 09/23/2024).

[33] P. Wadler, "Programming language foundations in Agda," in *Formal Methods: Foundations and Applications*, T. Massoni and M. R. Mousavi, Eds., Cham: Springer International Publishing, 2018, pp. 56–73, ISBN: 978-3-030-03044-5.

[34] T. Altenkirch and A. Kaposi, "Type theory in type theory using quotient inductive types," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16, St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 18–29, ISBN: 9781450335492. DOI: 10.1145/2837614.2837638. [Online]. Available: https://doi.org/10.1145/2837614.2837638.

[35] P. Sestoft, "Deriving a lazy abstract machine," *Journal of Functional Programming*, vol. 7, no. 3, pp. 231–264, 1997. DOI: 10.1017/S0956796897002712.

[36] P. Choudhury, H. Eades III, R. A. Eisenberg, and S. Weirich, "A graded dependent type system with a usage-aware semantics," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021. DOI: 10.1145/3434331. [Online]. Available: https://doi.org/10.1145/3434331.

[37] D. Marshall and D. Orchard, "Functional ownership through fractional uniqueness," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. DOI: 10.1145/3649848. [Online]. Available: https://doi.org/10.1145/3649848.

[38] A. Lorenzen, L. White, S. Dolan, R. A. Eisenberg, and S. Lindley, "Oxidizing OCaml with modal memory management," *Proc. ACM Program. Lang*, vol. 8, 2024. [Online]. Available: https://antonlorenzen.de/oxidizing-ocaml-modal-memory-management.pdf.

A

Reduction rules

$$\frac{H \vdash \mathsf{ren}\ E\ x \mapsto \mathsf{value}\ v\ E'}{(H, x, E, S) \rightsquigarrow_{\mathsf{u}} (H, v, E', S)}\ \textsc{U-Var} \qquad \frac{H \vdash \mathsf{ren}\ E\ x \mapsto_{|S|} \mathsf{value}\ v\ E' \mid H'}{(H, x, E, S) \rightsquigarrow_{\mathsf{p,m}} (H', v, E', S)}\ \textsc{G-Var}$$

$$\frac{}{(H, t^p u, E, S) \rightsquigarrow (H, t, E, (-^p u\ E, S))}\ \textsc{App}_1$$

$$\frac{}{(H, \lambda^0\ t, E', (-^0 u\ E, S)) \rightsquigarrow ((H, \mapsto_0 \lightning), t, \Uparrow E, \uparrow S)}\ \textsc{App}_{2e}$$

$$\frac{p \not\equiv 0}{(H, \lambda^p\ t, E', (-^p u\ E, S)) \rightsquigarrow (H, u, E, ((\lambda^p\ t)^p\ -\ E', S))}\ \textsc{App}_2$$

$$\frac{\mathsf{Value}\ u}{(H, u, E'', ((\lambda^p\ t)^p\ -\ E', S)) \rightsquigarrow ((H, \mapsto_{|S|\cdot p} \mathsf{value}\ u\ E''), t, \Uparrow E', \uparrow S)}\ \textsc{App}_3$$

$$\frac{}{(H, \mathbf{let}\ \star = t\ \mathbf{in}\ u, E, S) \rightsquigarrow (H, t, E, (\mathbf{let}\ \star = -\ \mathbf{in}\ u, S))}\ \textsc{Unit}_1$$

$$\frac{}{(H, \star, E', (\mathbf{let}\ \star = -\ \mathbf{in}\ u, S)) \rightsquigarrow (H, u, E, S)}\ \textsc{Unit}_2$$

$$\frac{\neg\mathsf{Value}\ t}{(H, \mathbf{suc}\ t, E, S) \rightsquigarrow (H, t, E, (\mathbf{suc}\ -, S))}\ \textsc{Suc}_1$$

$$\frac{\mathsf{Value}\ t}{(H, t, E, (\mathbf{suc}\ -, S)) \rightsquigarrow (H, \mathbf{suc}\ t, E, S)}\ \textsc{Suc}_2$$

$$\frac{}{(H, \mathbf{natcase}_p\ z\ s\ n, E, S) \rightsquigarrow (H, n, E, (\mathbf{natcase}_p\ z\ s\ -\ E, S))}\ \textsc{Nat-Case}_1$$

$$\frac{}{(H, \mathbf{zero}, E', (\mathbf{natcase}_p\ z\ s\ -\ E, S)) \rightsquigarrow (H, z, E, S)}\ \textsc{Nat-Case}_2$$

$$\frac{\mathsf{Value}\ t}{(H, \mathbf{suc}\ t, E', (\mathbf{natcase}_0\ z\ s\ -\ E, S)) \rightsquigarrow ((H, \mapsto_0 \lightning), s, \Uparrow E, \uparrow S)}\ \textsc{Nat-Case}_{3e}$$

$$\frac{\mathsf{Value}\ t \quad p \not\equiv 0}{(H, \mathbf{suc}\ t, E', (\mathbf{natcase}_p\ z\ s\ -\ E, S)) \rightsquigarrow ((H, \mapsto_p \mathsf{value}\ t\ E'), s, \Uparrow E, \uparrow S)}\ \textsc{Nat-Case}_3$$

$$\frac{p \not\equiv 0 \quad \neg \mathsf{Value}\ t}{(H, [^p t], E, S) \rightsquigarrow (H, t, E, ([^p -], S))}\ \textsc{Box}_1$$

$$\frac{\mathsf{Value}\ t}{(H, t, E, ([^p -], S)) \rightsquigarrow (H, [^p t], E, S)}\ \textsc{Box}_2$$

$$\frac{p \not\equiv 0}{(H, \mathbf{let}\ [^p] = t\ \mathbf{in}\ u, E, S) \rightsquigarrow (H, t, E, (\mathbf{let}\ [^p] = -\ \mathbf{in}\ u\ E, S))}\ \textsc{Box-Elim}_1$$

$$\frac{}{(H, \mathbf{let}\ [^0] = t\ \mathbf{in}\ u, E, S) \rightsquigarrow ((H, \mapsto_0 \lightning), u, \Uparrow E, \uparrow S)}\ \textsc{Box-Elim}_{1e}$$

$$\frac{\mathsf{Value}\ t \quad p \not\equiv 0}{(H, [^p t], E', (\mathbf{let}\ [^p] = -\ \mathbf{in}\ u\ E, S)) \rightsquigarrow ((H, \mapsto_{|S| \cdot p} \mathsf{value}\ t\ E'), u, \Uparrow E, \uparrow S)}\ \textsc{Box-Elim}_2$$

$$\frac{\neg \mathsf{Value}\ t \vee \neg \mathsf{Value}\ u}{(H, (t, u), E, S) \rightsquigarrow (H, t, E, ((-, u)\ E, S))}\ \textsc{Prod}_1$$

$$\frac{\mathsf{Value}\ t}{(H, t, E, ((-, u)\ E_1, S)) \rightsquigarrow (H, u, E, ((t, -)\ E_1, S))}\ \textsc{Prod}_2$$

$$\frac{\mathsf{Value}\ u}{(H, u, E_2, ((t, -)\ E_1, S)) \rightsquigarrow (H, (\mathsf{ren}\ E_1\ t, \mathsf{ren}\ E_2\ u), \mathsf{id}, S)}\ \textsc{Prod}_3$$

$$\frac{}{(H, \mathbf{let}\ (,) = t\ \mathbf{in}\ u, E, S) \rightsquigarrow (H, t, E, (\mathbf{let}\ (,) = -\ \mathbf{in}\ u\ E, S))}\ \textsc{Prod-Elim}_1$$

$$\frac{\mathsf{Value}\ t_1 \quad \mathsf{Value}\ t_2 \quad H' = H, \mapsto_{|S|} \mathsf{value}\ t_1\ E', \mapsto_{|S|} \mathsf{value}\ t_2\ (\uparrow E')}{(H, (t_1, t_2), E', (\mathbf{let}\ (,) = -\ \mathbf{in}\ u\ E, S)) \rightsquigarrow (H', u, \Uparrow\Uparrow E, \uparrow\uparrow S)}\ \textsc{Prod-Elim}_2$$

$$\frac{}{(H, \mathbf{linearly}\ k, E, S) \rightsquigarrow ((H, \mapsto_1 \mathsf{lin}), k, \Uparrow E, (\mathbf{linearly}\ 0, \uparrow S))}\ \textsc{Linearly}_1$$

$$\frac{H \vdash x \mapsto \mathsf{lin} \quad \mathsf{Value}\ k}{(H, k, E, (\mathbf{linearly}\ x, S)) \rightsquigarrow_{\mathsf{u}} (H, k, E, S)}\ \textsc{Linearly}_2$$

$$\frac{}{(H, \mathbf{consume}\ t, E, S) \rightsquigarrow (H, t, E, (\mathbf{consume}\ -, S))}\ \textsc{Consume}_1$$

$$\frac{H \vdash \mathsf{ren}\ E\ x \mapsto \mathsf{lin}}{(H, x, E, (\mathbf{consume}\ -, S)) \rightsquigarrow_{\mathsf{u}} (H, \star, E, S)}\ \textsc{U-Consume}_2$$

$$\frac{|S| \equiv 1 \quad H \vdash \mathsf{ren}\ E\ x \mapsto^1_1 \mathsf{lin} \mid H'}{(H, x, E, (\mathbf{consume}\ -, S)) \rightsquigarrow_{\mathsf{p,m}} (H', \star, E, S)}\ \textsc{G-Consume}_2$$

$$\frac{}{(H, \mathbf{duplicate}\ t, E, S) \rightsquigarrow (H, t, E, (\mathbf{duplicate}\ -, S))}\ \textsc{Duplicate}_1$$

$$\frac{H \vdash \mathsf{ren}\ E\ x \mapsto \mathsf{lin}}{(H, x, E, (\mathbf{duplicate}\ -, S)) \rightsquigarrow_{\mathsf{u}} ((H, \mapsto_{|S|} \mathsf{lin}, \mapsto_{|S|} \mathsf{lin}), (1, 0), \Uparrow\Uparrow E, \uparrow\uparrow S)}\ \textsc{U-Duplicate}_2$$

$$\frac{|S| \equiv 1 \quad H \vdash \mathsf{ren}\ E\ x \mapsto^1_1 \mathsf{lin} \mid H'}{(H, x, E, (\mathbf{duplicate}\ -, S)) \rightsquigarrow_{\mathsf{p,m}} ((H', \mapsto_1 \mathsf{lin}, \mapsto_1 \mathsf{lin}), (1, 0), \Uparrow\Uparrow E, \uparrow\uparrow S)}\ \textsc{G-Duplicate}_2$$

$$\frac{}{(H, \mathbf{new}\ t\ u, E, S) \rightsquigarrow (H, u, E, (\mathbf{new}\ t\ -\ E, S))}\ \textsc{New}_1$$

$$\frac{u \equiv \mathsf{fromNat}(n)}{(H, u, E', (\mathbf{new}\ t\ -\ E, S)) \rightsquigarrow (H, t, E, (\mathbf{new}\ -\ n, S))}\ \textsc{New}_2$$

$$\frac{H \vdash \mathsf{ren}\ E\ x \mapsto \mathsf{lin}}{(H, x, E, (\mathbf{new}\ -\ n, S)) \rightsquigarrow_{\mathsf{u}} ((H, \mapsto_{|S|} [0, ..., 0]_n), 0, \Uparrow E, \uparrow S)}\ \textsc{U-New}_3$$

$$\frac{|S| \equiv 1 \quad H \vdash \mathsf{ren}\ E\ x \mapsto^1_1 \mathsf{lin} \mid H'}{(H, x, E, (\mathbf{new}\ -\ n, S)) \rightsquigarrow_{\mathsf{p,m}} ((H, \mapsto_1 [0, ..., 0]_n), 0, \Uparrow E, \uparrow S)}\ \textsc{G-New}_3$$

$$\frac{}{(H, \mathbf{read}\ t\ u, E, S) \rightsquigarrow (H, u, E, (\mathbf{read}\ t\ -\ E, S))}\ \textsc{Read}_1$$

$$\frac{u \equiv \mathsf{fromNat}(i)}{(H, u, E', (\mathbf{read}\ t\ -\ E, S)) \rightsquigarrow (H, t, E, (\mathbf{read}\ -\ i, S))}\ \textsc{Read}_2$$

$$\frac{H \vdash \mathsf{ren}\ E\ x \mapsto xs \quad v \equiv \mathsf{fromNat}(xs[i])}{(H, x, E, (\mathbf{read}\ -\ i, S)) \rightsquigarrow_{\mathsf{u}} (H, (x, !\, v), E, S)}\ \textsc{U-Read}_3$$

$$\frac{|S| \equiv 1 \quad H \vdash \mathsf{ren}\ E\ x \mapsto^1 xs \quad v \equiv \mathsf{fromNat}(xs[i])}{(H, x, E, (\mathbf{read}\ -\ i, S)) \rightsquigarrow_{\mathsf{p,m}} (H, (x, !\, v), E, S)}\ \textsc{G-Read}_3$$

$$\frac{}{(H, \mathbf{write}\ t\ u\ v, E, S) \rightsquigarrow (H, v, E, (\mathbf{write}\ t\ u\ -\ E, S))}\ \textsc{Write}_1$$

$$\frac{t \equiv \mathsf{fromNat}(v)}{(H, t, E', (\mathbf{write}\ t\ u\ -\ E, S)) \rightsquigarrow (H, u, E, (\mathbf{write}\ t\ -\ v\ E, S))}\ \textsc{Write}_2$$

$$\frac{t \equiv \mathsf{fromNat}(u)}{(H, t, E', (\mathbf{write}\ t\ -\ v\ E, S)) \rightsquigarrow (H, t, E, (\mathbf{write}\ -\ u\ v, S))}\ \textsc{Write}_3$$

$$\frac{H \vdash \mathsf{ren}\ E\ t \mapsto xs \quad xs' = (xs[u] \coloneqq v)}{(H, t, E, (\mathbf{write}\ -\ u\ v, S)) \rightsquigarrow_{\mathsf{u}} ((H, \mapsto_{|S|} xs'), 0, E, S)}\ \textsc{U-Write}_4$$

$$\frac{|S| \equiv 1 \quad H \vdash \mathsf{ren}\ E\ t \mapsto^1_1 xs \mid H' \quad xs' = (xs[u] \coloneqq v)}{(H, t, E, (\mathbf{write}\ -\ u\ v, S)) \rightsquigarrow_{\mathsf{p}} ((H', \mapsto_1 xs'), 0, E, S)}\ \textsc{P-Write}_4$$

$$\frac{H \vdash \mathsf{ren}\ E\ t \coloneqq xs' \mid H' \quad xs' = (xs[u] \coloneqq v)}{(H, t, E, (\mathbf{write}\ -\ u\ v, S)) \rightsquigarrow_{\mathsf{m}} (H', t, E, S)}\ \text{M-WRITE}_4$$

$$\frac{}{(H, \mathbf{free}\ arr, E, S) \rightsquigarrow (H, u, E, (\mathbf{free}\ -, S))}\ \text{FREE}_1$$

$$\frac{H \vdash \mathsf{ren}\ E\ x \mapsto xs}{(H, x, E, (\mathbf{free}\ -i, S)) \rightsquigarrow_{\mathsf{u}} (H, star, E, S)}\ \text{U-FREE}_2$$

$$\frac{|S| \equiv 1 \quad H \vdash \mathsf{ren}\ E\ x \mapsto^{1}_{1} xs \mid H'}{(H, x, E, (\mathbf{free}\ -i, S)) \rightsquigarrow_{\mathsf{p,m}} (H', star, E, S)}\ \text{G-FREE}_2$$