Exploring the Uses of Quantitative Types

Andre Videla

Abstract

Idris2 is a programming language featuring Quantitative Type Theory, a Type Theory centred around tracking *usage quantities* in addition to dependent types. This is the result of more than 30 years of development spawned by the work of Girard on Linear logic. Until recently, our understanding of linear types and their uses in dependently-typed programs was hypothetical. However, this changes with Idris2, which allows us to explore the semantics of running programs using linear and dependent types. In this thesis, I explore multiple facets of programming through the lens of quantitative programming, from ergonomics to performance. I will present how quantitative annotations can help the programmer write programs that precisely match their intention, provide more information to the compiler to better analyse the program written, and enable the output bytecode to run faster.

Contents

1 Introduction

In this project, I will demonstrate different uses and results stemming from a programming practice that allows us to specify how many times each variable is being used. This information is part of the type system; in our case we are tracking if a variable is allowed to be used exactly once, or if it has no restrictions. Such a type system is called a "linear type system".

As we will see, there is a lot more to this story, so as part of this thesis, I will spend some time introducing dependent types as well as our host language: Idris. This warm-up will be followed by a context review outlining the existing body of work about linear types and related topics. Once both the theoretical and technical landscape have been set, I will introduce the concept of linear and quantitative types. Those two concepts together are the key to understanding the rest of this thesis which is about the ergonomics of Quantitative Type Theory [1][2] (QTT, for short) for software development and the performance improvement we can derive from the linearity features of Idris2.

The main goal of this thesis is to explore the uses of quantitative types in a dependently-typed setting for real-world programs, those last two chapters are the result of such exploration. With it, I aim to explain the limitations of quantitative types and propose ways of getting around them. Some of those ideas have been implemented in the Idris2 compiler and we discuss their details and consequences.

Let us begin slowly and introduce the basic concepts. The following will only make assumptions about basic understanding of imperative and functional programming.

1.1 Some Vocabulary and Jargon

Technical papers are often hard to approach for the uninitiated because of their heavy use of unfamiliar vocabulary and domain-specific jargon. While jargon is useful for referencing complicated ideas succinctly, it is a double-edged sword as it also tends to hinder learning by obscuring important concepts. Unfortunately, I do not have a solution for this problem, but I hope this section will help mitigate this feeling of helplessness when sentences seem to be composed of randomly generated sequences of letters rather than legitimate words.

You will find more complete definitions at the end, in the glossary.

Type A label associated with a collection of values. For example, `String` is the type given to strings of characters for text. `Int` is the type given to integer values. `Nat` is the type given to natural numbers.

Type-System Set of rules the types in a program have to follow to be accepted by the compiler. The goal of the type-system is to catch some classes of errors while helping the programmer reach their goal more easily.

Linear types Types that have a usage restriction. Typically, a value labelled with a linear type can only be used once, no less, no more.

Linearity / Quantity / Multiplicity Used interchangeably most of the time. They refer to the number of times a variable is expected to be used.

Syntax The structure of some piece of information, usually in the form of *text*. Syntax itself does not convey any meaning.

Semantics The meaning associated with a piece of data, most often related to syntax.

Pattern matching Destructuring a value into its constituent parts to access them or understand what kind of value we are dealing with.

Generic type / Polymorphic type / Type parameter A type that needs a concrete type to be complete. For example, `Maybe a` takes the single type `a` as parameter. Once we pass the type `Int`, it becomes the complete type `Maybe Int`.

1.2 Programming Recap

If you know about programming, you've probably heard about types and functions. Types are ways to classify values that the computer manipulates and functions are instructions that describe how those values are changed.

In *imperative programming*, functions can perform powerful operations like "malloc" and "free" for memory management or make network requests through the internet. While powerful in a practical sense, those functions are really hard to study because it is not obvious how to define them mathematically. To make our life easier we only consider functions in the *mathematical* sense of the word: A function is something that takes an input and returns an output.

```
f : A -> B
```

This notation tells us what type the function is ready to ingest as input and what type is expected as the output.

```
input    output
   ▼     ▼
f : A -> B
▲
name
```

This simplifies our model because it forbids the complexity related to operations like arbitrary memory modification or network access[1]. Functional programming describes a programming practice centred around the use of such functions. In addition, traditional functional programming languages have a strong emphasis on their type system which allows the types to describe the structure of the values very precisely.

During the rest of this thesis we are going to talk about Idris2, a purely functional programming language featuring Quantitative Type Theory (QTT), a type theory[2] based around managing resources. But before we talk about QTT itself, we have to explain what is Idris and what are dependent types.

1.3 Idris and Dependent Types

Before we jump into Idris2, allow me to introduce Idris[3], its predecessor. Idris is a programming language featuring dependent types.

A way to understand dependent types is to think of the programming language as having *first class types*, that is, types are values, just like any other in the language. Types being normal values means we can create them, pass them as arguments to functions and return them from functions. This also means the difference between "type-level" and "term-level" is blurred.

[1] We can recover those features by using patterns like "monad" but they do not fit in the scope of this brief introduction.

[2] *Type theory* is the abstract study of type systems, most often in the context of pure, mathematical logic. When we say "a Type Theory" we mean a specific set of logical rules that can be implemented into a *Type System*.

Simple Example

Let us start with a very simple example of a function in Idris without dependent types. This function returns whether or not a list[3] is empty:

```
isEmpty : List a -> Bool
isEmpty [] = True
isEmpty (x :: xs) = False
```

Every Idris function consists of two parts, the *Type signature* and the *implementation* (or body) of the function. Let us start by dissecting the Type signature:

```
--     ┌ Name of the function
--     │
--     │          ┌ Type of the argument
--     │          │
--     │          │        ┌ Return type
-- ┌───┴──┐   ┌───┴──┐   ┌─┴──┐
   isEmpty : List a -> Bool
--                ▲
--                │
--                └ Type parameter
```

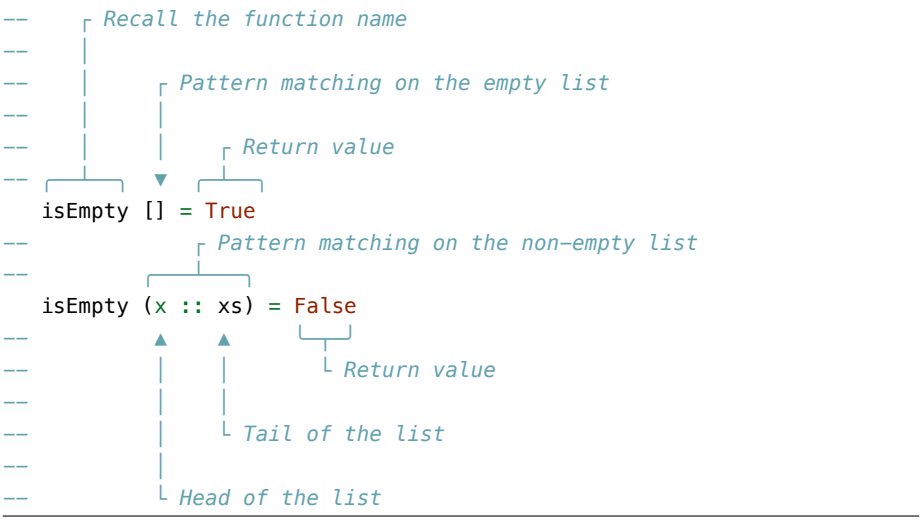
Every signature starts with a name, followed by a colon `:` and ends with a type. In this case, the type is `List a -> Bool` which represents a function that takes a list of `a` and returns a `Bool`. Interestingly, the type `a` is a *Type parameter* and could be anything; this function will work for all lists, irrespective of their content.

As for the body, here is how it is composed:

[3] A list is defined as

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

Which means a list is either empty (`Nil`) or non-empty (`Cons`) and will contain an element of type `a` and a reference to the tail of the list, which itself might be empty or not. It is customary to write the `Cons` case as `::`, and in fact the official implementation uses the symbol `::` instead of the word `Cons`. It is also customary to represent the empty list by `[]`.



Function bodies are implemented with *clauses* of the form `*function name* *arguments* = *value*`, everything before the `=` is called *the left-hand side* and everything after is called *the right-hand side*. Arguments on the left-hand side are pattern matched, and for each corresponding match, the function returns a suitable value.

In this example, we only have two cases, the empty list `[]` and the non-empty list `x :: xs`. When we match on a value we may discover that we can access new variables, and we give them names (or *bind* them). In this case, the `::` case allows us to bind[4] the head and the tail of the list to the names `x` and `xs`. Depending on whether we are dealing with an empty list or not we return `True` or `False` to indicate if the list is empty.

Pattern matching is particularly important in Idris because it allows the compiler to better understand how the types flow through our program. We are going to see an example of how dependent pattern matching[4] manifests itself when we introduce *type holes*.

Dependent Types Example

Now let us look at an example with dependent types.

[4]Binding a variable means to associate a value to a variable. A Binder is the piece of syntax that achieves the association.`let n = 3 in …` *binds* the value `3` to the name `n`.

```
intOrString : (b : Bool) -> if b then Int else String
intOrString True = 404
intOrString False = "we got a string"
```

This short snippet already shows a lot of things, but again the most important part is the type signature. Let us inspect it:

```
--              ┌ Name of the argument
--              │
--              │     ┌ Type of the argument
--              │     │
--              │     │                   ┌ Return type
--              ▼     ▼        ┌──────────┴──────────┐
intOrString : (b : Bool) -> if b then Int else String
--              │                  ▲
--              └─[dependency]─────┘
```

As you can see the return type is a *program in and of itself*! `if b then Int else String` returns `Int` if `b` is `True` and `String` otherwise. In other words, the return type is different depending on the value of `b`. This dependency is why they are called *dependent types*.

Pattern matching on the argument allows us to return different values for each branch of our program:

```
--              ┌ `b` is True so we expect to return `Int`
--              │
--              │       ┌ An Int as a return value
--              ▼     ┌─┴─┐
intOrString True = 404
intOrString False = "we got a string"
--              ▲     └─┬─────────┘
--              │       └ A String as a return value
--              │
--              └ `b` is `False` so we expect to return a String
```

This typically cannot be done in programming languages with conventional type systems. This is why one might want to use Idris rather than Java, C, or even Haskell in order to implement their programs. Some programming languages can exhibit some behaviours similar to dependent types but achieving this often comes at the expense of readability, performance, and ergonomics[5].

Holes in Idris

A very useful feature of Idris is *type holes*. One can replace any term with a variable name prefixed by a question mark, like this: `?hole` . This tells the compiler to infer the type at this position and report it to the user in order to better understand what value could possibly fit the expected type. In addition, the compiler will also report what it knows about the surrounding context to give additional insight.

If we take our example of `intOrString` and replace the implementation with a hole we have the following:

```
intOrString : (b : Bool) -> if b then Int else String
intOrString b = ?hole
```

We can then ask the compiler what is the expected type, information provided by the compiler will be shown with a column of `>` on the left side to distinguish it from code. When the compiler tells us about holes, it will display the variables available in the immediate scope of the hole above a horizontal bar, and the "goal" under the horizontal bar. Here is the type we get when asking about our `hole`:

```
> b : Bool
> --------------------------------
> hole : if b then Int else String
```

This information does not tell us what value we can use, however it informs us that the return type *depends on the value of* `b` . Therefore, pattern matching on `b` might give us more insight. It is worth noting that Idris has an interactive mode that allows the pattern matching to be done automatically by hitting a simple keystroke which will generate the code in the snippet automatically. This won't be showcased here, as it is not an Idris development tutorial, but this approach has already been put to use and proven effective in other programming languages with proof-assistant features[6][7]. Back to the example:

```
intOrString : (b : Bool) -> if b then Int else String
intOrString True = ?hole1
intOrString False = ?hole2
```

Asking again what is in `hole1` gets us:

```
> hole1 : Int
```

And `hole2` gets us:

```
> hole2 : String
```

Which we can fill with literal values like `123` or `"good afternoon"`. The complete program would look like this:

```
intOrString : (b : Bool) -> if b then Int else String
intOrString True = 123
intOrString False = "good afternoon"
```

What's wrong with non-dependent types?

Non-dependent type systems cannot represent the behaviour described by `intOrString` and have to resort to patterns like `Either`[5] to encapsulate the two possible cases our program can encounter.

```
eitherIntOrString :: Bool -> Either Int String
eitherIntOrString True = Left 404
eitherIntOrString False = Right "we got a string"
```

While this is fine in principle, it comes with a set of drawbacks that cannot be solved without dependent types. To see this, let us place a hole in the implementation of `eitherIntOrString`:

```
eitherIntOrString : Bool -> Either Int String
eitherIntOrString b = ?hole
```

```
> b : Bool
> ------------------------
> hole : Either Int String
```

While this type might be easier to read than `if b then Int else String` it does not tell us how to proceed to find a more precise type to fill. We can try pattern matching on `b`:

[5] Defined as : `data Either a b = Left a | Right b`

```
intOrString' : Bool -> Either Int String
intOrString' True = ?hole1
intOrString' False = ?hole2
```

But it does not provide any additional information about the return types to use.

```
> ------------
> hole1 : Either Int String
```

```
> ------------
> hole2 : Either Int String
```

In itself using `Either` isn't a problem; however, `Either`'s lack of information manifests itself in other ways during programming. Take the following program:

```
checkType : Int
checkType = let intValue = eitherIntOrString True
             in ?hole
```

```
> intValue : Either Int String
> ----------------
> hole : Int
```

In this program, we create the variable `intValue` by assigning it to the result of the function call to `eitherIntOrString` with the argument `True`. When we ask for the type of `hole` we see that we are expected to return an `Int`. The compiler is unable to tell us if `intValue` contains an `Int` or a `String`. Despite us *knowing* that `IntOrString` returns an `Int` when passed `True`, we cannot use this fact to convince the compiler to simplify the type for us. We have to go through a runtime check to ensure that the value we are inspecting is indeed an `Int`:

```
checkType : Int
checkType = let intValue = eitherIntOrString True in
                case intValue of
                     (Left i) => ?hole1
                     (Right str) => ?hole2
```

But doing so introduces the additional problem that we now need to provide a value for an impossible case (the `Right` case). What do we even return? We do not have an `Int` at hand to use. Our alternatives are:

- Panic and crash the program.
- Make up a default value, silencing the error but potentially introducing a bug.
- Change the return type to `Either Int String` and letting the caller deal with it.

None of which are ideal nor replicate the functionality of the dependent version. This is why dependent types are desirable; they help:

- Inform the programmer by communicating precisely which types are expected.
- The compiler better understand the semantics of the program.
- Avoid needless runtime checks.

1.4 The Story Begins

This concludes our introductory chapter. Dependent types are a thesis topic in themselves, but this introduction should be enough to give you a general idea. It remains to introduce and explain *linear types*, but we will get to it after the upcoming context review. In which I am going to present the existing literature surrounding the topic of this thesis: linear and quantitative types. Those research documents should steer us toward the goal of this thesis, exploring the uses of quantitative types for real-world programs, explaining their limitations and proposing ways of getting around them, and designing and implementing use cases for them within the Idris2 compiler.

I also want to stress that at the end is a glossary that lists all the important terms and concepts necessary to understand the body of this work. Please feel free to consult it if something is unclear.

2 Context Review

In this context review, I will comment on the existing literature about linear types and present in four parts: The first aims to tell the origin story of linear types and their youthful promises. The second will focus on the current understanding of their application for real-world use. The third will focus on the latest theoretical developments that linear types spun up. And the last will present existing research projects based on linear types. This section aims to motivate and explain why we use linear types before we introduce them formally in the next chapter.

2.1 Origins

Linear types were first introduced by J-Y. Girard in his 1987[8] publication simply named *Linear logic* . In this text he introduces the idea of restricting the application of the weakening rule and contraction rule from intuitionistic logic in order to allow to statements be managed as *resources*. Linear terms once used cannot be referred to again, premises cannot be duplicated and contexts cannot be extended. This restriction was informed by the necessity of real-world computational restrictions, in particular accessing information concurrently.

One of the pain-points mentioned was the inability to restrict usages to something more sophisticated than "used exactly once". Linear variables could be promoted to their unrestricted variants with the exponential operator (`!`) but it removes any benefit we get from linearity - a limitation that will be revisited in the follow-up paper: Bounded linear logic.

Already at this stage, the memory implication of linear types was considered. Typically, the exponential operator was understood as being similar to "long-term storage" of a variable such that it could be reused in the future, whereas linear variables were stored in short-term memory to be reclaimed, like a register or a stack.

Bounded linear logic[9] improves the expressivity of linear logic while keeping its benefits: intuitionistic-compatible logic that is computationally relevant. The key difference with linear logic is that weakening rules are *bounded* by a finite value such that each value can be used as many times as the bound allows. In addition, some typing rules might allow it to *waste* resources by *underusing* the variable, hinting that affine types might bring some concrete benefits to our programming model.

As before, there is no practical application of this in terms of programming language. However, this brings up the first step toward managing *quantities* in the language. An idea that will be explored again later with Granule[10][11][12][13] and Quantitative Type Theory[1][2].

2.2 Applications

Soon after the development of linear types, they appeared in a paper aimed at optimising away redundant allocations when manipulating lists: The deforestation algorithm.

Deforestation[14] is an algorithm designed to avoid extraneous allocation when performing list operations in a programming language close to System-F. The assumption that operations on lists must be linear was made to avoid duplicating operations. If a program was non-linear, the optimisation would duplicate each of the computation associated with the non-linear variable, making the resulting program less efficient.

While deforestation itself might not be the algorithm that we want to implement today, it is likely we can come up with a similar set of optimisation rules in Idris2 that relies on linearity.

Using linearity to avoid duplicating computations was again investigated in *Once upon a Type*[15] which formalises the detection and inference of linear variables. And uses this information for safe inlining. Indeed, arbitrarily inlining functions might result in duplicated computation:

```
let x = f y in
    (x, x)
```

In this example, inlining into `(f x, f x)` duplicates the work done by `f`.

Besides inlining, another way to use linear types for performance is memory space mutation. *Linear types can change the world*[16] showed that Linear types can be used for in-place update and mutation instead of relying on copying. They provide a safe API to update values by relying on the linearity of the argument being used.

However, the weakness of this result is that the API exposed to the programmer relies on nested continuations when traversing nested data types, which is largely seen as an undesirable user experience. *Is there a use for linear logic*[17] makes similar claims and introduces a concept of types with a *unique* pointer to them.

This removes the need for the clumsy continuation style, but comes at the cost of more complex typing rules. This restriction for unique pointers is something we are going to encounter again during this thesis.

The memory management benefits from linear types do not stop here, in *Reference counting as a computational interpretation of linear logic* [18], linear types are shown to correspond to reference counting in the runtime. This paper shows that a simple calculus augmented with memory management primitives can make use of linearity in order to control memory allocation and deallocation using linear types. While the calculus itself isn't very ergonomic to use for programming, one could imagine using it as a backend for a reference-counted runtime.

Linear types need not be used exclusively for compiler optimisations but can also greatly improve programming ergonomics and safety as we see in *Practical affine types*[19]; which aims to answer "What does it mean to have access to linear and affine types *in practice*"? Indeed, most of the results we've talked about develop a theory for linear types using a variant of linear logic and then present a toy language to showcase their contribution. However, this does not teach us how they would interact and manifest in existing programs or existing software engineering workflows. Do we see new programming patterns emerging? Is the user experience improved or diminished? In what regard is the code different to read or write? How is the code organised and shared? All those questions can only be answered by a fully-fledged implementation of a programming language equipped to interact with existing systems.

Practical affine types demonstrated that their implementation for linear+affine types allows us to express common operations in concurrent programs without any risk of data races[20]. They note that typical stateful protocols should also be implementable since their language is a strict superset of others which already provided protocol implementations. Those two results hint that linear types in a commercially relevant programming language would provide us with additional guarantees without impeding on the existing writing or reading experience of programs.

Haskell already benefits from a plethora of big and small extensions that are so powerful that combining them allows to obtain some form of dependent types[5]. Linear Haskell[21] is notable in that it extends the type system to allow linear functions to be defined. It introduces the linear arrow `-o` which declares a function to be linear. Because of Haskell's laziness, linearity doesn't mean "will

be used exactly once" but rather "*if it is used*, then it will be used exactly once".

This addition to the language was motivated by a concern for safe APIs, typically when dealing with unsafe or low-level code. Linear types allow restricting an API so that it cannot be misused while keeping the same level of expressivity and being backwards compatible. This backward compatibility is in part allowed thanks to parametric linearity, the ability to abstract over linearity annotations, and in part due to the approach of making *function arrows* linear instead of *types* themselves.

Linearity on the type has the benefit that values themselves can be marked linear, which would enforce their uniqueness. Often, this comes at the cost of splitting our context in two[22] between linear values and non-linear values which makes it unsuitable for a dependent type theory, in which both types and terms live in the same context[6].

2.3 Cutting Edge Linear Types

Granule is a language that features *quantitative reasoning via graded modal types* and features indexed types[7]. This effort is the result of years of research in the domain of effects, co-effects, resource-aware calculus, and co-monadic computation[23]. Granule itself makes heavy use of *graded monads*[11], which allow to precisely annotate co-effects[24] in the type system. This enables the program to model *resource management* at the type-level. What's more, graded monads provide an algebraic structure to *combine and compose* those co-effects[25]; this way, linearity can not only be modelled but also *mixed-in* with other interpretations of resources. While this multiplies the opportunities in resource tracking, this approach hasn't received the treatment it deserves regarding performance and tracking runtime complexity.

Until now, we have not addressed the main requirement of Idris2: We intend to use *both* dependent types *and* linear types within the same language. However, no satisfying theory came until *I got plenty o nuttin*[2] and its descendant, *Quantitative type theory* [1]. A previous attempt[26] noted that a single language could have both a linear arrow and a dependent arrow and while the linear arrow

[6] I personally find the unified context to be a very elegant solution.

[7] An index is a type parameter that changes with the values that inhabit the type. For example `["a", "b", "c"] : Vect 3 String` has index `3` and a type parameter `String`, because it has `3` elements and the elements are Strings. If the value was `[`"a", "b"]` then the type would become `Vect 2 String`, that is, the index changes from `3` to `2`, but the type parameter stays as `String`.

can depend on non-linear terms, the linear arrow cannot be dependent. This is achieved again by splitting our context into a linear one and non-linear one.

To implement full dependent types, QTT makes the following changes:

- Dependent typing uses *erased* variables
- Multiplicities are tracked on the *binder* rather than being a feature of the type of a variable.

While this elegantly merges the capabilities of a Martin-Löf-style type theory[27] and Linear Logic[8], the proposed result does not touch upon the potential performance improvement that such a language could achieve by removing indices from the runtime[28]. However, it has the potential to bring together linear types and dependent types in a way that allows precise resource tracking and strong performance guarantees. Resource tracking through semirings is not a novel idea but has been explored through the lens of explicit comonadic computation [29] instead of dependent linear type systems.

Unfortunately, this approach does not solve every long-standing problem of computer science like how to use the univalence axiom in a language like cubical Agda[30]. Interestingly enough, cubical type theory and homotopy type theory display a problem that could be solved in Idris2 very easily: they are non-computational. This means that they could be restricted to *erased* multiplicity such that they are enforced by the compiler to never appear at runtime.

Since this approach has already been tried in Agda[31], maybe Idris2 should be extended with Observational Type Theory [32] to solve the problem of functional extensionality. This would greatly improve the range of programs Idris2 can write as we see with Idris-CT[33], which require postulating function extensionality for a lot of their proofs. Such an extension to the language would beautifully mix the dependent type aspect of Idris with linear types.

As we've seen earlier, linearity has strong ties with resource and memory management, including reference counting. *Counting immutable beans*[34] does not concern itself with linearity, but it references several heuristics which are eerily similar to the rules enforced by linear logic. Those heuristics show up in the context of using reference counting for a purely functional programming language. While reference counting has, for a long time, been disregarded in favour of runtime garbage collectors, it has now proven to be commercially viable in languages like Swift or Python. The specific results presented here are focused on the performance benefits in avoiding unnecessary copies and reducing the amount of increment and decrement operations when manipulating the refer-

ence count at runtime. We are going to revisit this statement in section 5.3 and frame it in the context of quantitative types.

2.4 Innovative Uses of Linear Types

In this final section of the context review I will present some interesting uses for linear types in some detail.

Opening the Door to New Opportunities

Protocol descriptions and dependent types work marvellously well. State machines can be represented by data types and their index can ensure we compose them in ways that make sense.

As a reminder of how state machines can be encoded in Idris, here is a simple door protocol in Idris using *indexed monads* [35] where the indices behave like a Hoare Triple. An approach that's been put to practice in *State machines all the way down*[36] and gives rise to our example:

```
data DoorState = Open | Closed

data Door : Type -> DoorState -> DoorState -> Type where
    Open : Door () Closed Open
    Close : Door () Open Closed
    Play : Door () Open Open
    Pure : ty -> Door ty state state
    (>>=) : Door a state1 state2 ->
             (a -> Door b state2 state3) ->
             Door b state1 state3
```

This encodes a state machine with two states, `Open` and `Closed`. The protocol of this state machine has three operations `Open` ,`Close` and `Play`. It models a game room in which one can enter by opening the door, play some game in the room and leave the room by closing the door behind them.

Assuming we want to enforce operations on the door that start and end in the `Closed` state we can represent the computation with this signature:

```
doorProtocol : Door () Closed Closed
```

One issue with this approach is that it constrains us to a free-monadic style implementation where we need to write an interpreter for our monadic program,

and we cannot mix other protocols within it without changing the `Door` data type.

The following example uses Idris2, with linear types, to implement the same protocol. While Linear types in Idris2 have not been formally introduced (it will be the topic of the next chapter) it is enough to show that we can use them to replace our previous implementation with one that is simpler and more intuitive:

```
data DoorState = Open | Closed

data Door : (state : DoorState) -> Type where
  MkDoor : Door Closed
  Open : (1 _ : Door Closed) -> Door Open
  Close : (1 _ : Door Open) -> Door Closed
  Play : (1 _ : Door Open) -> Door Open

-- Take an operation on doors and execute it
operateDoor : (op : (1 _ : Door Closed) -> Door Closed)
           -> Door Closed
operateDoor op = op MkDoor
```

In this example, instead of monadic composition, we use plain *function composition* to ensure that our protocol obeys the rules of our protocol. This result is desirable if only because monads do not compose with each other, which prevents us from interleaving another protocol in the *indexed monad* example, but functions do compose.

`operateDoor` verifies the protocol because it forces the `op` function to make use of the provided door. The client of the API has no choice but to manipulate the existing door until the protocol is over.

```
-- Doesn't typecheck because door is ignored.
operateDoor (\door => MkDoor)
```

We can now define the program simply by combining our door operations:

```
infixr 1 &>

-- Sequential linear function composition.
(&>) : ((1 _ : a) -> b) ->
       ((1 _ : b) -> c) ->
        (1 _ : a) -> c
(&>) f g x = g (f x)

operateDoor : (1 _ : Door Closed) -> Door Closed
operateDoor = Play &> Close &> Open &> Play &> Play &> Close
```

This also makes combining other protocols trivial, assume we have another protocol to connect to the internet and is characterised by the signature `connect : (1 _ : Socket Open) -> Socket Closed` we can take the product of the two functions and interleave their arguments:

```
operateProd : (1 _ : (Socket Open, Door Open))
           -> (Socket Closed, Door Closed)
operateProd (socket, door) =
  let openDoor = Open door
      socket' = send "message" socket
      door' = Enjoy openDoor in
      (close socket', Close door')
```

Which is much more natural than having to rewrite our door protocol and rewrite our interpreter for the operations of the protocol.

More Granular Dependencies

Granule is a programming language with *graded modal types*, types which rely on *graded modalities*, which are annotations that span a *range* of different values to describe each type. Those values can themselves be paired up together and combined to represent even more complex behaviour than our linear multiplication. For now, let us stick to multiplication and see what a future version of Idris supporting graded modalities could look like.

Granule's syntax is very close to Idris, Agda, and Haskell, however, linearity in *Granule* is the default so there is nothing to specify for a linear variable. In addition, `Nat` is not a `Type` in *Granule* but a *modality* modelling exact usage. This means that, to work with `Nat` and write dependencies between them, we will create a data type indexed on `Nat`:

```
data INat (n : Nat) where
  Z : INat 0;
  S : INat n -> INat (n + 1)
```

This allows us to write the add function as follows:

```
linearAdd : forall {n m : Nat} . INat n -> INat m -> INat (n + m)
linearAdd Z m = m;
linearAdd (S n) m = S (linearAdd n m)
```

If we were to omit the `m` in the first branch and write `linearAdd Z m = Z` we would get the error:

```
> Linearity error: multiplication.gr:
> Linear variable `m` is never used.
```

Which is what we expect.

Now that we have indexed `Nat` we can implement a `multiplication` function and properly carry around the usage characteristic of our program:

```
multiplication : forall {n m : Nat} . INat n -> (INat m) [n] -> INat (n * m)
multiplication Z [m] = Z;
multiplication (S n) [m] = linearAdd m (multiplication n m)
```

As you can see, we annotate the second argument of the type signature with `[n]` which indicates that the modality of the second argument depends on the value of the first argument. This syntax repeats in the implementation where the second argument `m` has to be "unboxed" using the `[m]` syntax which will tell the compiler to correctly infer the usage allowed by the indexed modality. In the first branch, there are `0` uses available, and in the second there are `n + 1` uses available.

While *Granule* doesn't have dependent types, indexed types are enough to implement interesting programs such as multiplication. More recent developments have made progress toward implementing full type dependency between quantities and terms in the language[37].

Invertible Functions

Yet another use of linearity is to use them to implement *invertible functions*. That is, functions that have a counterpart that can undo their actions. Such

functions are extremely common in practice but aren't usually described in terms of their ability to be undone. Here are a couple of examples:

- Addition and subtraction
- `::` and `tail`
- serialisation/deserialisation

The paper about Sparcl[38] goes into details about how to implement a language that features invertible functions, they introduce a new (postfix) type constructor `• : Type -> Type` that indicate the type in argument is invertible. Invertible functions are declared as linear functions `A• -o B•`. Invertible functions can be called to make progress one way or the other given some data using the `fwd` and `bwd` primitives:

```
fwd : (A• -o B•) -> A -> B
bwd : (A• -o B•) -> B -> A
```

Invertible functions aren't necessarily total, for example `bwd (+ 1) Z` will result in a runtime error. This is because of the nature of invertible functions: the `+ 1` function effectively adds an `S` layer to the given data. In order to undo this operation, we need to *peel off* an `S` from the data. But `Z` doesn't have an `S` constructor surrounding it, resulting in an error.

This type of runtime error can be avoided in Idris by adding a new implicit predicate that ensures the data has the correct shape:

```
bwd : (f : (1 _ : A•) -> B•) -> (v : B) -> {prf : v = fwd f x)} -> A
```

This ensures that we only take values of `B` that come from a `fwd` operation, that is, it only accepts data that has been correctly built instead of arbitrary data. If we were to translate this into our nat example it would look like this:

```
undo+1 : (n : Nat) -> {prf : n = S k} -> Nat
```

Which ensures that the argument is an `S` of `k` for any `k`.

3 Idris2 and Linear Types

We've seen in the introduction how Idris features dependent types and some basic demonstrations about how to use them. Idris2 takes things further and introduces *linear types* through Quantitative Type Theory, allowing us to define how many times a variable will be used. Three different quantities exist in Idris2 : `0`, `1` and `ω`. `0` means the value cannot be used in the body of a function, `1` means it has to be used exactly once, no less, no more. `ω` means the variable isn't subject to any usage restrictions, just like other (non-linear) programming languages. In Idris2, `ω` is implicit, when no quantity is specified and left blank, it is assumed to be unrestricted. This is why we have not seen any usage quantities during the introduction.

We are going to revisit usage quantities later, as there are more subtleties, especially with the `0` usage. For now, we are going to explore some examples of linear functions and linear types, starting with very simple functions such as incrementing numbers.

3.1 Incremental Steps

When designing examples, natural numbers are a straightforward data type to turn to. Their definition is extremely simple: `data Nat = Z | S Nat` which means that a `Nat` is either zero `Z` or the successor of another natural number, `S`.

Given this definition let us look at the function that increments a natural number:

```
increment : Nat -> Nat
increment n = S n
```

As we've seen before with our `intOrString` function, we can name our arguments in order to refer to them later in the type signature. We can do the same here even if we do not use the argument in a dependent type. Here we are going to name our first argument `n`.

```
increment : (n : Nat) -> Nat
increment n = S n
```

In this case, the name `n` doesn't serve any other purpose than documentation, but our implementation of linear types has one particularity: quantities have

to be assigned to a *name*[8]. Since the argument of `increment` is used exactly once in the body of the function we can update our type signature to assign the quantity `1` to the argument `n`:

```
--           ┌ We declare `n` to be linear
--           ▼
increment : (1 n : Nat) -> Nat
increment n = S n
--              ▲
--              │
--              └ We use n once here
```

The compiler will now check that `n` is used exactly once.

Additionally, the rules of linearity also apply to each pattern variable that was bound. That is, if the value we are matching is linear, then we need to use the pattern variables linearly.

```
sum : (1 n : Nat) -> (1 m : Nat) -> Nat
sum Z m = m
--  ▲
--  └ We match on the argument here
sum (S n) m = S (sum n m)
--  └─┘              ▲
--    │              └ We use `n`
--    │
--    └ We match on the argument and bind `n`
```

In this last example, we match on `S` and bind the argument of `S` (the predecessor) to the variable `n`. Since the original argument was linear, `n` is linear too and is indeed used later with `sum n m`.

3.2 Dropping and Picking It Up Again

This programming discipline does not allow us to express every program the same way as before. Here are two typical examples that cannot be expressed :

```
drop : (1 v : a) -> ()

copy : (1 v : a) -> (a, a)
```

[8]This is because QTT assigns quantities to *binders* rather than types.

Here, the first function aims to ignore the argument and the second function duplicates its argument and packages it in a pair.

We can explore what is wrong with those functions by trying to implement them and making use of holes.

```
drop : (1 v : a) -> ()
drop v = ?drop_rhs
```

```
> 0 a : Type
> 1 v : a
> ------------
> drop_rhs : ()
```

As you can see, each variable is annotated with an additional number on its left, `0` or `1`, that informs us how many times each variable has to be used (If there is no restriction, the usage number is simply left blank).

Here we need to use `v` (since it is marked with `1`) but we are only allowed to return `()`. This would be solved if we had a function of type `(1 v : a) -> ()` to consume the value and return `()`, but this is exactly the signature of the function we are trying to implement!

If we try to implement the function by returning `()` directly we get the following:

```
drop : (1 v : a) -> ()
drop v = ()
```

```
> There are 0 uses of linear variable v
```

This indicates that `v` is supposed to be used, but no uses have been found.

Similarly, for `copy` we have:

```
copy : (1 v : a) -> (a, a)
copy v = ?hole
```

```
> 0 a : Type
> 1 v : a
> ------------
> hole : (a, a)
```

In which we need to use `v` twice, but we're only allowed to use it once. Using it twice results in this program, with this error:

```
copy : (1 v : a) -> (a, a)
copy v = (v, v)
```

```
> There are 2 uses of linear variable v
```

Interestingly enough, partially implementing our program with a hole gives us an amazing insight:

```
copy : (1 v : a) -> (a, a)
copy v = (v, ?hole)
```

```
> 0 a : Type
> 0 v : a
> ----------
> hole : a
```

The hole has been updated to reflect the fact that though `v` is in scope, no uses of it are available. Despite that, we still need to make up a value of type `a` out of thin air, which is impossible because we do not know how to construct a value of type `a`.

While there are experimental ideas that allow us to recover those capabilities, they are not currently present in Idris2. We will talk about those limitations and how to overcome them in section 5.

3.3 Dancing Around Linear Types

Linear values cannot be duplicated or ignored, but provided we know how the values are constructed, we can work hard enough to tear them apart and build them back up. This allows us to dance around the issue of ignoring and duplicating linear variables by exploiting pattern matching to implement functions such as `copy : (1 v : a) -> (a, a)` and `drop : (1 v : a) -> ()`.

The next snippet shows how to implement those for `Nat`:

```
dropNat : (1 v : Nat) -> ()
dropNat Z = ()
dropNat (S n) = dropNat n

copyNat : (1 v : Nat) -> (Nat, Nat)
copyNat Z = (Z, Z)
copyNat (S n) = let (a, b) = copyNat n in
                    (S a, S b)
```

Notice that `dropNat` effectively spends `O(n)` doing nothing, while `copyNat` *simulates* allocation by constructing a new value that is identical and takes the same space as the original one (albeit very inefficiently).

We can encapsulate those functions in the following interfaces:

```
interface Drop a where
    drop : (1 v : a) -> ()

interface Copy a where
    copy : (1 v : a) -> (a, a)
```

Since we know how to implement those for `Nat` we can ask ourselves how we could give an implementation of those interfaces to additional types.

```
module Main

import Data.List

data Tree a = Leaf a | Branch a (Tree a) (Tree a)

interface Drop a where
    drop : (1 v : a) -> ()

interface Copy a where
    copy : (1 v : a) -> (a, a)

Drop a => Drop (List a) where
    copy ls = ?drop_list_impl

Copy a => Copy (List a) where
    Copy ls = ?copy_list_impl

Drop a => Drop (Tree a) where
    copy tree = ?drop_Tree_impl

Copy a => Copy (Tree a) where
    Copy tree = ?copy_tree_impl
```

Going through this exercise would show that this process of tearing apart values and building them back up would be very similar to what we've done for `Nat`, but there is an additional caveat to this approach: primitive types do not have constructors like `Nat` and `Tree` do. `String`, `Int`, `Char`, etc, are all assumed to exist in the compiler and are not declared using the typical `data` syntax. This poses a problem for our implementation of `Copy` and `Drop` since we do not have access to their constructors nor structure. A solution to this problem will be provided in section 5.1 .

3.4 Linear Intuition

One key tool in understanding this thesis is to develop an intuition for linear types. Linear types are rare in other programming languages which make them seem foreign at first glance. In the absence of familiarity, it is especially important to build up intuition.

To that end, we are going to explore several examples that illustrate how linearity can be understood.

You Can't Have Your Cake and Eat It Too

Imagine this procedure:

```
eat : (1 cake : Cake) -> Full
eat (MkCake ingredients) = digest ingredients
```

This allows you to eat your cake. Now take this other one:

```
keep : (1 cake : Cake) -> Cake
keep cake = cake
```

This allows you to keep your cake to yourself. Given those two definitions, you can't both *have* your cake *and eat it too*:

```
notPossible : (1 cake : Cake) -> (Full, Cake)
notPossible cake = (eat cake, keep cake)
```

This fails with the error:

```
> Error: While processing right hand side of notPossible.
>   There are 2 uses of linear name cake.
>
>     |
>     | notPossible cake = (eat cake, keep cake)
>     |                    ^^^^^^^^^^^^^^^^^^^^
>
> Suggestion: linearly bounded variables must be used exactly once.
```

A linear variable must be used exactly once, therefore, you must choose between having it, or eating it, but not both.

Note that removing every linearity annotation makes the program compile:

```
eat : (cake : Cake) -> Full
eat (MkCake ingredients) = digest ingredients

keep : (cake : Cake) -> Cake
keep cake = cake

nowPossible : (cake : Cake) -> (Full, Cake)
nowPossible cake = (eat cake, keep cake)
```

Since we have not said anything about `Cake`, there is no restriction in usage, and we can have as much cake as we want (and keep it too!).

Drawing Unexpected Parallels

Take the following picture:

This is a simple *connect the dots* game. If you have this text printed out and have a pencil handy, I encourage you to try it out and discover the picture that hides behind those dots.

The point of this exercise is to show what happens to a linear variable once it's consumed: It cannot be reused anymore. The dot pattern is still visible, the numbers can still instruct you how to connect them. But since the drawing has already been carried out, there is no possible way to repeat the experience of connecting the dots.

Linear variables are the same, once they are used, they are "spent". This is shown explicitly in Idris2's type system by using holes:

```
let 1 dots = MkDots
    1 drawing = connect dots in
    ?rest
```

Inspecting the hole we get:

```
>  0 dots : Graph
>  1 drawing : Graph
> ------------------------------
> rest : Fun
```

It indicates that, while we can still *see* the dots, we cannot do anything with them, they have linearity `0`. However, we ended up with a `drawing` that we can now use![9]

Safe Inlining with `1`

Linear variables have to be used exactly once, no less, no more. An extremely nice property this gives us can be summarised with the following statement:

> A linear variable can always be safely inlined

Inlining refers to the ability of a compiler to replace a function call by the body of the function. This is a typical optimisation technique aimed at enabling further optimisations on the resulting program.

Safely inlined means that the inlining process will not result in a bigger and less efficient program. Take the following example:

[9] You will notice that the program asks us to return a value of type `Fun`, this is because the goal of this exercise is to have fun.

```
let x = f y in
    (x, x)
```

After inlining `x`, that is, replace every occurrence of `x` by its definition, we obtain:

```
(f y, f y)
```

Which is less efficient than the original program. Indeed, imagine that `f` is a function that takes 5 days to run. The first case calls `f` once and duplicates its result, which would take 5 days. But the second case calls `f` twice, which would take 10 days in total.

If `x` were to be *linear* this problem would be caught:

```
let 1 x = f y in
    (x, x)
```

```
> There are 2 uses of linear variable x
```

Conversely, if a program typechecks while using a linear variable, then all linear variables can be inlined without loss of performance. What's more, inlining can provide further opportunities for optimisations down the line. In the following example, while `y` cannot be inlined, `x` can be.

```
let 1 x = 1 + 3 in
    y = x + 10 in
    (y, y)
```

The result of inlining `x` would be as follows[10]:

```
let y = 1 + 3 + 10 in
    (y, y)
```

Erased Runtime for `0`

In Idris2, variables can also be annotated with linearity `0`, this means that the value is *inaccessible* and cannot be used. But if that were truly the case, what would be the use of such a variable?

[10]This example does not quite work in Idris2 as it stands, Idris2 is very conservative about propagating linearity to variables that are bound to linear function calls.

Those variables are particularly useful in a dependently-typed programming language because, while they cannot be used in the body of our program, they can be used in type signatures. Take this example with vector:

```
length : Vect n a -> Nat
length [] = Z
length (_ :: xs) = S (length xs)
```

The length of the vector is computed by pattern matching on the vector and recursively counting the length of the tail of the vector and adding `+1` to it (recall the `S` constructor for `Nat`). If the vector is empty, the length returned is zero (`Z`).

Another way to implement the same function in Idris 1 (without linear types) was to do the following:

```
-- This works in Idris1
length : Vect n a -> Nat
length _ {n} = n
```

The `{n}` syntax would bring the value from the *type level* to the *term level*, effectively making the type of the vector a value that can be used within the program. However, doing the same in Idris2 is forbidden:

```
> Error: While processing right hand side of length.
>   n is not accessible in this context.
>
>     |
>     | length _ {n} = n
>     |                ^
```

It is hard to understand why this is the case just by looking at the type signature `Vect n a -> Nat` and this is because it is not complete. Behind the scenes, the Idris2 compiler is adding implicit arguments [11] for `n` and `a` and automatically gives them linearity `0`. The full signature looks like this:

[11] Implicit arguments are arguments to functions that are not given by the programmer, but rather are filled in by the compiler automatically. Implicit arguments are extremely important in dependently-typed languages because without them every type signature would be extremely heavy. Moreover, since the distinction between types and terms is blurry, the mechanism to infer *types* is the same as the mechanism to infer *terms* which is how the compiler can infer which value to insert whenever a function requires an implicit argument.

```
length : {0 n : Nat} -> {0 a : Type} -> Vect n a -> Nat
length _ {n} = n
```

The `0` means we cannot use the variable outside of the type signature, but if we can use it in the signature, why can't we use it in the implementation of the `length` function?

This is explained by the fact that linearity `0` variables are available *at compile time* and are forbidden to appear *at runtime*. The compiler can use them, compute types with them, but they cannot be allocated and used during the execution of the program.

This is why linearity `0` variables are also called `erased` variables: because they are removed from the execution of the program. We can use them to convince the compiler that some invariants hold, but we cannot allocate any memory for them at runtime.

Finally, another subtlety is that erased variables can appear inside the body of a function, but only in a position with `0` usage. Such functions are:

1. Arguments annotated with `0`

```
toNat : (0 n : Nat) -> INat n -> Nat
toNat Z Zero = Z
toNat (S n) (Succ m) = S (toNat n m)
--       ▲                      ▲
--       │                      └ Used even if erased
--       │
--       └ Bound with linearity 0
```

Here the recursive call uses `n` which has linearity `0`, but this is allowed because the first argument of `toNat` takes an argument of linearity `0`. In other words, `n` cannot be consumed, but `toNat` does not consume its first argument anyway, so all is good.

2. Rewrites

```
sym : (0 prf : x = y) -> y = x
sym prf = rewrite prf in Refl
```

Rewriting a type does not consume the proof.

```
--          ┌ `n` is erased
--          ▼
reverse' : {0 n : Nat} -> (1 vs : Vect n Nat) -> Vect n Nat
reverse' vs = let v2 : Vect n Nat = reverse vs in v2
--                          ▲
--                          └ `n` appears here
```

Even if `n` appears in the body of the function, appearing in a type signature does not count as a use.

No Branching with `0`

In general, we cannot match on erased variables, there is however an exception to this rule. Whenever matching on a variable *does not* result in additional branching, then we are allowed to match on this variable, even if it erased. Such matches are called *uninformative*, and they are characterised by the fact that they do not generate new codepaths.

"No new codepaths" means that, whether we match or not, the output bytecode would be the same. The difference lies in that matching on those variables would inform us, and the compiler, of very important properties from our types. Just like the `intOrString` example informed us of the return type of our function, an uninformative match can reveal useful information relevant to typechecking.

A good example of an uninformative match is `Refl` which has only one constructor:

```
data (=) : (a, b : Type) -> Type where
  Refl : (a : Type) -> a = a
```

This suggests that, even if our equality proof has linearity `0`, we can match on it; since there is only 1 constructor we are never going to generate new branches.

But an uninformative match can also happen on types with multiple constructors. Take this indexed `Nat` type:

```
data INat : Nat -> Type where
  IZ : INat Z
  IS : INat n -> INat (S n)
```

This is simply a duplicate for `Nat` but carries its own value as index. Now let us write a function to recover the original `Nat` from an `INat`:

```
toNat : (0 n : Nat) -> (1 m : INat n) -> Nat
toNat Z IZ = Z
toNat (S n) (IS m) = S (toNat n m)
```

Even if we annotated `n` with linearity `0`, we are allowed to match on it. To understand why, let us add some holes and remove the matching:

```
toNat : (0 n : Nat) -> (1 m : INat n) -> Nat
toNat n IZ = ?branch
toNat (S n) (IS m) = S (toNat n m)
```

Idris2 will not allow this program to compile and will fail with the following error:

```
> Error: While processing left hand side of toNat.
>   When unifying INat 0 and INat ?n.
> Pattern variable n unifies with: 0.
>
>     |
>     |   IZ : INat Z
>     |              ^
>     |   IS : INat n -> INat (S n)
>     |  toNat n IZ = ?branch
>     |        ^
>
> Suggestion: Use the same name for both pattern variables, since they
>   unify.
```

It tells us that `n` unifies with `Z` and forces the user to spell out the match, effectively forcing uninformative matches to be made. A similar error appears if we try the same thing on the second branch, trying to remove `S n`.

4 Quantitative Type Theory in Practice

QTT is still a recent development and because of its young age, it has not seen widespread use in commercial applications. The Idris2 compiler itself stands as the most popular example of a complex program that showcases uses for QTT and quantitative types. Linear types already benefit from a body of work that showcase their uses[39][16][40][41][42][20][17][15][14], but one of the goals of this thesis was to list and discover some new and innovative uses for linear types and QTT. In this chapter, I will mention some specific uses for QTT that I discovered during my study.

4.1 Linear Multiplication

We've seen how we can write addition of natural numbers using linear types. But can we write a multiplication algorithm using linear types? Let us inspect the traditional multiplication algorithm and see if we can update it with linear types.

Here is a multiplication function without any linear variables:

```
multiplication : (n : Nat) -> (m : Nat) -> Nat
multiplication Z m = Z
multiplication (S n) m = m + (multiplication n m)
```

Just like with addition, we notice that some variables are only used once, but some aren't: `n` is used exactly once in both cases, but `m` is not used in the `Z` one, and used twice in the `S` case. Which leads to the following program:

```
-- `m` is unrestricted, `n` is linear
multiplication : (1 n : Nat) -> (m : Nat) -> Nat
multiplication Z m = Z
multiplication (S n) m = m + (multiplication n m)
```

Which compiles correctly, but how could we go about implementing a completely linear version of `multiplication`?

Indeed, writing `multiplication : (1 n : Nat) -> (0 m : Nat) -> Nat` gets us the error:

```
Error: While processing right hand side of multiplication.
    m is not accessible in this context.

    |
    | multiplication (S n) m = m + (multiplication n m)
    |                          ^
```

Which catches the fact that `m` is used twice in the second branch (but the first branch is fine).

Ideally, we would like to write this program:

```
--        The multiplicity depends on the first arugment
--                                ▼
multiplication : (1 n : Nat) -> (n m : Nat) -> Nat
multiplication Z m = Z
multiplication (S n) m = m + (multiplication n m)
```

However, Idris2 and QTT do not support *dependent linearities* or *first class linearity* where linearity annotations are values within the language.

We can, however, attempt to replicate this behaviour with different proxies:

```
provide : Copy t => Drop t => (1 n : Nat) -> (1 v : t)
       -> (DPair Nat (\x => n = x), Vect n t)
provide 0 v = let () = drop v in (MkDPair Z Refl, [])
provide (S k) v = let (v1, v2) = copy v
                      (MkDPair n prf, vs) = provide k v1
                  in MkDPair (S n) (cong S prf), v2 :: vs)

multiplication : (1 n, m : Nat) -> Nat
multiplication n m = let (MkDPair n' Refl, ms) = provide n m in mult n' ms
  where
    mult : (1 n : Nat) -> (1 vs : Vect n Nat) -> Nat
    mult 0 [] = 0
    mult (S k) (m :: x) = m + (mult k x)
```

This program attempts to simulate the previous signature by creating a dependency between `n` and a vector of length `n` containing copies of the variable `m`[12] with the type `mult : (1 n : Nat) -> (1 _ : Vect n Nat) -> Nat` .

[12]For the purposes of this example there is no proof that the vector *actually* contains only copies of `m` but this is an invariant that could be implemented at the type level. But doing so would introduce lots of equality proofs which would render the code even harder to read.

As we've demonstrated, we technically can express a more complex relationship between linear types provided they implement our interfaces `Drop` and `Copy`. However, the extra work to make the dependency explicit in the type isn't worth the effort. Indeed, giving up this dependency allows us to write the following program:

```
lmult : (1 n, m : Nat) -> Nat
lmult 0 m = let () = drop m in Z
lmult (S k) m = let (a, b) = copy m in a + lmult k b
```

Which is a lot simpler and achieves the same goal, it even has the same performance characteristics.

4.2 Permutations

The following example is certainly my favourite since it combines both dependent types and linear types in a way that wasn't possible before. It has been done in the context of my work for Statebox, a company that strongly relies on dependent types to write formally verified software.

One of their projects is a validator for petri-nets[43] and petri-net executions: FSM-oracle[13]. While the technical details of this project are outside the scope of this text, there is one aspect of it that is fundamentally linked with linear types, and that is the concept of permutation.

FSM-Oracle describes petri-nets using *hypergraphs* [44] , hypergraphs themselves use *permutations*[14] in order to model that a wire inside it can be moved around. This concept is key in a correct and proven implementation of hypergraphs. However, permutations turn out to be extremely complex to implement using only dependent types, as shown by the files trying to fit[15] their definition into a Category[16].

The relevant bit about permutation can be found in the two files `Permutations.idr` and `SwapDown.idr`, a permutation relies on a list of "swaps" that

[13] https://github.com/statebox/fsm-oracle

[14] https://github.com/statebox/fsm-oracle/blob/master/src/Permutations/Permutations.idr#L31

[15] https://github.com/statebox/fsm-oracle/blob/master/src/Permutations/PermutationsCategory.idr

[16] https://github.com/statebox/fsm-oracle/blob/master/src/Permutations/PermutationsStrictMonoidalCategory.idr

describe the operations that we can do on a list to generate a new permutation of the same list.

```
data Perm : {o : Type} -> List o -> List o -> Type where
  Nil : Perm [] []
  Ins : Perm xs ys -> SwapDown (a::ys) zs -> Perm (a::xs) zs

data SwapDown : List t -> List t -> Type where
  HereS  : SwapDown (a::as) (a::as)
  ThereS : SwapDown (a::as) bs -> SwapDown (a::b::as) (b::bs)
```

If you don't have access to the internet to witness the proofs in their full glory, here is an example of what we are dealing with:

```
permAssoc : (ab : Perm aas bbs) -> (bc : Perm bbs ccs)
         -> (cd : Perm ccs dds)
         -> permComp ab (permComp bc cd) = permComp (permComp ab bc) cd
permAssoc Nil bc cd = Refl
permAssoc (Ins {xs=as} {ys=bs} ab' abb) bc cd
  with (shuffle abb (permComp bc cd)) proof bdPrf
  | Ins {ys=ds} bd' add with (shuffle abb bc) proof bcPrf
    | Ins {ys=cs} bc' acc with (shuffle acc cd) proof cdPrf
      | Ins {ys=ds'} cd' ad'd =
        let (Refl, Refl, Refl) = shuffleComp abb bc cd bcPrf cdPrf bdPrf
         in insCong5 Refl Refl Refl (permAssoc ab' bc' cd') Refl
```

This function ensures that the composition of permutation is associative. It relies on helper functions such as `shuffleComp` which is postulated in Idris1 because it is too hard to implement.

Linear types can thankfully ease the pain by providing a very simple representation of permutations :

```
Permutation : Type -> Type
Permutation a = (1 ls : List a) -> List a
```

That is, a `Permutation` parameterised over a type `a` is a linear function from `List a` to `List a`[17].

This definition works because no elements from the input list can be omitted or reused for the output list. *Every single element* from the argument has to

[17]This has already been formally proven by Bob Atkey https://github.com/bobatkey/sorting-types/blob/master/agda/Linear.agda

find a new spot in the output list. Additionally, since the type `a` is unknown, no special value can be inserted in advance. Indeed, the only way to achieve this effect would be to pattern match on `a` and create values once `a` is known, but this would require `a` to be bound with a multiplicity greater than `0`:

```
fakePermutation : {a : Type} -> (1 _ : List a) -> List a
fakePermutatoin {a = Int} ls = 42 :: ls
fakePermutation {a = _} ls = reverse ls
```

In this example, `a` is bound with *unrestricted* multiplicity, which gives us the hint that it *is* inspected and the permutation might not be a legitimate permutation.

What's more, viewing permutations as a function gives it extremely simple categorical semantics: It is just an instance of the category of types with linear functions as morphisms.

Assuming `Category` is defined this way:

```
-- operator for composition
infix 2 .*.
-- operator for morphisms
infixr 1 ~>

record Category (obj : Type)  where
  constructor MkCategory
  (~>)          : obj -> obj -> Type -- morphism
  identity      : {0 a : obj} -> a ~> a
  (.*.)         : {0 a, b, c: obj}
                -> (a ~> b)
                -> (b ~> c)
                -> (a ~> c)
  leftIdentity  : {0 a, b : obj}
                -> (f : a ~> b)
                -> identity .*. f = f
  rightIdentity : {0 a, b : obj}
                -> (f : a ~> b)
                -> f .*. identity = f
  associativity : {0 a, b, c, d : obj}
                -> (f : a ~> b)
                -> (g : b ~> c)
                -> (h : c ~> d)
                -> f .*. (g .*. h) = (f .*. g) .*. h
```

We can write an instance of `Category` for `List o`:

```
Permutation : List o -> List o -> Type
Permutation a b = Same a b

permutationCategory : Category (List o)
permutationCategory = MkCategory
  Permutation
  sid
  linCompose
  linLeftIdentity
  linRightIdentity
  linAssoc
```

Using the definitions and lemmas for `Same` which is a data type that represents a linear function between two values of the same type:

```
-- a linear function between two values of the same type
record Same {0 o : Type} (input, output : o) where
  constructor MkSame
  func : LinearFn o o
  -- check the codomain of the function is correct
  check : (func `lapp` input) = output

sid : Same a a
sid = MkSame lid Refl

linCompose : {0 o : Type}
  -> {0 a, b, c : o}
  -> Same a b
  -> Same b c
  -> Same a c
linCompose (MkSame fn Refl) (MkSame gn Refl)
  = MkSame (lcomp fn gn) Refl

linRightIdentity : {0 o : Type}
   -> {0 a, b : o}
   -> (f : Same a b)
   -> linCompose f (MkSame Main.lid Refl) = f
linRightIdentity (MkSame (MkLin fn) Refl) = Refl

linLeftIdentity : {0 o : Type}
   -> {0 a, b : o}
   -> (f : Same a b)
   -> linCompose (MkSame Main.lid Refl) f = f
linLeftIdentity (MkSame (MkLin fn) Refl) = Refl

linAssoc : (f : Same a b) ->
           (g : Same b c) ->
           (h : Same c d) ->
           linCompose f (linCompose g h) = linCompose (linCompose f g) h
linAssoc (MkSame (MkLin fn) Refl)
         (MkSame (MkLin gn) Refl)
         (MkSame (MkLin hn) Refl) = Refl
```

`LinearFunction` is defined as follows:

```
record LinearFn (a, b : Type) where
  constructor MkLin
  fn : (1 _ : a) -> b

lid : LinearFn a a
lid = MkLin (\1 x => x)

lapp : LinearFn a b -> (1 _ : a) -> b
lapp f a = f.fn a

lcomp : LinearFn a b -> LinearFn b c -> LinearFn a c
lcomp f g = MkLin (\1 x => g.fn (f.fn x))
```

While this looks like a lot of code, the entire definition holds within 100 lines (including the `Category` definition), and a lot of the definitions like `LinearFn` and `Same` are generic enough to be reused in other modules.

Additionally, this approach is extremely straightforward. So much that in the future, it wouldn't seem extravagant to have the type-system automatically generate the code as part of a derived interface.

Finally and most importantly, this program could not exist without using both dependent types to declare the necessary proofs to formally verify our Categorical structure, and linear types to implement `Permutation`. It is a beautiful example where the two features meet and merge in a way that is both practical and elegant.

4.3 Levitation Improvements

The gentle art of levitation[45] shows that a dependently typed language has enough resources to describe all indexed data types with only a few constructors. The ability to define types as a language library rather than as a language feature allows a great deal of introspection which in turn allows a realm of possibilities. Indeed, we can now define operations on those data types that will preserve the semantics of the type but make the representation more efficient. We can generate interface implementations for them automatically. And we can use them to construct new types out of existing ones[46][47][48], including representations for primitive types such as `Int` or `String`, an approach that's been proven effective with Typedefs[49].

The practical guide to levitation[45] shows that those features are plagued by multiple shortcomings: the verbosity of the definitions not only make the data

declaration hard to write and read, it makes the compiler spend a lot of time constructing and checking those terms, and it has trouble identifying what is a type parameter and what is an index.

Thankfully, the performance inefficiency from levitation can be alleviated by a smart use of erasure. In his thesis, Ahmad Salim relies on erasing terms with the `.` (dot) syntax, which does its best but cannot enforce erasure of terms.

While this has not been implemented, it shows Idris2 has a lot of promise in lifting previous performance limitations. This is because, in Idris2, we can perform and enforce erasure by annotating proofs with `0` and use data types such as `Exists` instead of `DPair`.

More challenges arise when we try to use levitation to represent Idris data definitions. Indeed, linear and erased variables in constructors cannot be represented. We cannot represent the following constructor.

```
(::) : {n : Nat} -> {0 a : Type} -> a -> Vect n a -> Vect (S n) a
```

This suggests that levitation could be extended to support constructors with linear and erased arguments, but it is unknown if *levitation* itself (defining the description of linear data types in terms of itself) would be achievable.

Interestingly enough, encoding linearity in levitated description might also help fix one of the shortcomings of levitation in Idris: automatically discerning between type parameters and type indices. Combined with the ability to pattern match on types, in turn, would allow the Idris2 compiler to generate definitions for interfaces such as `Functor` and `Applicative`.

4.4 Compile-time String Concatenation

Strings are ubiquitous in programming. That is why a lot of programming languages have spent a considerable effort in optimising string usage and string API ergonomics. Most famously, Perl is notorious for its extensive and powerful string manipulation API including first-class regex support with more recent additions including built-in support for grammars.

One very popular feature to ease the ergonomics of string literals is *string interpolation*. String interpolation allows you to avoid this situation:

```
show (MyData arg1 arg2 arg3 arg4) =
    "MyData (" ++ show arg1 ++ " " ++ show arg2 ++ " " ++ show arg3 ++ ++ show arg4 ++ ")"
```

It allows string literals to include expressions *inline* and leave the compiler to build the expected string concatenation. One example of string interpolation syntax would look like this:

```
show (MyData arg1 arg2 arg3 arg4) = "MyData ({arg1} {arg2} {arg3} {arg4})"
```

The benefits are numerous, but I won't dwell on them here[18]. One of them however is quite unexpected: Predict compile-time concatenation with linear types.

As mentioned before, the intuition to understand the *erased linearity* `0` is to consider those terms absent at runtime but available at compile-time. In the case of string interpolation, this intuition becomes useful in informing the programmer when the compiler can perform compile-time concatenation.

```
let name = "Susan"
    greeting = "hello {name}" in
    putStrLn greeting
```

In the above example, it would be reasonable to expect the compiler to notice that the variable `name` is a string literal and that, because it is only used in a string interpolation statement, it can be concatenated at compile time. Effectively being equivalent to the following:

```
let greeting = "hello Susan" in
    putStrLn greeting
```

But this kind of translation can lead to very misleading beliefs about string interpolation and its performance implications. In this following example the compiler would *not* be able to perform the concatenation at compile time:

```
do name <- readLine
   putStrLn "hello {name}"
```

This is because the string comes from the *runtime*; indeed, static strings can be inserted at compile-time while strings from the runtime need to be concatenated. This means the following program typechecks:

[18] A proposal to implement this in the compiler is under way https://github.com/idris-lang/Idris2/issues/555.

```
let 0 name = "Susan"
    1 greeting = "hello {name}" in
    putStrLn greeting
```

Since the variable `name` has linearity `0`, it cannot appear at runtime, which means it cannot be concatenated with the string `"hello "`, which means the only way this program compiles is if the string `"Susan"` is inlined with the string `"hello "` at compile-time.

Using holes we can describe exactly what would happen in different circumstances. As a rule, string interpolation would do its best to avoid allocating memory and performing operations at runtime. Much like our previous optimisation, it would look for values that are constructed in scope and simply concatenate the string without counting it as a use.

```
let 1 name = "Susan"
    1 greeting = "hello {name}" in
    putStrLn greeting
```

Would result in the error:

```
> There are 0 uses of linear variable name
```

Adding a hole at the end would show:

```
let 1 name = "Susan"
    1 greeting = "hello {name}" in
    ?interpolation
```

```
> 1 name : String
> 1 greeting : String
> --------------------------
> interpolation : String
```

As you can see, the variable `name` has not been consumed by the string interpolation since this transformation happens at compile time.

Having the string come from a function call, however, means we do not know if it has been shared before or not, which means we cannot guarantee (unless we restrict our programming language further) that the string was not shared before, and therefore the string cannot be replaced at compile time.

```
greet : (1 n : String) -> String
greet name = let 1 greeting = "hello {name}" in ?consumed
```

```
> 0 name : String
> 1 greeting : String
> ----------------------------
> consumed : String
```

The string `name` has been consumed and the core will therefore perform a runtime concatenation.

5 Quantitative Type Theory and Programming Ergonomics

In section 4 we highlighted uses of linear types in Idris2. In this section, we are going to show the limitations of linear types in terms of ergonomics and provide solutions for them. A lot of these limitations stem from our intent to use linear types for performance analysis in section 6 and are discussed in detail in section 6.6.

5.1 Mapping Primitive to Data Types and Vice-versa

At the end of section 3.3 we encountered a problem with primitive types and linearity. We could not implement `copy` and `drop` because we do not have access to the constructors of primitive types.

This is because those types are not defined using the `data` syntax used for normal declarations. Rather, they are assumed to exist by the compiler and are built using custom functions which are different for each backend. If only `String` and `Int` were defined as plain data types, we could implement functions such as `copy` and `drop`.

It turns out this is possible, we can use a clever encoding that maps plain data types to primitive types and have the compiler "pretend" they are plain data types until the codegen phase where they are substituted by their primitive variants. Just like in *Haskell*, `String` could be represented as a `List` of `Char`, and `Char` could be represented as `Vect 8 Bool`. Using both those definitions our primitive types have now become plain data with regular constructors:

```
-- A bit only has two states, like a boolean
Bit : Type
Bit = Bool

-- An int is a vector of 32 bits
Int32 : Type
Int32 = Vect 32 Bit

-- A character is a vector of 8 bits
Char : Type
Char = Vect 8 Bit

-- A string is a list of characters of arbitrary length
String : Type
String = List Char
```

This allows us to implement `Copy` and `Drop` by inheriting the instance from `List` and `Vect`. Additionally, since the procedure to implement those instances is very mechanical, it could almost certainly be automatically derived.

This approach is reminiscent of projects like levitation[45], Typedefs[19], or containers[48] , which describe data types as data structures. And the benefits of this approach would translate quite well to our situation, beyond the ability to dance around linearity with primitive types.

Indeed, once our types are represented as data types we can use their structure to infer its properties. For example, if a data type has *type parameters*, then we can generate instances for `Functor`, `Applicative`, etc. If the data type resembles `List Char` , then we can replace it by the primitive type `String`. Finally, we can use semantic-preserving operations on our data types in order to optimise their representations in the generated code. For example, `data Options = Folder Int | Directory Int` is equivalent to `Vect 33 Bool` which can be represented as an unboxed `Int64` . Even more optimisations could be performed by mapping `Vect` of constant length to buffers of memory of constant size and index through them in `O(1)` instead of `O(n)`.

As the cherry on top, this mapping would help the coverage checker to infer missing cases accurately for primitive types, allowing us to write proofs about primitive types easier. In the following example the Idris2 compiler is unable to check the coverage of all strings, even though it should:

[19] http://typedefs.com

```
-- Here we ensure that our string is either "True"
-- or "False".
data IsTrueOrFalse : String -> Type where
  IsTrue : IsTrueOrFalse "True"
  IsFalse : IsTrueOrFalse "False"

fromString : (str : String) -> IsTrueOrFalse str => Bool
fromString "True" @{IsTrue} = True
fromString "False" @{IsFalse} = False
```

However, the following works correctly, suggesting that the special treatment of primitives is the culprit.

```
data Bit = I | O

MyChar : Type
MyChar = Vect 8 Bit

MyString : Type
MyString = List MyChar

-- Here we ensure that our string is either the \0 character
-- or the \1 character.
data IsTrueOrFalse : MyString -> Type where
  IsTrue : IsTrueOrFalse [[O,O,O,O,O,O,O,O]]
  IsFalse : IsTrueOrFalse [[O,O,O,O,O,O,I,O]]

fromString : (str : MyString) -> IsTrueOrFalse str => Bool
fromString [[O,O,O,O,O,O,O,O]] @{IsTrue} = True
fromString [[O,O,O,O,O,O,I,O]] @{IsFalse} = False
```

5.2 Semirings and Their Semantics for Resource Tracking

Linear types allow us to declare a variable to be used either `0`, `1` or any number of times. However, we've seen this approach is pretty restrictive, in this thesis we are interested in how to use multiplicities and quantitative types for the benefit of the user in terms of ergonomics and performance.

This section is the result of our findings in section 6 about performance. While the results are encouraging, they suggest that more drastic changes are required in order to derive a competitive advantage over other programming environments, especially the ones targeted at systems-programming. However, making linear types easier to use is an obvious first step toward expressing the seman-

tics of performant programs in the type, and is the reason for the following development.

Quantitative type theory tracks usage through a generic semiring, any implementation of QTT can use any positive semiring for it to function. Idris2 uses the following semiring:

```
data ZeroOneOmega = Rig0 | Rig1 | RigW

-- Addition on the Semiring
rigPlus : ZeroOneOmega -> ZeroOneOmega -> ZeroOneOmega
rigPlus Rig0 a = a
rigPlus a Rig0 = a
rigPlus Rig1 a = RigW
rigPlus a Rig1 = RigW
rigPlus RigW RigW = RigW

-- Multiplication on the Semiring
rigMult : ZeroOneOmega -> ZeroOneOmega -> ZeroOneOmega
rigMult Rig0 _ = Rig0
rigMult _ Rig0 = Rig0
rigMult Rig1 a = a
rigMult a Rig1 = a
rigMult RigW RigW = RigW
```

This semiring captures the semantics of linear logic (with erasure), where terms have to be used exactly once, or can be completely unrestricted. But semirings are extremely generic structures. Here is an example of another semiring:

```
data Cycle5 = Zero | One | Two | Three | Four

add1 : Cycle5 -> Cycle5
add1 Zero = One
add1 One = Two
add1 Three = Four
add1 Four = Zero

add : Cycle5 -> Cycle5 -> Cycle5
add Zero = id
add One = add1
add Two = add1 . add1
add Three = add1 . add1 . add1
add Four = add1 . add1 . add1 . add1

mutl : Cycle5 -> Cycle5 -> Cycle5
mutl Zero = const Zero
mult One = id
mult Two = \x => x `add` x
mult Three = \x => x `add` x `add` x
mult Four = \x => x `add` x `add` x `add` x
```

Which is the finite group[20] of size 5. Using this semiring would mean that we cannot have more than 5 of every variable. It would also mean that sharing a variable too many times results in linearity `0` which makes this semiring unsuitable for resource tracking.

One very useful semiring is the `InfNat` semiring, the natural numbers equipped with a `top` value:

```
data InfNat = N Nat | Top
```

This semiring tracks precise uses that including the ones that are greater than `1` but finite. With `InfNat` as a semiring one could imagine implementing `copy` like so:

```
--      ┌ We changed our multiplicity to 2
--      ▼
copy : (2 v : a) -> (a, a)
copy v = (v, v)
```

This encoding of resources is extremely expressive since it encapsulates both

[20] A group is a semiring in which every element is invertible.

linearity and variable sharing. We are going to see how to use it for a very practical purpose in section 5.3. In the meantime, here is another very useful semiring:

```
data Interval = Interval InfNat InfNat
```

Using an interval as semiring has semantics akin to *affine types* and would allow us to write the following program:

```
--            ┌ We changed our multiplicity to a range
--          ┌─┴──┐
isEmpty : ([0..1] v : Maybe a) -> Bool
isEmpty Nothing = True   -- Fully consumed here
isEmpty (Just _) = False -- Value ignored here
```

There are infinitely many semirings, some make more sense than others. We could for example combine semirings in pairs since pairs of semirings are also semirings. Another semiring is showcased in *Granule*[10] which has a good example of *privacy access* as a resource semiring.

In conclusion, different semirings allow us to express different resource semantics. In the next few pages, I will explore how the *precise usage* semiring `InfNat` is useful and will show a refactoring that allows us to use it in Idris2.

5.3 Memory Management Semantics

This section summarises three competing strategies for a memory management model that does not rely on manual memory management (which is known to be the source of countless bugs and security issues).

Tracing garbage collection (which we will refer to as "GC") generally refers to algorithms of automatic memory management such as *mark & sweep* or to runtimes using them such as the *Boehm-GC*[21]. The greatest benefit of garbage collection is that the burden of memory management is completely lifted from the programmer. But this feature is not always good news, the benefits come at the expense of predictability and control.

Since the garbage collector runs automatically without any input from the programmer, there is no way to tell how often the garbage collector will run and for how long, just by looking at a program. What's more, garbage collection

[21].

does not happen instantly, some algorithms stop the execution of the program for an unknown amount of time to reclaim memory. And this process cannot be stopped or controlled. A number of programming patterns emerged as a result (Object-pool pattern), but instead of making programming easier, they exhibit a structure akin to manual memory management (mutate existing memory spaces, avoid memory leaks), which is exactly what GC advertises to eliminate.

On the other hand, Rust [50] manages to remove all manual memory management, but it comes at the cost of a restricted programming model based on borrowing and lifetimes. Functions define lifetimes that ensure the arguments will not be freed during the execution of the function and avoid use-after-free errors. Borrowing and move semantics are very close to linear type semantics and ensure no two mutable pointers are being accessed at the same time on the same memory space. This programming practice might seem a bit foreign at a cursory look, but in fact, it is similar to what is considered good programming practice in manual memory management programming languages like C++.

Finally, *reference counting* describes an automatic memory management system that tracks how many references runtime variables have. The runtime does this by pairing each heap-allocated value with a *counter* that will be incremented every time the value is shared and decremented every time the value is consumed. Once the counter reaches 0, the value is freed. This guarantees the maximum amount of memory is always available since objects are deleted immediately at the end of their lifetime, instead of having to wait for the garbage collector to delete them. While appealing, this approach has multiple shortcomings compared to garbage collection:

- It still has a runtime cost, since each dereferencing and aliasing induces a decrement/increment operation.
- A typical implementation does not support reference cycles, which may introduce memory leaks.
- The memory management is not completely removed from the control of the user since they now need to pay attention to cycles and lifetimes.

This list might look discouraging at first glance, but we have a solution to provide for each one of them.

While it is true that reference counting incurs a runtime cost, there are a lot of opportunities for optimisations. Not all aliasing and not all dereferencing needs to lead to an increment or decrement operation. As for cycles, Idris2 being a functional programming language, there are very few opportunities to construct

cyclical data structures.

As for the last shortcoming of reference counting, the following sections will focus on how to address manual memory management in a reference-counted runtime leveraging quantitative types. This will not remove the need for thinking about memory management, but it will make memory management a *linearity error* rather than an invisible property of programs. Finally, encouraging users to think about memory management need not be a bad thing. Such a powerful link between quantities and memory could make Idris2 competitive with other commercial programming language and even systems programming.

Putting Everything Together: Linearity and Reference Counting

The idea of mapping linear usage to reference counting isn't new[18], but with QTT, we can have much greater control over the semantics of the semiring we use to make it correspond precisely with the behaviour of our runtime.

Counting immutable beans[34] shows us how effective reference counting can be in a purely functional programming language. Their intuition and heuristics lead to performant code without affecting the source language. Using QTT we can improve this approach in two regards:

- Make the semantics of reference counting explicit in the type
- Remove the need for heuristics and formalise them in the type-system

Those two points mean the same thing; to illustrate this, let us take an example they use, `makePairOf`:

```
makePairOf : a -> (a, a)
makePairOf x = inc x ; (x, x)
--               ▲
--               └ Increment the reference count
```

This function is exactly the same as our `copy` function from the chapter on linearity, except it has an additional instruction `inc`. This instruction tells the runtime to *increase* the reference count of `x` because it's been duplicated. To understand the intuition behind this behaviour let us examine a simpler function, the identity function:

```
id : a -> a
id x = x
```

This function needs not allocating, freeing, incrementing, or decrementing of a variable or a counter. It simply takes the argument and returns it without modifications, its memory space is left untouched. Modifying this function to return a pair instead makes it so that we share *an additional reference* to our value. This additional reference must be accounted for and this is why `makePairOf` increments the reference count by one. If it were to return a triple, the variable's reference count would be increased by two.

We've seen how different semirings can have different semantics. If Idris2 used `Nat` as a semiring we could write the following program:

```
makePairOf : (2 x : a) -> (a, a)
makePairOf x = (x, x)
```

Without requiring a reference increment. Why? Because the *type signature* already informs the caller that `x` needs to have at least a reference count of 2 to be called by `makePairOf`. This defers the reference increment to the caller instead of the function. The deferring of reference counting is significant because it now allows the runtime to allocate a value with the correct reference count from the beginning, instead of allocating it with a reference count of 1 and then incrementing it for each sharing. Effectively cutting a large chunk of `inc` operations.

```
let 2 v = "hello" -- This already has a ref count of 2
 in makePairOf v -- this doesn't have any inc or dec operations
```

Another way to avoid `inc` and `dec` operations would be to allow *borrowing* of variables, that is, if a variable is not updated, then it is safe to share. In the following program we inspect a `Maybe` value and return if it contains a value or not:

```
isEmpty : Maybe a -> Bool
isEmpty Nothing = True
isEmpty (Just _) = False -- Argument unused
```

In the current implementation of Idris2, this function cannot be linear since it matches on its argument. However, one could make the argument that the *content* of the pattern match is not used. Therefore, the pattern match only inspects the tag of the constructor, not its fields, allowing the field to be safely shared elsewhere.

```
isEmpty : (& x : Maybe a) -> Bool
isEmpty Nothing = True
isEmpty (Just _)= False
```

In this example, we use the custom syntax `&` for the multiplicity to indicate that the value is borrowed. This tells the compiler that the value should *not* be erased from the runtime, but that calling this function does not constitute a use, just like the `0` quantity. This would make the following program legal:

```
let 1 useOnce = Nothing in
    (isEmpty useOnce, useOnce)
```

In which, even if `useOnce` is used twice, the first usage is borrowed and therefore does not constitute a use.

Finally, there is another way to remove `inc` and `dec` operations leveraging quantities, and that is to use the resource tracking we do at compile time to ensure the liveliness of our variables (much like the lifetime checker of Rust) and use reference counting only when the compiler detects a variable has been spent. In the following example, we allocate a value to be used 5 times, we also have access to functions that use our variable 2 times and 3 times. If we were to simply use linearity as our reference counter we would expect 5 `dec` operation. Fortunately, we can reduce this to 2 `dec` operations:

```
useTwice : (2 _ : Nat) -> ()
useTwice n = let () = drop n in
                drop n -- usage count dropped to 0, we can `dec`

useThrice : (3 _ : Nat) -> ()
useThrice n = let () = drop n
                  () = drop n
              in drop n -- usage count dropped to 0, we can `dec`

let 5 v = [0 .. 99999] in
let () = useTwice in -- one `dec` here
    useThrice -- and a second one here
```

Combining our first and last optimisation we can reduce the amount of `dec` from 5 to 2 and the amount of `inc` from 5 to 0.

5.4 Implementing Generic Semirings

In the following I am going to showcase the biggest changes necessary to implement resource tracking through different semirings in a fully fledged programming language such as Idris. Going through this exercise made me discover additional structures for proper resource tracking in a practical programming language (as opposed to a theorem prover). The goal of this section is to demonstrate the role of those additional structures and explain why they are necessary.

The current Idris2 compiler uses `data RigCount = Rig0 | Rig1 | RigW` for linearity `0`, `1`, `ω` respectively. To support a different semiring we need to abstract over the operations of semirings and update the existing codebase to make use of them instead of relying on our definition of `RigCount`.

A `Semiring`[22] is defined as follows:

```
interface Semiring a where
  (|+|) : a -> a -> a -- Addition
  plusNeutral : a     -- Neutral for addition
  (|*|) : a -> a -> a -- Multiplication
  timesNeutral : a    -- Neutral for multiplication
```

Where `plusNeutral` is the neutral element for `|+|` and `timesNeutral` is the neutral element for `|*|`. `|+|` and `|*|` are binary functions which distribute as expected.

This change is quite disruptive since it forbids us to pattern match on `RigCount` and instead forces us to be entirely generic on the multiplicity we use. Signatures like:

```
linearCheck : RigCount -> RigCount -> Bool
```

Become:

```
linearCheck : Semiring r => r -> r -> Bool
```

To recover the functionality from pattern matching we introduce an eliminator for `Semiring`, `elimSemi`:

[22] As a reminder, a semi ring is defined by two operations, `+` and `*` and have neutral elements `0` and `1` such that `0 * n = 0`, `1 * n = n` and `0 + n = n`. See the glossary for more details.

```
elimSemi : (Semiring a, Eq a) => (zero : b) -> (one : b) -> (a -> b)
       -> a -> b
elimSemi zero one other r {a} =
  if r == Semiring.plusNeutral {a}
     then zero
     else if r == Semiring.timesNeutral {a}
             then one
             else other r
```

Which reads "if we are looking at the neutral element for `+`, then return the value associated with zero, otherwise if we are looking at the neutral element for $*$, then return the value associated with one, otherwise, we are looking at a number that is not one of the neutrals from our semiring, and we need to eliminate it using our function `other`".

To replace pattern matching we also need a set of helper functions. They are all based upon `elimSemi` and all represent different semantics associated with our generic semiring. The most important ones are `isLinear` and `isErased` since they explicitly spell what we expect from the semantics of the language:

```
-- Returns True if the value is the neutral for addition
isErased : (Semiring a, Eq a) => a -> Bool
isErased = elimSemi True False (const False)

-- Returns True if the value is the neutral for multiplication
isLinear : (Semiring a, Eq a) => a -> Bool
isLinear = elimSemi False True (const False)

-- Returns True if the value is not a neutral for any operation
isRigOther : (Semiring a, Eq a) => a -> Bool
isRigOther = elimSemi False False (const True)
```

Our first discovery is that those functions are not enough! Indeed, the compiler also makes assumptions about *ordering* within the multiplicity. An assumption that was not visible before our generalisation and isn't directly implemented in QTT. In the following program, we check that multiplicities are compatible by checking if the left one is smaller than the right one.

```
rigSafe : RigCount -> RigCount -> Core ()
  rigSafe Rig1 RigW = throw
    (LinearMisuse fc !(getFullName x) Rig1 RigW)
    -- `x` comes from the enclosing scope
rigSafe Rig0 RigW = throw
    (LinearMisuse fc !(getFullName x) Rig0 RigW)
rigSafe Rig0 Rig1 = throw
    (LinearMisuse fc !(getFullName x) Rig0 Rig1)
rigSafe _ _ = pure ()
```

You can see that pattern matching doesn't spell out the relation between the first and the second argument. However, the same program can be refactored using an ordering on a generic semiring:

```
rigSafe : Semiring r => r -> r -> Core ()
rigSafe lhs rhs =
  when (lhs < rhs) -- Here is where we need an ordering relation
       (throw (LinearMisuse fc !(getFullName x) lhs rhs))
```

This now reads much more naturally as "if the left resource is smaller than the right one, then throw an error".

Semirings in general do not enforce an order, and we do not want to add too many restrictions to our multiplicity types to avoid making it less flexible than it needs. Typically, if affine types were to be implemented using intervals, we could not rely on a *total order*[23] since some intervals cannot be compared with each other. For example, `[3..5]` and `[0..4]` are not directly related to each other. For this reason, we are going to use a `Preorder` which only requires equality to be valid for some pairs of values and requires transitivity and associativity to hold. In Idris2 this is formulated as:

[23] A *total order* is an ordering for which every value is related to every other value, unlike a partial order where only some values are comparable.

```
interface Preorder a where

  -- the ordering relation
  (<=) : a -> a -> Maybe Bool

  -- The proofs of reflexivity and transitivity of the ordering
  preorderRefl : {x : a} -> x <= x = Just True
  preorderTrans : {x, y, z : a} -> x <= y = Just True
                                -> y <= z = Just True
                                -> x <= z = Just True
```

For engineering reasons, this is not exactly the version used in the Idris2 compiler but this should not detract from the observation that a semiring alone is not enough.

Another missing piece is a default value. In a lot of the existing implementation `RigW` is used as a default value when creating values with unknown linearity. This is because Idris2 is *unrestricted by default* instead of *linear by default*. In our case, the default value has a very neat property: it is greater or equal to every other value of the semiring. In technical terms, in addition to having a semiring and a preorder we also need our multiplicity type to have a *top* value, which means we are dealing with a *meet semi-lattice*.

`top` is easily defined as follows:

```
interface Preorder a => Top a where
  -- top is the default multiplicity
  top : a
  -- top absolves everything else
  topAbs : {x : a} -> x <= top = Just True
```

Now that we finally have all the pieces required by the Idris2 compiler we can replace the existing semiring `0`, `1`, `ω` by any other that obeys the laws of a meet-semi-lattice.

Finally, looking at how linearity interacts with pattern matching we see another restriction show up. Where previously we would see:

```
checkUsageOK used Rig0 = pure ()
checkUsageOK used RigW = pure ()
checkUsageOK used Rig1
         = if used == 1
              then pure ()
              else throw (LinearUsed fc used nm)
```

We now have:

```
checkUsageOK used r = when (isLinear r && used /= 1)
                           (throw (LinearUsed fc used nm))
```

Again, where the first snippet is unclear about the semantics of the operation, the second one reveals that we are checking if the number of uses matches the linearity. Or in other words, "if the multiplicity is linear, and we have not used the variable exactly once, then throw an error".

This suggests that to implement pattern matching for a generic semiring, we also need to add a way to associate the multiplicity with a natural number. This way we can compare the value of the resource with its number of occurrences. Or we need a concept of *leftover typing* where a resource can be partially consumed via subtracting. That is, if we have `5` of a resource, then we can partially use it `3` times, leaving us with `5 - 3 = 2` resources.

Next Steps

This exercise was particularly informative in teaching us what is *really needed* in practice to build up a programming language based on QTT. *Meet semi-lattices* aren't mentioned in Quantitative Type Theory[1] but they appear in Granule [10]; while this is not entirely surprising, it is noteworthy that both approaches reach the same conclusion: ordered semirings are the foundation of a sound implementation of a calculus with quantities into a useful programming language.

This implementation is only the first step into extending the language for full reference counting without garbage collection, in the meantime, there are a lot of intermediate steps. Additionally, now that the seal is broken on multiple types of multiplicities, nothing stops us from implementing different multiplicities for Idris2 and seeing what happens. We can now discover semantics for semirings, rather than use different semirings to reach different semantics. Granule already has products, intervals, access variables and natural numbers in addition to

linearity as their multiplicity. It would be interesting to see what matrices or polynomials offer in terms of semantics as a resource tracking semiring.

The immediate next step for this project is to repeat this experiment (generalising some aspect of the compiler) but with linearity checking rather than semirings. Indeed, given an implementation of infinite Nats as a semiring, we should be able to replace our previous semiring by the new one and enjoy the benefits in terms of added functionality.

Unfortunately, while the compiler accepts Nat-multiplicity programs, it keeps the semantics of `0`, `1`, `ω` because everything bigger than 1 is considered unrestricted in the linearity checker.

6 Performance Improvement Using Linear Types

This section aims to answer the question "what performance improvement can we get by using linear types *today*?" by implementing a well-known optimisation technique that linear types allow: memory reuse.

First, we are going to talk about the assumptions necessary for our optimisation to work. As we will see, they are not as simple as one might expect. The original formulation for linear types suggests that every linear function could reuse the memory space taken by linear variables. But this is not necessarily the case in Idris2 and additional requirements will have to be met for the optimisation to trigger. We will see in the conclusion that those requirements are quite limiting and prompted the work in section 5.

After clarifying our needs regarding our optimisation, we are going to look at the compiler changes necessary to implement our optimisation. This section is particularly useful as a compiler-engineering challenge and might help anyone who is interested in implementing a similar optimisation in other languages featuring linear types (such as linear Haskell for example).

Then we will talk about the methodology employed and the expected results from testing our optimisation, this will involve benchmarking and some discussion on the nature of the tests.

Finally, the results are presented along with their discussion. The discussion helps us understand the chronology in which those tests were performed and motivate our conclusion which will be done in section 6.6.

6.1 Conditions Under Which We Can Run Our Optimisation

Idris2 already allows defining linear functions, but despite the extra information that the argument cannot be used twice, the compiler does not perform any special optimisation for linear types. One obvious observation one can make is that once a linear value has been consumed, its memory space can be reclaimed, or reused. In practice this manifests this way:

```
isTrue : (1 _ : a) -> (f : (1 _ : a) -> Bool) -> String
isTrue arg predicate
--                ┌ We can free `arg` after this
--                ▼
  = if predicate arg then "It's true!"
                     else "Fake news."
```

The value `arg` can be freed immediately after being called by `predicate`. Indeed, it has been used, and therefore it cannot be used again, which means its memory space won't ever be accessed again. In what follows we are going to reuse the intuition originally suggested by [15][16][17] and rephrase it as :

> If you own a linear variable, and it is not shared, you can freely mutate it

Freeing memory might not sound very exciting, but the main benefit is that we do not need to wait for the garbage collector to notice our value can be freed. Removing the reliance on garbage collection is a huge step toward predictable performance. The same intuition can be used for a safe-update function:

```
data Numbers = Inc Nat | Dec Nat

update : (1 _ : Numbers) -> Numbers
update (Inc n) = Inc (S n)
update (Dec Z) = Dec Z
update (Dec (S n)) = Dec n
```

We have a data type `Numbers` with two cases and `update` checks which case we have and then increments or decrements the `Nat` value accordingly.

This function should perform in constant space (`O(1)`) but currently, the Idris compiler always allocates new values when a constructor is called. In addition, the old value is now ready to be freed, but we have to wait on the garbage collector to catch it.

Unfortunately, we cannot naively implement the free and update optimisation to see what kind of performance improvement we can get out of linear types. This is because a linear variable is not guaranteed to be unique when it is called on a linear function. One example of this sharing mechanism can be seen with function subtyping.

Issues with Linear Function Subtyping

To make the interplay between unrestricted and linear functions easier, Idris2 features subtyping on functions. As a refresher, subtyping allows a function to accept another type than the one that it has been specified with, as long as the other type is a *subtype* of the original one. In the following example we are going to assume we have access to two types, `A` and `B`, where `B` is a subtype of `A`, noted with `B <: A`

```
f : A -> A

g : B -> B

-- Assume we have B <: A

let a : A = …
    b : B = …
    y1 = f a -- Expected
    y2 = f b -- b is a subtype, it's valid
    no = g a -- a is a supertype of b, invalid
 in ?rest
```

In Idris2, types cannot have a subtyping relation, except function types. In fact, linear functions are considered to be a subtype of unrestricted functions. In formal notation it means `((1 _ : a) -> b) <: (a -> b)`. This detail is extremely relevant for our optimisation because it means that *linearity does not guarantee uniqueness*, which is the property we are interested in when performing those optimisations.
The motivation for this subtyping is immediately visible when using them as higher-order functions.

```
map : (f : a -> b) -> List a -> List b
linearMap : (f : (1 _ : a) -> b) -> (1 _ : List a) -> List b
```

Those functions do the same thing but won't typecheck the same way. Given an unrestricted `f` the second function will refuse to typecheck. Whereas feeding a linear function to `map` will compile correctly.

```
inc : Nat -> Nat

linInc : (1 _ : Nat) -> Nat

-- Success because everything unrestricted
map inc [1,2,3]

-- Success because got linear and expected unrestricted
map linInc [1,2,3]

-- Fail because inc is unrestricted but linMap expected linear
linearMap inc [1,2,3]

-- Success because everything is linear
linearMap linInc [1,2,3]
```

In the following example we show how this breaks down our assumption of safe updates:

```
update : (1 _ : Nat) -> Nat
update n = ... -- The body of the function might assume that `n`
               -- is unique but this is not true when `update`
               -- is passed as argument to a function like in the
               -- following

do let list1 = [1,2,3]
   let list2 = list1
   printLn $ map update list1
   printLn list2
```

This program typechecks, even if we are using a linear function in an unrestricted setting. We expect the output to be:

```
> [2, 3, 4]
> [1, 2, 3]
```

But if the update was performed naively we would get this instead:

```
> [2, 3, 4]
> [2, 3, 4]
```

This suggests that our intuition that linear functions can be used for safe updates

and safe memory frees is not restrictive enough. We need an additional level of restriction to ensure *uniqueness* of the values instead of linearity.

To ensure uniqueness, we are only going to consider situations where a variable has been created in scope and is declared to be linear. In the next section, I will talk about the changes that are necessary to a compiler for a functional language such as Idris. The changes should reflect the nature of our restrictions in a language that features pattern matching, a core lambda-calculus, and immutable data structures.

6.2 Implementing Our Optimisation

Here is a concrete example of the situation we can optimise using linear types with Idris:

```
update : (1 _ : List a) -> List a

let 1 v = x :: xs in
    update v
```

As we've seen in the previous section, uniqueness is ensured by the fact that `v` is declared in the immediate scope of its use, and is annotated with linearity 1.

The changes to the Idris AST are as follows:

- Allow constructors to know if they are bound linearly
- Allow linear functions to mutate constructors that are bound linearly
- Add a new instruction to the backend AST to mutate values

Conceptually the compiler architecture is quite simple, in the following illustration we link trees transformations with their corresponding functions. Each tree is its own data declaration:

`PTerm` represents the surface level language, it is desugared into `RawImp` which is a front end for our typed terms `Term`. `Term` is indexed over the list of variables in context, which helps avoid indexing errors (especially when manipulating de Bruijn indices) and makes explicit the rules of context extension (for example when binding arguments under a lambda). `CExp` is a tree of compiled expressions ready for codegen, it keeps the index for variables from `Term` and the index is then dropped when variable indices are replaced by their names in `NamedCExp`. Which is the same but with all variable substitutions from context performed.

Since our changes do not touch the type checker or elaborator, they do not

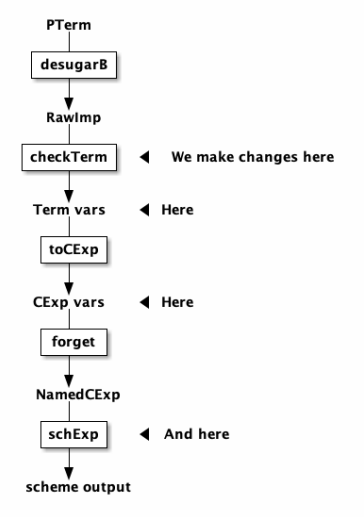


Figure 1: The entire compiler pipeline

change the `checkTerm` stage. However, we do need information from the control flow and we do need to access the case tree to perform our changes between `Term` and `CExp`.

Adding the Mutating Flag

Since this optimisation cannot be triggered automatically for every linear function, we are going to tell the compiler where to apply it using a flag directive to functions we want to optimise. In practice, we want the following program to compile and be optimised:

```
%mutating
update : Ty -> Ty
update (ValOnce v) = ValOnce (S v)
update (ValTwice v w) = ValTwice (S v) (S (S w))
```

Here, the `%mutating` annotation indicates that the value manipulated will be subject to mutation rather than construction.

If we were to write this code in a low-level c-like syntax we would like to go from the non-mutating version here:

```
void * update(v * void) {
    Ty* newv;
    if v->tag == 0 {
        newv = malloc(sizeof(ValOnce)); // Allocation here
        newv->tag = 0;
        newv->val1 = 1 + v->val1;
    } else {
        newv = malloc(sizeof(ValTwice)); // Allocation here
        newv->tag = 1;
        newv->val1 = 1 + v->val1;
        newv->val2 = 1 + 1 + v->val2;
    }
    return newv;
}
```

To the more efficient mutating version here:

```
void * update(v * void) {
    if v->tag == 0 {
        v->val1 = 1 + v->val1;
    } else {
        v->val1 = 1 + v->val1;
        v->val2 = 1 + 1 + v->val2;
    }
    return v; // Return the mutated argument
}
```

The two programs are very similar, but the second one mutates the argument directly instead of mutating a new copy of it.

There is however a very important limitation:

Only mutate uses of the constructor we are matching on The following program would see no mutation since the constructor we are matching does not appear on the right-hand side.

```
%mutating
update : Ty -> Ty
update (ValTwice v w) = ValOnce (S v)
update (ValOnce v) = ValTwice (S (S v)) (S (S w))
```

This is to avoid implicit allocation when we mutate a constructor which has more fields than the one we are given. Or avoid memory corruption when updating a field outside of the allocated buffer. Imagine representing data as records:

```
ValOnce = { tag : Int , val1 : Int }
ValTwice = { tag : Int , val1 : Int val2 : Int }
```

If we are given a value `ValOnce` and asked to mutate it into a value `ValTwice` we would have to allocate more space to accommodate for the extra `val2` field.

Similarly, if we are given a `ValTwice` and are asked to mutate it into a value `ValOnce` we would have to carry over extra memory space that will remain unused. In the worst case, reckless access to the fields in memory would result in accessing out of bound memory and provoke either segfault or memory corruption. For example, while attempting to mutate the second argument of a constructor that has only one.

Ideally, our compiler would be able to identify data declaration that shares the same layout and replace allocation for them by mutation, but for the purpose of this thesis, we will ignore this optimisation and carefully design our benchmarks to make use of it.

Mutating Instruction in the AST

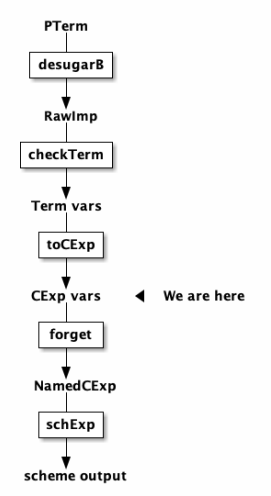


Figure 2: Location of backend AST changes (CMut)

For this to work, we need to add a new constructor to the AST that represents *compiled* programs `CExp`. We add the constructor `CMut`:

```
CMut : (ref : Name) -> (args : List (CExp vars)) -> CExp vars
```

It represents mutation of a variable identified by its `Name` in context and using the argument list to modify each of its fields.

Once this change reaches the code generator, it needs to output a `mutation` instruction rather than an allocation operation. Here is the code for the scheme backend:

```
||| Mutates the given vector at the given index
mutateValue : (ref : String) -> (index : Nat) -> String -> String
mutateValue ref idx newVal =
  "(vector-set! " ++ ref ++  " " ++ show idx ++ " " ++ newVal ++ ")"

||| Mutate all fiels of a given constructor
||| We skip the first value in the vector because we do not change
||| @vecRef : The vector to update
||| @args : The list of new arguments
schMutate : (ref : String) -> (args : List String) -> String
schMutate ref args =
  -- we start indexing at 1 since 0 is the tag and doesn't change
  let indices = [1 .. (length args + 1)]
      zipped : List (Nat, String) = zip indices args
      mutation = (showSep " " (map (uncurry $ mutateValue ref) zipped)) in
      "(begin " ++ mutation ++ " " ++ ref ++ ")"
```

As you can see we generate one instruction per field to mutate as well as the final instruction to *return* the value passed in argument, this is to keep the semantics of the existing assumption about constructing new values. The function `schMutate` is then called to output the scheme program from a `NamedCExp`:

```
schExp i (NmMut fc ref args)
    = schMutate (schName ref) <$> (traverse (schExp i) args)
```

Tracking Lost References

There is however an additional detail that hasn't been expressed and that is how to *construct* a value of `CMut`. `CMut` is only constructed in the following circumstances:

- The function is linear.
- The function is annotated with `%mutating`.

- The function uses the same constructor on the left-hand side and the right-hand side of its definition.
- The value passed in argument is *unique*.

Even though the first one is obvious, it will be left out for trivial reasons: lots of the standard library is not defined linearly such that the following will not compile:

```
linInc : (1 _ : Int) -> Int
linInc n = n + 1
```

This is because `(+)` is not declared linearly. This could be solved with an alternative standard library that exposes a linear API.

The second condition is easy to implement as a boolean test but requires a bit of thought about *where* to put it. Since the next condition requires us to change the *case tree* of our function definition, we are going to add the boolean test in `src/TTImp/ProcessDef.idr` which will process every definition of our program.

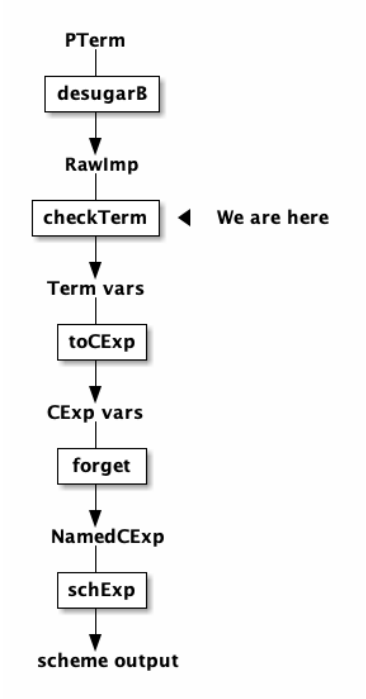


Figure 3: Location of case tree changes

We are going to short-circuit the case-tree generation by replacing the compiled one by our modified version of it:

```
--             ┌ The previously compiled case tree
--         ┌──────┐
(rargs ** (tree_rt', _)) <- getPMDef …
tree_rt <- if Mutating `elem` flags gdef
              then makeMutating tree_rt'
              else pure tree_rt'
```

The third constraint is the most important one without which we corrupt the memory of our program as we've seen earlier.

Let's look at our `update` function to understand the nature of the changes and change it slightly:

```
%mutating
update : (1 _ : Ty) -> Ty
update arg = case arg of -- Pattern matching on arg
                 ValTwice v => ValTwice (S (S v))
                 ValOnce v => ValOnce (S v)
```

This version makes use of a temporary variable `arg` instead of matching on the function argument directly. The calls to constructors `ValTwice` and `ValOnce` allocate memory, but we want to replace them with `CMut` which will reuse memory. To reuse the memory space taken by `arg` they need to store a pointer to it. This is reflected in the signature of `CMut : FC -> (ref : Name) -> List (CExp vars) -> CExp vars` where the second argument is the reference to reuse.

```
case arg of
     ValTwice v w => -- Access `arg` and mutate it with S (S v)
                     ValTwice (S (S v)) (S (S w))
     ValOnce v => -- Access `arg` and mutate it with S v
                  ValOnce (S v)
```

However, looking at the AST for pattern matching clauses we see that it does not carry any information about the original value that was matched:

```
data CaseAlt : List Name -> Type where

      -- This is the case for constructor matches
      -- There is no reference to the variable we inspect
      ConCase : Name -> (tag : Int) -> (args : List Name) ->
                CaseTree (args ++ vars) -> CaseAlt vars

      DelayCase : (ty : Name) -> (arg : Name) ->
                  CaseTree (ty :: arg :: vars) -> CaseAlt vars

      ConstCase : Constant -> CaseTree vars -> CaseAlt vars

      DefaultCase : CaseTree vars -> CaseAlt vars
```

Indeed, the `Name` we see refers to the constructor we are matching on, and not to the variable name we are inspecting.

Thankfully this reference can be found earlier in `CaseTree` :

```
  data CaseTree : List Name -> Type where

      Case : {name, vars : _} ->
             (idx : Nat) -> -- Here
             (0 p : IsVar name idx vars) ->
             (scTy : Term vars) -> List (CaseAlt vars) ->
             CaseTree vars
      STerm : Int -> Term vars -> CaseTree vars

      Unmatched : (msg : String) -> CaseTree vars

      Impossible : CaseTree vars
```

The reference is carried by `idx` which is the index of the variable in its context, as supported by the proof `p`. Once we get a hold of this reference, we can move on to the actual tree transformation. The transformation itself occurs in `src/Core/CaseBuilder.idr` and can be summarised with this function:

```
replaceConstructor : (cName : Name) -> (tag : Int) ->
                     (rhs : Term vars) ->
                     Core (Term vars)
replaceConstructor cName tag
  (App fc (Ref fc' (DataCon nref t arity) nm) arg) =
    if cName == nm
       then pure (App fc (Ref fc'
                     --           ┌ here is the change
                     --       ┌───┘
                     (DataCon (Just ref) t arity) nm) arg)
        else App fc (Ref fc' (DataCon nref t arity) nm)
         <$> replaceConstructor cName tag arg
```

Which will replace every application of a data constructor with one which has a reference to the variable we mutate `ref` as indicated by the `Just ref`.

This refers to another AST change in how the `Term` data type needs to change in order to track which data constructors are allowed to mutate a reference instead of constructing a new value.

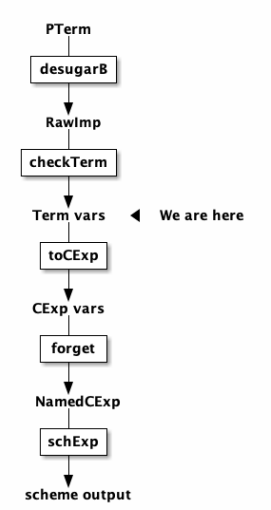


Figure 4: Location of the core calculus changes

For reference here is the `Term` data type indexed on the list of names in context. This represents the core calculus of Idris and closely resembles a typical lambda calculus with `Bind` for lambda abstraction and `App` for application.

```
data Term : List Name -> Type where
     Local : FC -> (isLet : Maybe Bool) ->
             (idx : Nat) -> (0 p : IsVar name idx vars) -> Term vars
     -- I our case, we're interested in references that
     -- point to a Data constructor
     Ref : FC -> NameType -> (name : Name) -> Term vars
     Meta : FC -> Name -> Int -> List (Term vars) -> Term vars
     Bind : FC -> (x : Name) ->
            (b : Binder (Term vars)) ->
            (scope : Term (x :: vars)) -> Term vars
     App : FC -> (fn : Term vars) -> (arg : Term vars) -> Term vars
```

The `Ref` constructor has a `NameType` argument which is defined as follows:

```
data NameType : Type where
     Bound   : NameType
     Func    : NameType
     -- DataCon refers to data constructors
     DataCon : (tag : Int) -> (arity : Nat) -> NameType
     TyCon   : (tag : Int) -> (arity : Nat) -> NameType
```

We are interested in the `DataCon` constructor that indicates that our `Ref` is a data constructor. The change we need to make is to add a reference to the name of the variable in scope that we can update :

```
--                ┌ If we allow mutation, the variable to
--                │ mutate will be here as a `Just`
--        ┌───────┴─────────┐
DataCon : (ref : Maybe Name) -> (tag : Int) -> (arity : Nat) -> NameType`.
```

The last step of this chain of changes is to inspect the `DataCon` constructor during compilation and emit the correct `CMut` instruction. We do this in `toCExp`:

```
  toCExpTm n (Ref fc (DataCon (Just ref) tag arity) fn)
      = pure $ CMut fc ref []
```

Ensuring Uniqueness

The final piece of the puzzle is to ensure variables are *unique*, and for this, we are going to use the property that linear variables that have been constructed in scope are unique:

```
let 1 v = MkValue 3 in
    f v -- v is unique
```

In order to track this change, we are going to update our `DataCon` once again to carry the linearity information necessary `DataCon : (rig : RigCount) -> (ref : Maybe Name) -> (tag : Int) -> (arity : Nat) -> NameType`. This step is done when checking let-bindings in `TTImp/Elab/Binders.idr`. We implement a function that looks for a data constructor in application position:

```
lineariseDataCon : RigCount -> Term vars
                -> Maybe (Term vars)
lineariseDataCon rig
  (App fc (Ref fc' (DataCon r ref tag ary) name) arg) =
    toMaybe
      (rig /= r) --              ┌ Replace the multiplicity inferred
                 --              │ by the one given
                 --              ▼
      (App fc (Ref fc' (DataCon rig ref tag ary) name) arg)
lineariseDataCon _ _ = Nothing
```

And call the function when let-bindings are being type-checked.

An astute reader might notice that this implementation has multiple shortcomings:

- The unique tag isn't removed from `DataCon` once they leave the let scope, allowing the mutating mechanism to trigger on values that have been shared.
- The `%mutating` flag does not emit compile errors when used on functions called with non-unique variables.
- The `%mutating` flag does not emit compile errors when used on non-linear functions or when the optimisation does not run.
- The optimisation is incompatible with "newtype-optimisation" which removes needless pattern matching and constructing of values when they have a single constructor and a single argument.

Thankfully, most of those limitations are a matter of user ergonomics and not program performance. This means we are now ready to benchmark linear programs!

6.3 Methodology

In order to test the effectiveness of this optimisation, I ran a series of benchmarks using a modified version of the Idris2 compiler. The changes are summarised in section 6.2. This section presents the methodology employed to measure the effectiveness of the optimisation.

First I am going to talk about the core assumptions necessary to measure performance. Then I will present 3 benchmark programs that will help verify or contradict our assumption about performance. Finally, I will present the tools used to run the benchmarks and the conditions in which they were run.

Core Assumptions

Our expectation is that our programs will run faster for 2 reasons:

- Allocation is slower than mutation
- Mutation avoids short-lived variables that need to be garbage collected

Indeed, allocation will always be slower than simply updating parts of the memory. Memory allocation requires finding a new memory spot that is big enough, writing to it, and then returning the pointer to that new memory address. Sometimes, allocating big buffers will trigger a reshuffling of the memory layout because the available memory is so fragmented that a single continuous buffer of memory of the right size isn't available.

Obviously, all those behaviours are hidden from the programmer through *virtual memory* which allows us to completely ignore the details of how memory is actually laid out and shared between processes. Operating systems do a great job at sandboxing memory space and avoiding unsafe memory operations. Still, those operations happen and make the performance of a program a lot less consistent than if we did not have to deal with it.

In addition, creating lots of short-lived objects in memory will create memory pressure and trigger garbage collection during the runtime of our program. A consequence of automatic garbage collection is that memory management is now outside the control of the programmer and can trigger at times that are undesirable for the purpose of the program. Real-time applications, in particular, suffer from garbage collection because it makes the performance of the program hard to predict, an unacceptable trade-off when execution needs to be guaranteed to run within a small time frame.

In order to test our performance hypothesis, I am going to use a series of pro-

grams and run them multiple times under different conditions in order to measure different aspects of performance - typically, observing how memory usage and runtime varies depending on the optimisation we use.

Each benchmark will be compared to its control. The control will be the same program but running without any optimisations, this way, any deviation from the control can be attributed to our compiler change.

It is important to stress that those are *synthetic benchmarks* designed to showcase our improvements, and do not necessarily represent their effect in production software. This has often been the source of many misleading claims about performance and is the topic of endless discussion. The nofib[51] project (https://gitlab.haskell.org/ghc/nofib) makes the effort to distribute a set of programs more representative of real-world production software, especially for functional languages. In our case, synthetic benchmarks will be enough to tell if linear types provide any benefit worth pursuing.

6.4 Benchmarks

Now that we have stated the assumptions under which we are working and the results we are looking for, I am going to present 3 programs that will test our hypothesis.

Fibonacci

Our first benchmark is the traditional Fibonacci exercise. Three variants of this program will be used:

- The base version with no optimisation
- A modified version that incurs memory allocation
- The same modified version but with our optimisation applied

---

The first version is the one you would expect from a traditional implementation in a functional programming language:

```
tailRecFib : Nat -> Int
tailRecFib Z = 1
tailRecFib (S Z) = 1
tailRecFib (S (S k)) = rec 1 1 k
  where
    rec : Int -> Int -> Nat -> Int
    rec prev curr Z = prev + curr
    rec prev curr (S j) = rec curr (prev + curr) j
```

As you can see it does not perform any extraneous allocation since it only makes use of primitive values like `Int` which are not heap-allocated. If our optimisation works perfectly, we expect to reach the same performance signature as this implementation.

The second version allocates a new value for each call of the `update` function. We expect this version to perform worse than the previous one, both in memory and runtime because those objects are allocated on the heap (unlike ints), and allocating and reclaiming storage takes more time than mutating values.

```
data FibState : Type where
  MkFibState : (prev, curr :  Int) -> FibState

                            --   ┌ Allocation happens here
next : FibState -> FibState --   ▼
next (MkFibState prev curr) = MkFibState curr (prev + curr)

rec : FibState -> Nat -> Int
rec (MkFibState prev curr) Z = prev + curr
rec state (S j) = rec (next state) j

tailRecFib : Nat -> Int
tailRecFib Z = 1
tailRecFib (S Z) = 1
tailRecFib (S (S k)) = rec (MkFibState 1 1) k
```

The last version is almost the same as the previous one except our `update` function should now avoid allocating any memory, while this adds a function call compared to the first version we do expect this version to have a similar performance profile as the first one.

```
import Data.List
import Data.Nat

data FibState : Type where
  MkFibState : (prev, curr :  Int) -> FibState

-- Allocation should be removed when using the flag
%mutating
next : (1 _ : FibState) -> FibState
next (MkFibState prev curr) = MkFibState curr (prev + curr)

tailRecFib : Nat -> Int
tailRecFib Z = 1
tailRecFib (S Z) = 1
tailRecFib (S (S k)) = rec (MkFibState 1 1) k
  where
    rec : FibState -> Nat -> Int
    rec (MkFibState prev curr) Z = prev + curr
    rec state (S j) = rec (next state) j
```

For those three programs, we expect the first one to be the fastest, and the second one to be the slowest. Our third implementation should be identical in terms of performance as the first one.

We will run our benchmarks on the Chez backend, and we know that the Chez runtime is already really smart. With that in mind, we expect the absolute difference between the three results to be within tens of seconds of each other. However, since what we are removing is not running time per se, but *garbage collection uncertainty* we expect the variance (and therefore standard deviation) to be higher on the second benchmark than in the first or third.

Mapping Lists

We know that we can remove the need for intermediate data-structures when mapping them using linear operations[14], but can we reproduce this result here? This benchmark aims to simply map a list from `Int` to `Int`.

```
mapList : (1 _ : List Int) -> List Int
mapList [] = []
mapList (x :: xs) = x + 1 :: mapList xs

main : IO ()
main = printLn (length (mapList [0 .. 9999]))
```

Similar to the deforestation algorithm, we are going to see if we can improve the performance of linear functions across data structures like lists. When mapping across a list with a function of type `a -> a` that only mutates the input we can avoid allocating an entirely new list.

```
%mutating
mapList : (List Int) -> List Int
mapList [] = []
mapList (x :: xs) = x + 1 :: mapList xs

main : IO ()
main = printLn (length (mapList [0 .. 9999]))
```

A small detail to notice here is that the function is non-linear, indeed using the signature `mapList : (1 _ : List Int) -> List Int` will not compile because `n + 1` does not consume `n` linearly.

In a similar fashion as the Fibonacci benchmark, we are not removing running time, but uncertainty from having the garbage collector running. This means the second implementation should have a smaller standard deviation than the first one.

A SAT solver

SAT solvers themselves aren't necessarily considered "real-world" programs in the same sense that compilers or servers are. However, they have two benefits:

- You can make them arbitrarily slow to make the performance improvement very obvious by increasing the size of the problem to solve.
- They still represent a real-life case study where a program needs to be fast and where traditional functional programming has fallen short compared to imperative programs.

In this case, we are interested in solving the core loop of the state monad that threads through the entire program:

```
(>>=) : (1 _ : LState a) -> (1 f : ((_ : a) -> LState b))
     -> LState b
(>>=) (MkLState f) c = MkLState
    (\x => let (res # s') = f x in
           let (MkLState r2) = c res in
           r2 s')
```

Our optimisation is subsumed by the "newtype-optimisation" which erases any boxing around types with a single constructor. We can however safely inline this function since `(>>=)` will only be composed with itself, since it takes linear arguments the inlining when composed is safe. This time we expect the running time to be consistently smaller. Additionally, since we can make SAT problems arbitrarily complicated we can have a benchmark that measures in tens of seconds and record differences in runtime in the order of seconds rather than micro-seconds.

6.5 How to Run the Benchmarks

Results are only useful if they are reproducible. This section aims to inform the reader of the conditions in which the benchmarks have been run and which tools have been used in order to collect and analyse the results. This is with the goal to make those results reproducible by you, the reader, either exactly by re-using the tools, or by replicating the functionality of the tools presented.

Since the benchmarks have to be run many times and have to be run in an identical and reproducible setting I've written a program that will do just that: Idris-bench. Idris-bench will run all the idris2 programs in a folder using a given compiler version, a number of runs for each program and will measure how long each run took, and will aggregate the results in a CSV file.

Additionally, as mentioned in section 6.1, we are going to be very interested in the standard deviation of our results, this is why I have written another tool to analyse the results: Idris-stats. Idris-stats ingests the CSV file from Idris-bench and outputs statistical information about it in another CSV file.

Idris-bench

Idris-bench is our benchmarking program and can be found at https://github.com/andrevidela/idris-bench.

Idris-bench takes the following arguments:

- `-p | --path IDRIS_PATH` the path to the idris2 compiler we want to use to compile our tests. This is used to test the difference between different compiler versions. Typically, running the benchmarks with our optimised compiler and running the benchmarks without our optimisation can be done by calling the program with two different versions of the idris2 compiler.

- `-t | --testPath TEST_PATH` The path to the root folder containing the test files.
- `-o | --output FILE_OUTPUT` The location and name of the CSV file that will be written with our results.
  - Alternatively, `--stdout` can be given in order to print out the results on the standard output.
- `-c | --count` The number of times each file has to be benchmarked. This is to get multiple results and avoid lucky/unlucky variations.
- `--node` If the node backend should be used instead. If this flag is absent, the Chez backend will be used instead.

A hidden feature of it is the ability to use the `realRunTime` function in order to parse the result of the time spent executing a program. This requires recompiling the Idris-bench binary and is not documented. The reason for this is that it makes the assumption that the scheme backend has been modified to wrap programs into a call to `(time ...)`, which is not a feature officially supported by the compiler but which proves to be useful to output the time spent executing *after* the startup time from the Chez runtime.

Idris-stats

Once our results have been generated with Idris-bench, we are going to analyse them by computing the minimum time, maximum time, mean and variance of the collection of benchmark results. Idris-stats will take our CSV output and compute the data we need and output them as another CSV file. Again the code can be found here https://github.com/andrevidela/idris-bench/blob/master/script/idris-stats.idr.

The CSV file in input is expected to have the following format:

```
name of first benchmark, result1, result2, result3, etc
name of second benchmark, result1, result2, result3, etc
...
name of final benchmark, result1, result2, result3, etc
```

The first column is ignored for the purpose of our data analysis.

For each row we compute the minimum value, the maximum value, the mean, the standard deviation and the standard error to compare data sets with different numbers of samples.

Running Instructions

All the benchmarks were run on a laptop with the following specs:

- Intel core-i5 8257U (8th gen), 256KB L2 cache, 6MB L3 cache
- 16Gb of ram at 2133Mhz

While this computer has a base clock of 1.4Ghz, it features a boost clock of 3.9Ghz (a feature of modern CPUs called "turbo-boost") which is particularly useful for single-core application like ours. However, turbo-boost might introduce an uncontrollable level of variance in the results since it triggers based on a number of parameters that aren't all under control (like ambient temperature, other programs running, etc). Because of this I've disabled turboboost on this machine and run all benchmarks at a steady 1.4Ghz.

The benchmarks have been run using Idris-bench using the following options:

For 100 runs of the Fibonacci suite:

```
build/exec/benchmarks -d ../idris2-fib-benchmarks
                      -o results.csv
                      -p idris2dev
                      -c 99
```

The result files were then fed into Idris-stat:

```
build/exec/stats results.csv
```

Which outputs its analysis to stdout.

6.6 Results & Discussion

In this section, I will present the results obtained from the compiler optimisation. The methodology and the nature of the benchmarks are explained in the "Benchmarks & methodology" section.

Our first test suite runs the benchmark on our 3 Fibonacci variants and our list mapping. As a refresher, they are as follows:

- The first one is implemented traditionally, carrying at all times 2 Ints representing the last 2 Fibonacci numbers and computing the next one
- The second one boxes those Ints into a datatype that will be allocated every time it is changed

- The third one will make use of our optimisation and mutate the boxes' values instead of discarding the old one and allocating a new one.

The hypothesis is as follows: Chez is a very smart and efficient runtime and our example is small and simple. Because of this, we expect a small difference in runtime between those three versions. However, the memory pressure incurred in the second example will trigger the garbage collector to interfere with execution and introduce uncertainty in the runtime of the program. This should translate in our statistical model as a greater standard deviation in the results rather than a strictly smaller mean.

Fibonacci

Here are the results of running our benchmarks 100 times in a row:

	no allocation	with allocation	with optimisation
minimum	2.027443000s	2.063915000s	2.103887000s
maximum	2.161514000s	2.195077000s	2.401920000s
average	2.034254150s	2.072595070s	2.118569650s
std deviation	0.014956143s	0.014123120s	0.040342991s
std error	0.001495614s	0.001412312s	0.004034299s

As you can see the results are not consistent with our predictions, the version which aims to remove allocations has a higher average and a higher standard deviation. Additionally, the allocating version and the original implementation without allocation have *very similar* performance profiles. To ensure our results are not the product of random fluctuations, we are going to run the same benchmark 1000 times instead of 100.

	no allocation	with allocation	with optimisation
minimum	2.027936000s	2.066721000s	2.108056000s
maximum	2.300934000s	2.476587000s	2.535437000s
average	2.059218259s	2.109659476s	2.186972896s
std deviation	0.054314868s	0.066678467s	0.083581987s
std error	0.001752092s	0.002150918s	0.002696193s

Now the standard error has gone down for our optimised program compared to the experiment with 100 runs, which is good news. However, the average and the standard deviation of our optimisation is both *higher* than the other two, which contradicts our hypothesis.

There is, however, something we have not taken into account yet, and that is to subtract the startup time of the scheme runtime. Indeed, every program is

measured using the difference between the time it started and the time it ended. But this time also includes the time it takes to launch Scheme and then execute a program on it. Indeed, the following program:

```
main : IO ()
main = pure ()
```

Takes 0.3 seconds to run despite doing nothing. Therefore, we expect more accurate results with less standard deviation and an average of about 0.3 seconds shorter.

Fibonacci without startup time To remove the startup time, we are going to change the emitted bytecode to wrap our main function inside a time-measuring function `(time …)`[24]. Since the timer won't start until the program is ready to run the startup time will be eliminated. Running our empty program we get the following (expected) result:

```
> 0.000000000s elapsed cpu time
```

This time we will run our benchmarks 1000 times using the same command as before. Running our statistical analysis gives us those results:

	no allocation	with allocation	with optimisation
minimum	1.65627213s	1.696760896s	1.734708117s
maximum	1.88141296s	1.977075901s	2.152951106s
average	1.67685517s	1.744737712s	1.786299514s
std deviation	0.30236463s	0.030303518s	0.047403167s
std error	0.00975369s	0.000977532s	0.001529134s

The results do not differ significantly from the previous measurement, indicating that the startup time was not responsible for our unexpected result. Though we note that the average of the optimised version has gone down a bit.

Another possible explanation is that Scheme performs JIT compilation and correctly identifies the hot-loop in our unoptimised example but is unable to perform such optimisation with our mix of pure and mutating code in the non-mutating one.

[24]This change also has the benefit of reporting how many memory allocations and garbage collection cycles happen and it turns out that our optimised program allocates exactly as much memory as the non-allocating one.

Fibonacci without startup time, small loop In order to test the JIT hypothesis we are going to run the same test, *without* startup time but with a much smaller loop so that the results are measured in milliseconds rather than seconds. This should be enough to prevent the runtime from identifying the loop and performing its optimisation.

To reduce the running time from seconds to milliseconds, we simply change the loop count from `8*10^6` to `8*10^4` reducing it by two orders of magnitude reduces the running time accordingly.

	no allocation	with allocation	with optimisation
minimum	0.006385357s	0.007216185s	0.006543267s
maximum	0.007625528s	0.008861124s	0.010942671s
average	0.006624731s	0.007520532s	0.006867369s
std deviation	0.000154925s	0.000189282s	0.000230448s
srd error	0.000004997s	0.000006105s	0.000007433s

This data suggests that Scheme was not able to optimise the allocating version, and given two programs, one that allocates and one that does not, the second one is faster. Our hypothesis about variance has been contradicted throughout with the variance of this run being again higher than the other two.

Mapping a list

In this example, we are going to aggregate the result of our control and our experimental run of `map`; which increments a list of numbers and returns its size. We ran each program 100 times using `idris-bench`. As before we expect a negligible average speedup but a significant decrease in standard deviation.

	allocating map	mutating map
minimum	0.000873337s	0.000523816s
maximum	0.005067636s	0.001308711s
average	0.001146103s	0.000685042s
std deviation	0.000470263s	0.000140894s
std error	0.000047026s	0.000014089s

This time our performance improvements are more noticeable on average than before. Though it might not be significant since it runs in tens of microseconds.

SAT solver

For this benchmark, we didn't use `idris-bench` rather we compiled two versions of `isat` (https://git.sr.ht/~cypheon/idris-minisat), one with the inlining

optimisation, and one without. And ran each version 10 times on the same input (in appendix **??**) :

	Allocating SAT	Inlined SAT
minimum	3.51878s	3.47685s
maximum	3.96815s	3.87899s
average	3.84008s	3.71382s
std deviation	0.17506s	0.15493s
srd error	0.05835s	0.05164s

The results show an improvement of about 3.3% on average which is encouraging. This small improvement could be taken further by implementing *safe inlining by default* on every variable that is linearly bound since it does not hurt performance and hoping that the inlined function will be subject to further optimisations after being inlined.

Discussion

Overall the results show that linear types do affect performance, however, the results also show that those benefits are not strictly superior to other forms of optimisations such as JIT or automatic unboxing of newtypes (another feature that we have not discussed here, but which subsumes our mutation optimisation when the data constructor has only one case with one value in it).

Particularly in the Fibonacci example, despite our best efforts, we see that removing allocation through linear types impedes performance by making other forms of optimisation harder to detect.

Ideally, the best way to test our optimisation would be to write our own runtime which runs on *bare metal* or some approximation of it (WASM/LLVM) which would (probably) be even faster than Scheme. It would also give us more control over which optimisations play nicely together (for example with JIT) and which ones are redundant or even harmful to performance. But this exercise would pertain more to *beating* the Racket runtime rather than *working* with it.

From this I draw two conclusions:

- We need linearity to be easier to use so that it appears in more programs so that we can replicate the performance gains from mapping lists and inlining values.
- To show drastic improvement in runtime using linearity we need to drastically change the nature of the memory model.

Both those ideas have been explored in section 5 in which the idea of using different semirings allows for better ergonomics as well as mapping to a memory model that is directly manipulated through the type system.

One final note about backends; Idris2 has an alternative javascript backend, however, the javascript code generation is unable to translate our programs into plain loops free of recursion. Because of this, our benchmark exceeds the maximum stack size and the program aborts. When the stack size is increased the program segfaults. This could be solved with the use of trampolines in the runtime, instead of plain recursion.

7 Future Work

Work never ends and progress instils more progress. While a lot of ground has been covered, exploring the forest of all knowledge not only yields results, it uncovers new paths to explore. This section outlines additional work that could be done in order to ease the use of linear types as well as make use of them for additional functionality.

7.1 Enlarging the Scope

The optimisation of section 6 only looks at the *immediate* scope of let bindings. Technically speaking, nothing is preventing our optimisation from working with a more indirect scoping mechanism. Indeed, the following should trigger our optimisation:

```
defaultVal : MyData
defaultVal = MkDefault 3

update : (1 rec : MyData) -> MyData
update (MkDefault n) = MkDefault (S n)
update (MkOther n) = MkOther (S (S n))

operate : MyData
operate = let 1 def = defaultVal
              1 newVal = update def in
              update newVal
```

But it will not because `defaultVal` is not a constructor, it's a function call that itself returns a constructor.

One implementation strategy would be to wait for the compiler to inline those definitions and then run our optimiser without further changes.

Another optimisation would be to follow references to see if they result in plain data constructors and replace the entire call chain by the constructor itself, and then run our optimisation.

While both those strategies are valid they incur a cost in terms of complexity and compile-time that may not be worth the effort in terms of performance results. They could be hidden behind a -O3 flag, but that kind of effort is probably better spent in making the ergonomics of linear types more streamlined to help those optimisations be more commonplace.

7.2 Making Linearity Easier to Use

Multiple barriers make linearity harder to use than one might expect. They roughly end up in two buckets:

- I want to use linearity but I cannot
- I have a linear variable and that's bothersome

Not Linear Enough

The first one appears when the programmer tries to make thoughtful usage of linear and erased annotation but finds that other parts of existing libraries do not support linearity. The following program:

```
operate : (1 n : Nat) -> (1 m : Nat) -> Int
operate n m = n + m
```

Gives the error:

```
> Trying to use linear name n in non-linear context
```

Because the `+` interface is defined as:

```
interface Num ty where
    (+) : ty -> ty -> ty
```

Despite addition on `Nat` being defined linearly:

```
plus : (1 n : Nat) -> (1 m : Nat) -> Nat
plus Z m = m
plus (S n) m = S (plus n m)
```

A similar problem occurs with interfaces

```
interface Monad (Type -> Type) where
    ...

data MyData : (0 ty : Type) -> Type where
    ...

instance Monad MyData where
    ...
```

```
> Expected Type -> Type
```

```
> got (0 ty : Type) -> Type
```

One way to solve those issues would be to have *linearity polymorphism* and be able to abstract over linearity annotations. For example, the map function could be written as:

```
map : forall l . ((l v : a) -> b) -> (l ls : List a) -> List b
map f [] = []
map f (x :: xs) = f x :: map f xs
```

That is, the list is linearly consumed iff the higher-order function is linear. What it means for our interface problem is that it could be rewritten as:

```
interface forall l . Functor (m : (l _ : Type) -> Type) where
    ...
interface forall l . Functor {l} m ⇒ Applicative {l} m where
    ...
interface forall l . Applicative {l} m ⇒ Monad {l} m where
    ...
```

A similar solution could be provided for `Num`

```
interface Num ty where
    (+) : forall l. (l n : ty) -> (l m : ty) -> ty
```

So that it can be used with both linear and non-linear variables.

Too Linear Now

We've already mentioned before how beneficial it would be for our optimisation strategy to be *100%* linear in every aspect. We also mentioned how this is a problem to implement basic functionality like `drop` and `copy`, but those are artificial examples, rarely does a programmer need to call `copy` or `drop` in industrial applications. In the following example, I will show how linear types hold back programming and how QTT can fix it.

A common scenario when debugging effectful code is sprinkling around log statements hoping that running the program will give insight into how it's running.

```
do datas <- getData arg1 arg2
   Just success <- trySomething datas (options.memoized)
     | _ ⇒ pure $ returnError "couldn't make it work"
   case !(check_timestamp success) of
      Safe t v ⇒ functionCall t v
      Unsafe t ⇒ trySomethingElse t
      UnSynchronized v ⇒ functionCall 0 v
      Invalid ⇒ pure $ returnError "failed to check"
```

Assuming everything is linear, there is no possible way to add a new print statement without getting a linearity error:

```
do datas <- getData arg1 arg2
   Just success <- trySomething datas (options.memoized)
     | _ ⇒ pure $ returnError "couldn't make it work"
   putStrLn $ show success
   --                 ▲
   --                 └── One use here
   --   And one use there ──┐
   --                       ▼
   case !(check_timestamp success) of
      Safe t v ⇒ functionCall t v
      Unsafe t ⇒ trySomethingElse t
      UnSynchronized v ⇒ functionCall 0 v
      Invalid ⇒ pure $ returnError "failed to check"
```

We've already mentioned how borrowing and quantitative types would help us implement reference counting. In this case, they have a more user-facing property, allowing programming updates without using unrestricted variables, which loses us the benefits of quantitative types. Using *precise usage* as our quantity would result in the following program:

```
do datas <- getData arg1 arg2
   2 (Just success) <- trySomething datas (options.memoized)
     | _ ⇒ pure $ returnError "couldn't make it work"
   putStrLn $ show success
   case !(check_timestamp success) of
      …
```

This would allow `success` to be bound with linearity `2` and therefore used twice. We can also allow *borrowing* of read-only variables to fix the same problem:

```
do datas <- getData arg1 arg2
   Just success <- trySomething datas (options.memoized)
     | _ ⇒ pure $ returnError "couldn't make it work"
   putStrLn $ show &success -- this doesn't count as a use
   case !(check_timestamp success) of
```

We've seen that read-only variables have been a topic for linearly typed systems with *Linear Types can change the world*[16], where linear read-only variables are allowed within some restriction on their scope. It would be interesting to implement a similar set of rules, adapted for QTT, that allows multiple uses of linear types as long as they are read-only and our `%mutating` implementation to differentiate between read-only and mutating functions.

8 Conclusion

I have learned a lot during this Master's project. Not only about linear types, quantitative types, graded modal types, containers, categories, optimisation techniques, benchmarking, program safety, compiler design and implementation, etc.

But I also learned about myself and how I best approach research topics. Research is not an activity to carry out alone, it is not a mindless job, and it is not limited to the ivory tower that is higher education. I learned that I do my best research work when I explore the paths that open in the forest of knowledge after clearing one of them. I would never have understood graded modalities if I wasn't distracted by semirings. I would never have understood semirings if I wasn't frustrated porting some code to Idris2 and noticing the limitations of `0`, `1` and `ω`. I would have never started porting code to Idris2 if I wasn't looking for an escape from writing tests for my performance improvements.

While this behaviour is dangerous, in that it can result in nothing of value produced, it also taught me the importance of having a strong and reliable support group. Friends, both in research and otherwise, focus the mind, which results in more focused work. Finally, research is not something one can *tune out of*, thoughts linger, ideas sprawl out of control and new connections are drawn during every event of daily life. I have discovered that sharing my research through teaching has been an extremely effective means to contextualise my work and to put boundaries between personal and research life.

Quantitative types, just like me, still have a long way to go. The work displayed here is only a minuscule sliver of what can be done, and needs to be done, for them to become commercially significant. In line with the two main topics of this thesis, I think the areas quantitative types need to improve are ergonomics and performance. They are two pain points that functional programming has failed to fix, while other programming languages have been successful basing their entire value proposition upon them (Rust, Javascript, Python, etc).

Ergonomics have made huge strides forward with interactive development environments as we find in Idris, Agda or Coq. But their gimmicks cannot sustain the behemoth of features that commercially successful programming languages showcase. Linear types won't turn the balance around, but they will help in some important aspects: ensuring APIs contracts and protocols are respected, showing this information in the type, and therefore in the documentation, helping the compiler generate more useful error messages, and helping guide new

users in navigating complex programs by being explicit about usage rules that were left unwritten before.

Until the recent development in resource tracking technologies, it wasn't clear how the concept of performance could be approached and studied in functional programming. We did not have a way to get a hold of it. But just like we did not have a way to get a hold on side-effect until monads, I strongly suspect quantitative types will allow us to turn performance from an abstract concept into a concrete term in a programming language, just like `IO` became a concrete term in functional programming languages.

I hope you enjoyed reading this report on linear and quantitative types as much as I did writing it. I wish I explored the paths of knowledge that I discovered in more depth. Yet, I am satisfied with the result of this year of exploration. I come out of it with a strong intuition about quantitative types, how to use them, and how to improve them. I strongly believe this thesis, as the first detailed exploration of quantitative types used in practice with dependent types, will be the start of a series of great discoveries.

9 Appendices: Glossary and definitions

While the initial list of fancy words in the introduction is nice it suffers from being superficial and therefore incomplete. These are more detailed definitions using examples and imagery.

9.1 Affine types

Affine types describe values that can be used at most 0 times, at most 1 times or at most infinitely many times (aka no restrictions)

9.2 Co-monad / Comonad

A mathematical structure that allows us to encapsulate *access to a context*. For example, `List` is a Comonad because it allows us to work in a context where the value we manipulate is one out of many available to us, those other values available to us are the other values of the list.

9.3 Indexed type

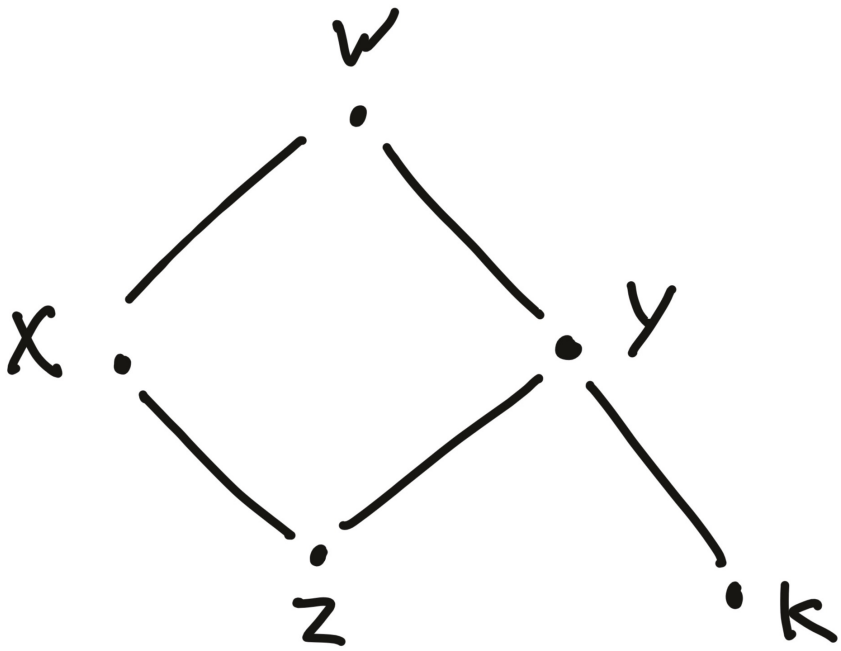
A *type parameter* that changes with the values that inhabit the type. For example, `["a", "b", "c"] : Vect 3 String` has index `3` and a type parameter `String` because it has `3` elements and the elements are Strings. If the value was `[`"a", "b"]` then the type would become `Vect 2 String`, the index would change from `3` to `2`, but the type parameter would stay as `String`.

9.4 Implicit argument

An implicit argument is one that does not need to be filled out at call-site because the compiler will fill in the argument for you. In the type signature: `length : {n : Nat} -> Vect n a -> Nat` the argument `n` is implicit.

9.5 Lattice

A mathematical structure that relates values to one another in a way that doesn't allow arbitrary comparison between two arbitrary values. Here is a pretty picture of one:



As you can see we can't really tell what's going on between X and Y, they aren't related directly, but we can tell that they are both smaller than W and greater than Z

9.6 Linear types

Linear types describe values that can be used exactly 0 times, exactly 1 time, or have no restrictions put on them

9.7 Linearity / Multiplicity

Used interchangeably most of the time. They refer to the number of times a variable is expected to be used. Technically, there is a slight difference between those three terms:

- Linearity refers to variables that can be used exactly once
- Multiplicity refers to the number of times a variable can be used. Not only exactly once, and not only just numbers.

9.8 Monad

A mathematical structure that allows us to encapsulate *change in a context*. For example, `Maybe` is a Monad because it creates a context in which the values we are manipulating might be absent. Formally, a monad is a triple between an indexed set X_i a `bind` operation $X_a \to (a \to X_b) \to X_b$ and a `pure` operation $1 \to X_a$

9.9 Pattern matching

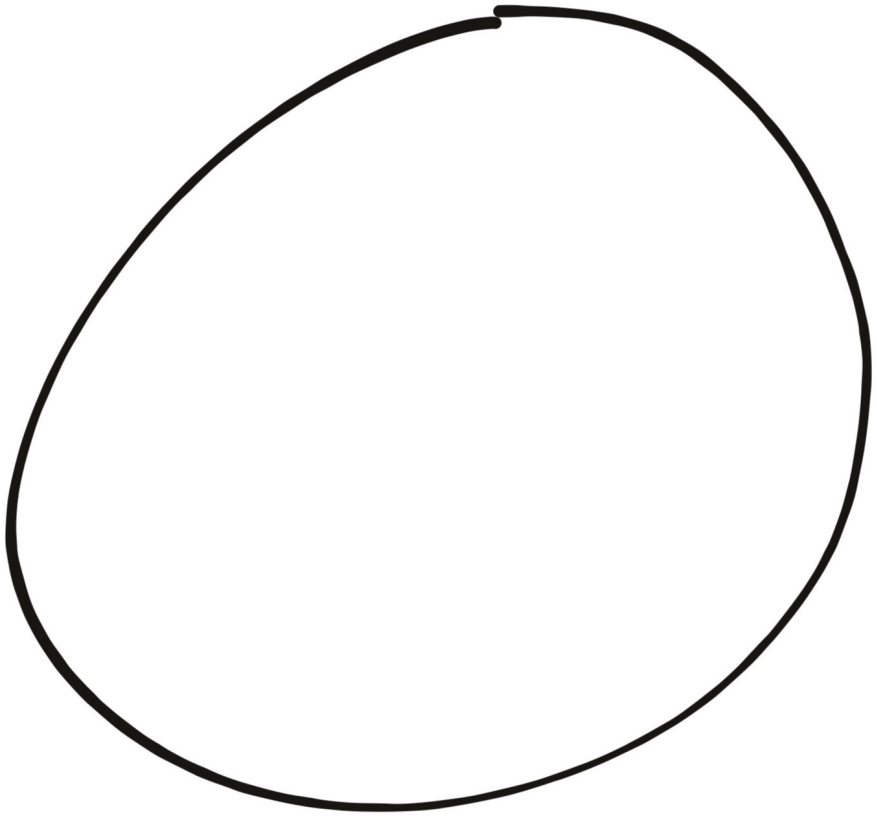
Pattern matching consists in destructuring a value into its constituent parts in order to access them or understand what kind of value we are dealing with.

9.10 Semiring

A mathematical structure that requires its values to be combined with `+` and `*` in the ways you expect from natural numbers. That is `0 + n = n`, `1 * n = n`, `0 * n = 0` , `a * (b + c) = (a * b) + (a * c)` , and both operators are symetric.

9.11 Syntax

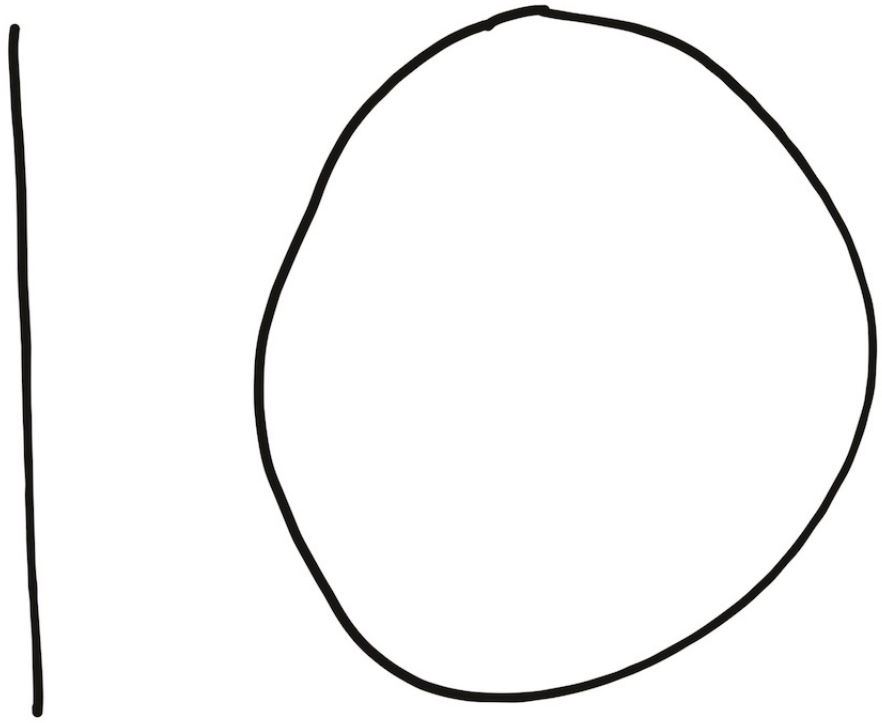
The structure of some piece of information, usually in the form of *text*. Syntax itself does not convey any meaning. Imagine this piece of data:



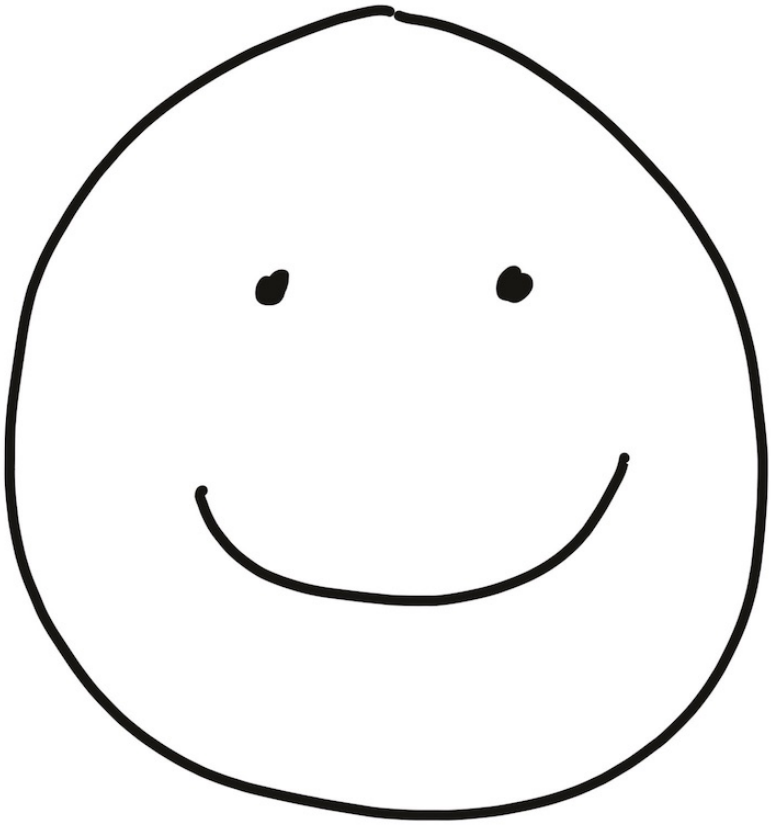
We can define syntactic rules that allow us to express this circle, here is one: all shapes that you can draw without lifting your pen or making angles. From this definition lots of values are allowed, including `|`, `-`, `0` but not `+` because there is a 90^o angle between two bars. Is it supposed to be the letter "O", the number "0" the outline of a planet? The back of the head of a stick figure from the back?

9.12 Semantics

The meaning associated to a piece of data, most often related to syntax. From the *syntax* definition if we have:



We can deduce that the circle means “the second digit of the number 10” which is the number “0”. We were able to infer semantics from context. Similarly, in this picture:



We can deduce that the meaning of the circle was to represent the head of a stick figure from the front.

9.13 Type system

Set of rules the types in a program have to follow in order to be accepted by the compiler. The goal of the type-system is to catch some classes of errors while helping the programmer reach their goal more easily.

9.14 Type theory

Type theory is the abstract study of type systems, most often in the context of pure, mathematical logic. When we say "a Type Theory", we mean a specific set of logical rules that can be implemented into a *Type System*.

References

[1] R. Atkey, "Syntax and semantics of quantitative type theory," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, p. 56–65, Association for Computing Machinery, Jul 2018.

[2] C. McBride, *I Got Plenty o' Nuttin'*, vol. 9600 of *Lecture Notes in Computer Science*, p. 207–233. Springer International Publishing, 2016.

[3] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, p. 552–593, Sep 2013.

[4] C. Mcbride and J. Mckinna, "The view from the left," *Journal of Functional Programming*, vol. 14, p. 69–111, Jan 2004.

[5] S. Lindley and C. McBride, "Hasochism: the pleasure and pain of dependently typed haskell programming," in *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, Haskell '13, p. 81–92, Association for Computing Machinery, Sep 2013.

[6] E. Brady, *Type-driven development with Idris*. Manning, 2017.

[7] U. Norell, "Interactive programming with dependent types," in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming - ICFP '13*, p. 1, ACM Press, 2013.

[8] J.-Y. Girard, "Theoretical computer science," in *Linear logic*, vol. 50, pp. 1–102, 1987.

[9] J.-Y. Girard, A. Scedrov, and P. J. Scott, "Bounded linear logic: a modular approach to polynomial-time computability," *Theoretical computer science*, vol. 97, no. 1, pp. 1–66, 1992.

[10] D. Orchard, V.-B. Liepelt, and H. Eades III, "Quantitative program reasoning with graded modal types," *Proceedings of the ACM on Programming Languages*, vol. 3, p. 1–30, Jul 2019.

[11] M. Gaboardi, S.-y. Katsumata, D. Orchard, F. Breuvart, and T. Uustalu, "Combining effects and coeffects via grading," *ACM SIGPLAN Notices*, vol. 51, p. 476–489, Dec 2016.

[12] A. L. Smirnov, "Graded monads and rings of polynomials," *Journal of Mathematical Sciences*, vol. 151, p. 3032–3051, Jun 2008.

[13] D. Orchard, T. Petricek, and A. Mycroft, "The semantic marriage of monads and effects," *arXiv:1401.5391 [cs]*, Jan 2014. arXiv: 1401.5391.

[14] P. Wadler, "Deforestation: Transforming programs to eliminate trees," in *European Symposium on Programming*, pp. 344–358, Springer, 1988.

[15] D. N. Turner, P. Wadler, and C. Mossin, "Once upon a type," in *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pp. 1–11, 1995.

[16] P. Wadler, "Linear types can change the world!," 1990.

[17] P. Wadler, "Is there a use for linear logic?," May 1991.

[18] J. Chirimar, C. A. Gunter, and J. G. Riecke, "Reference counting as a computational interpretation of linear logic," *Journal of Functional Programming*, vol. 6, p. 195–244, Mar 1996.

[19] J. A. Tov and R. Pucella, "Practical affine types," Jan 2011.

[20] K. Kokke, J. Morris, and P. Wadler, "Towards races in linear logic," in *Coordination Models and Languages* (H. Riis Nielson and E. Tuosto, eds.), vol. 11533 of *Lecture Notes in Computer Science*, pp. 37–53, Springer, Cham, Aug. 2019. 21st IFIP WG 6.1 International Conference, COORDINATION 2019, International Conference on Coordination Models and Languages : Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17â 21, 2019, COORDINATION 2019 ; Conference date: 17-06-2019 Through 21-06-2019.

[21] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack, "Linear haskell: practical linearity in a higher-order polymorphic language," Dec 2017.

[22] K. Mazurak, J. Zhao, and S. Zdancewic, "Lightweight linear types in system f°," in *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation - TLDI '10*, p. 77, ACM Press, 2010.

[23] D. Orchard, "Should i use a monad or a comonad," 2012.

[24] T. Petricek, D. Orchard, and A. Mycroft, "Coeffects: a calculus of context-dependent computation," *ACM SIGPLAN Notices*, vol. 49, p. 123–135, Nov 2014.

[25] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic, *A Core Quantitative Coeffect Calculus*, vol. 8410 of *Lecture Notes in Computer Science*, p. 351–370. Springer Berlin Heidelberg, 2014.

[26] N. R. Krishnaswami, P. Pradic, and N. Benton, "Integrating linear and dependent types," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15*, p. 17–30, ACM Press, 2015.

[27] P. Martin-Löf, *An intuitionistic theory of types*.

[28] E. Brady, C. McBride, and J. McKinna, *Inductive Families Need Not Store Their Indices*, vol. 3085 of *Lecture Notes in Computer Science*, p. 115–129. Springer Berlin Heidelberg, 2004.

[29] S.-y. Katsumata, "Parametric effect monads and semantics of effect systems," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '14*, p. 633–645, ACM Press, 2014.

[30] A. Vezzosi, A. Mörtberg, and A. Abel, "Cubical agda: a dependently typed programming language with univalence and higher inductive types," Jul 2019.

[31] U. Norell, *Towards a practical programming language based on dependent type theory*. Doktorsavhandlingar vid Chalmers Tekniska Högskola, Chalmers Univ. of Technology, 2007.

[32] T. Altenkirch and C. Mcbride, "Towards observational type theory," in *In preparation*, 2006.

[33] F. Genovese, A. Gryzlov, J. Herold, A. Knispel, M. Perone, E. Post, and A. Videla, "idris-ct: A library to do category theory in idris," *Electronic Proceedings in Theoretical Computer Science*, vol. 323, p. 246–254, Sep 2020. arXiv: 1912.06191.

[34] S. Ullrich and L. de Moura, "Counting immutable beans: Reference counting optimized for purely functional programming," *arXiv:1908.05647 [cs]*, Mar 2020. arXiv: 1908.05647.

[35] R. Atkey, "Parameterised notions of computation," *Journal of Functional Programming*, vol. 19, p. 335–376, Jul 2009.

[36] E. Brady, "State machines all the way down," ML 2017, Sep 2017.

[37] D. O. Benjamin Moon, Harley D. Eades III, "Graded modal dependent type theory," TyDe 2020, Aug 2020.

[38] K. Matsuda and M. Wang, "Sparcl: a language for partially-invertible computation," *Proceedings of the ACM on Programming Languages*, vol. 4, p. 1–31, Aug 2020.

[39] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce, "Linear dependent types for differential privacy," *ACM SIGPLAN Notices*, vol. 48, p. 357–370, Jan 2013.

[40] P. Wadler, "Propositions as sessions," 2012.

[41] P. Wadler, "There is no substitute for linear logic," 2003.

[42] S. Fowler, S. Lindley, and P. Wadler, "Mixing metaphors: Actors as channels and channels as actors," 2017.

[43] F. Genovese, "The essence of petri net gluings," *arXiv:1909.03518 [cs, math]*, Sep 2019. arXiv: 1909.03518.

[44] P. Sobocinski, P. W. Wilson, and F. Zanasi, "Cartographer: A tool for string diagrammatic reasoning (tool paper)," p. 7 pages, 2019.

[45] J. Chapman, P.- . Dagand, C. McBride, and P. Morris, "The gentle art of levitation," Sep 2010.

[46] M. Abbott, T. Altenkirch, and N. Ghani, *Categories of Containers*, vol. 2620 of *Lecture Notes in Computer Science*, p. 23–38. Springer Berlin Heidelberg, 2003.

[47] M. Abbott, T. Altenkirch, C. McBride, and N. Ghani, " for data: Differentiating data structures," *IOS Press*, vol. 65, pp. 1–28, 2005.

[48] T. Altenkirch, N. Ghani, P. Hancock, C. Mcbride, and P. Morris, "Indexed containers," *Journal of Functional Programming*, vol. 25, p. e5, 2015.

[49] Statebox, *Typedefs*.

[50] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology - HILT '14*, p. 103–104, ACM Press, 2014.

[51] W. Partain, *The nofib Benchmark Suite of Haskell Programs*, p. 195–202. Workshops in Computing, Springer London, 1993.