Graded Modalities as Linear Types

ASHER FROST

We present an approach that models graded modal bindings using only linear ones. Inspired by Granule, we present an approach in Idris 2 to model limited grades using a construction, called "mu" (M). We present the construction of M, related ones, operations on them, and some useful properties

Todo list

1 Introduction

Update everything

One of the more interesting developments in Programming Language Theory is Quantitative Type Theory, or QTT. Based off Girard's linear logic, it forms the basis of the core syntax of Idris 2's core language, and also as a starting point for that of Linear Haskell [5, 3, 8, 1]. It has many theoretical applications, including creating a more concrete interpretation of the concept of a "real world", constraining memory usage, and allowing for safer foreign interfaces Need citation. Apart from just the theoretical insert, QTT has the potential to serve as an underlying logic for languages like Rust, where reasoning about resources takes the forefront.

However, new developments in Graded Modal Type Theory (GrTT), in particular with Granule, serve to create a finer grained[1] notion of usage than QTT. In GrTT, any natural number may serve as a usage, and in some cases even things that are first glance not natural numbers. GrTT is part of a larger trend of "types with algebras" being used to create inferable, simple, and intuitive systems for models of various things. Of these, some of the more notable ones are Koka, Effekt, and Flix, all of which seek to model effects with algebras [10, 4, 9].

[1] Hence the name

Author's Contact Information: Asher Frost.

```
data QNat : Type where
    Zero : QNat
    Succ : (1 k : QNat) -> QNat
```

Listing 1. Definition of linear natural numbers

2 Background

2.1 Copying and Dropping

Utlitmatly, one of the notable facts about a language like Rust with single use bindings is the notion of cloning, or copying, a value. Given that Rust's ownership system can be (partially) modeled by a linear type system, Need citation, it makes sense then that Idris' linear library has two interfaces, `Duplicable` and `Discardable`, which model duplication and droping of linear resources, respectivly.

However, a choice was made not use `Duplicable`, and its associated `Copies`, for a couple of reasons:

- It is hard to use in practice
- It relies heavily on `Copies`, which is quite similar to the main construction in this paper, mu types

For these reasons, we define a new interface, `Copy`, which has two methods, $\mathsf{copy} : ((x : a) \rightarrow_1 (y : a) \rightarrow_1 (pxy$ This essientally "uses a value twice" in an arbitrary (potentially dependent) function, and an erased proof that $\mathsf{copy} f z =_? f z z$.

We also redefine a `Drop` interface that is just `Consumable` with a different name, though this is more a style choice than anything else.

2.2 Linear Functions on Numbers

While Idris' does have a linear library, there are a couple problems with the support for linear bindings:

- These are not as well developed as their unrestricted counterparts
- They can't be converted to their unrestricted counterparts

One of the best examples of this is the natural numbers. Clearly, for any binding of 2, we expect that to be exactly one binding of 1 inside a succesor function. However, this is not the case, rather, Idris by default has it so that data constructor arguements are unrestricted by default. This is very important for a case like the natural numbers, where this is taken to the extreme, it is almost always impossible to talk about the `Nat` datatype in a useful way with linear bindings.

Natural Numbers. The solution to this, then, is to define the *linear* natural numbers. While the linear library also defines `LNat`, we also don't use that, because of the fact that, again, it relies on the `Copies` construction, in addition to already not having that large of an implementation to begin with. Granted, our version is almost exactly the same, and is defined as follows:

The rest of the operations use a model as close to the simple inductive definitions as possible, using `copy` instead of the more complex `duplicate`.

Finish this

Conatural Numbers.

```
record Exists (t : Type) (p : (t -> Type)) where
        constructor Evidence
        0 fst' : t
        1 snd' : p fst'
```

Listing 2. Definition of linear existentials

```
data Mu : (n : QNat) -> (t : Type) -> (w : t) -> Type where
        MZ :
                Mu 0 t w
        MS :
                (1 w : t) ->
                (1 xs : Mu n t w) ->
                Mu (Succ n) t w
```

Listing 3. The definition of M in Idris

2.3 Existential Types

Existential types, regarding dependent types, usually refers to dependent pair (Σ) types [13]. In this context, we see the first element of the pair as "evidence" for the type of the second element of the pair. In Idris, this is formalized (in Idris's *base*) by stating that the first argument is runtime erased, similarly to which universal quantification is a function that is erased.

However, for our usage here this is not suitable, as we are dealing with principally linear values, and `Exists` is unrestricted. Fourtanetly, however, the change from unrestricted to linear existentials is quite trivial, the construction of which may be found at **??**

Not much is notable about this, we define the operations as is usual, except every time a function would be unrestricted over the second value it is instead linear. We also have a operator, #?, which serves as sugar for this, using Idris's `typebind`, mechanism.

3 Mu types

The core construction here is M, or, in Idris, `Mu`, type. This is made to model a "source" of a given value. It is indexed by a natural number, a type, and an erased value of that type. The definitions of this in Idris are given at **??**.

Definition 3.1. M is a polymorphic type with signature $M : (n : \mathbb{N}) \to (t : *) \to (w : t) \to *$

In addition, M has two constructors

Definition 3.2. There are two constructors of M, $\diamond$ `MZ`, and $\odot$, `MS`, Which have the signatures $\diamond : M\ n\ t\ w$ and $\odot : (w : t) \to_1 M\ n\ t\ w \to_1$

Firstly, it should be noted that the names were chosen due to the fact that the indices of them are related. That is, $\diamond$ or "mu zero" will always be indexed by 0, and $\odot$, "mu successor", is always indexed by the successor of whatever is given in.

Remark 3.3. $M\ 0\ t\ w$ can only be constructed by $\diamond$.

Intuitively, M represents n copies of t, all with the value w, very much inspired by the paper "How to Take the Inverse of a Type". For instance, if we want to construct $x : M\ 2$ `String` `"value"`, we can only construct this through $\odot$, an we know, but nothing but the first argument of the value, that we must take as a initial argument `value`, and as a second value of the form $M\ 1\ \ String$`"value"`, which we then repeat one more time and get another `"value"` an finally we match $M\ 0$ `String` `"value"` and get that we must have $\diamond$.

We can then say that we know that the only constructors of $M\ 2$ `String` `"String"` is `"value"`$\odot$`"value"`$\odot\diamond$. Note the similarity between the natural number and the constructors. Just as we get 2 by applying `S`twice to `Z`, we get $M\ 2$ `String` `"value"` by applying $\odot$ twice to $\diamond$. This relationship between M types and numbers is far more extensive, as we will cover later.

The windex or “witness” is the value being copied. Notably, if we remove the `w` index, we would have $\diamond : Mnt$, and $\odot : t \rightarrow_1 Mnt \rightarrow_1 M(Sn)t$, which is simply `LVect`. The reason that this is undesirable is that we don't want this as it allows for M to have heterogeneous elements. However, if we are talking about “copies” of something, we know that should all be the exact same.

There are two very basic functions that bear mention with respect to M. The first of these is `witness`, which is of the form `witness` $:_0\ \forall(w : t) \rightarrow_0 M\ n\ t\ w\ \rightarrow\ t$. Note that this is an *erased* function. Its implementation is quite simple, just being `witness`$\{w\}\_ := w$, which we can create, as erased functions can return erased values Need citation. In addition, we also have `drop` $: M\ 0\ t\ w \rightarrow \top$, which allows us to drop a value. Its signature is simple `drop`$\diamond := ()$.

Finally, we have `once`, which takes a value of form $M1$ and extracts it into the value itself. It has a signature `once` $: M\ 1\ t\ w \rightarrow_1 t$, where we have `once`$(x\odot\diamond) := x$

3.1 Uniqueness

While `w`serves a great purpose in the interpretation of M it perhaps serves an even greater purpose in terms of values of M. We can, using w, prove that there exists exactly one inhabitant of the type $M\ n\ t\ w$, so long as that type is well formed.

Segway into proof

Lemma 3.4. *$x : M\ n\ t\ w$ only if the concrete value of x contains n applications of $\odot$.*

Proof. Induct on n, the first case, $M\ 0\ t\ w$, contains zero applications of $\odot$, as it is $\diamond$. The second one splits has $x : M\ (Sm')\ t\ w$, and we can destruct on the only possible constructor of M for S, $\odot$, and get $y :_1 t$ and $z :_1 Mntw$ where $x =_? (y\odot z)$. We know by the induction hypothesis that z contains exactly n' uses of $\odot$, and we therefore know that the constructor occurs one more times than that, or Sn' □

This codifies the relationship between M types and natural numbers. Next we prove that we can establish equality between any two elements of a given M instance. This is equivalent to the statement “there exists at most one M” Need citation.

Lemma 3.5. *If both x and y are of type $M\ n\ t\ w$, then $x =_? y$.*

Proof. Induct on n.

- The first case, where x and y are both $M\ 0\ t\ w$, is trivial because, as per **??** they both must be $\diamond$
- The inductive case, where, from the fact that for any a and b (both of $M\ n'\ tw$) we have $a =_? b$, we prove that we have, for x and y of $M\ (Sn')\ t\ w$ that $x =_? y$. We note that we can destruct both of these, with $x_1 : M\ n'\ tw$ and $y_1 : M\ n'\ tw$, into $x = x_0\odot x_1$ and $y = y_0\odot y_1$, where we note that both x_0 and y_0 must be equal to w, and then we just have the induction hypothesis, $x_1 =_? y_1$

We thereby can simply this to the fact that $M\ n'\ tw$ has the above property, which is the induction hypothesis. □

```
0 unique :
        {n : Nat} -> {t : Type} -> {w : t} ->
        {a : Mu n t w} -> {b : Mu n t w} ->
        (a === b)
unique {n=Z} {w=w} {a=MZ,b=MZ} = Refl
unique {n=(S n')} {w=w} {a=MS w xs, b=MS w ys} = rewrite__impl
        (\zs => MS w xs === MS w zs)
        (unique {a=ys} {b=xs})
        Refl
```

Listing 4. Proof of ?? in Idris

Finish proof

However, we can make an even more specific statement, given the fact we know that w is an inhabitant of t.

Theorem 3.6 (Uniqueness). *If $M\ n\ t\ w$ is well-formed, then it must have* exactly *one inhabitant.*

Proof. We know by **??** that there is *at most* one inhabitant of $M\ n\ t\ w$. We then induct on n to show that there must also exist *at least* one inhabitant of the type.

- The first case is that $M\ 0\ t\ w$ is always constructible, which is trivial, as this is just $\diamond$.
- The second case, that $M\ n\ '\ tw$ provides $M\ (Sn')\ t\ w$ being constructible is also trivial, namely, if we have the construction on $M\ n\ '\ tw$ as x, we *know* that the provided value must be w.

Given that we can prove that there must be at least and at most one inhabitant, we can prove that there is exactly one inhabitant. □

This is very important for proofs on M. It corresponds to the fact that there is only one way to copy something, to provide another value that is the exact same as the first.

3.2 Graded Modalities With Mu

A claim was made earlier that M types can be used to model graded modalities within QTT. The way it does this, however, is not by directly equating M types with [] types [12]. It instead does this similarly to how others have embedded QTT in Agda [7, 6].

Namely, rather than viewing $M\ n\ t\ w$ as the *type* $[t]_n$, we instead view it as the *judgment* $[w] : [t]_r$. Fortunately, due to the fact that this is Idris 2, we don't need a separate $\Vdash$, as M is a type, which, like any other, can be bound linearly, so, the equivalent to the GrTT statement $\Gamma \vdash [x] : [a]_r$ is $\Gamma \vdash \phi : M\ r\ a\ x$, which can be manipulated like any other type.

This is incredibly powerful. Not only can we reason about graded modalities in Idris, we can reason about them in the language itself, rather than as part of the syntax, which allows us to employ regular proofs on them. This is very apparent in the way constructions are devised. For instance, while Granule requires separate rules for dereliction, we do not, and per as a matter of fact we just define it as `once`; a similar relationship exists between weakening and `drop`where the exact relationship is shown in **??**.

Remark 3.7. We assume that $Mntw$ is equivalent to $[w] : [t]_n$.

Unfortunately, there is no way to prove this in either language, as M can't be constructed in Granule, and graded modal types don't exist in Idris 2.

$$\frac{\Gamma \vdash \alpha}{\Gamma, [w] : [t]_0 \vdash \alpha}\ \text{Weak}$$

```
weak : (ctx -@ a) -@ ((LPair ctx (Mu 0 t w)) -@ a)
weak f (x # MZ) = f x
```

Fig. 1. The meta-logical drop rule and its Idris 2 equivalent

$$\diamond \otimes x \qquad := x \tag{1}$$

$$Z + x \qquad := x \tag{2}$$

$$(a \odot b) \otimes \qquad x := a \odot (b \qquad \otimes x) \tag{3}$$

$$(Sn) + \qquad x := S(n \qquad +x) \tag{4}$$

3.3 Operations on Mu

There are a number of operations that are very important on M. The first of these that we will discuss is combination. We define this as $\otimes : \forall (m : \mathbb{N}) \to_0 \forall n \to_0 \mathbb{N} M\ m\ t\ w \to_1 M\ n\ t\ w \to_1 M\ (m + n)\ t\ w$ Need code and we define it inductively as $\odot \otimes x := x$, and $(a \odot b) \otimes x := a \odot (b \otimes x)$. Note the similarity between the natural number indices and the values, where $0 + x := x$ and $(Sn) + x := S(n + x)$

Fix the alignment of these

In addition, using the assumption of ??, we can liken the function $\otimes$ to context concatenation [12]. However, unlike Granule, we define this in the language itself. Per as a matter of fact, it isn't actually possible to construct $\otimes$in Granule, as it would require a way to reason about type level equality, which isn't possible.

Lemma 3.8. *$\otimes$ is commutative*[2]*, that is, $x \otimes y =_? y \otimes x$.*

Proof. These types are the same, and by ??, they are equal. □

Given that we can "add" (Or, as we will see later, multiply) two M, it would seem natural that we could also subtract them. We can this function, which is the inverse of `combine`, `split`, which has the signature `split` $: \forall (m : \mathbb{N}) \to_0 \forall (n : \mathbb{N}) \to_1$ $.Weactuallyusethisafairbitmorethanweuse$`combine`$,as$`split`$doesn'timposeanyrestrictionsonthewitness,$

In a similar manner to how we proved (in ??) the commutativity of $\otimes$, we can prove that these are inverses.

Add something on meta-logical interpretation

Lemma 3.9. *Given that we have $f :=$ `split`, and $g :=$ `uncurry`$^1(\otimes)$*[3]*, then f and g are inverses.*

Proof. The type of f is $M\ (m + n)\ t\ w \to_1 M\ m\ t\ w \times^1 M\ n\ t\ w$, and that of g is $M\ m\ t\ w \times^1 M\ n\ t\ w \to_1 M\ (m + n)\ t\ w$, so there composition $f \circ g : (- : M\ (m + n)\ t\ w) \to_1 M\ (m + n)\ t\ w$ (the same for $g \circ f$). Any function from a unique object and that same unique object is an identity, thereby these are inverses Need citation □

Another relevant construction is multiplicity "joining" and its inverse, multiplicity "expanding".

Definition 3.10. We define `join` $: M\ m\ (M\ n\ t\ w)\ ?$[4]. Its definition is like that of natural number multiplication, with it defined as follows:

[2] You can also prove associativity and related properties, the proof is the same

[3] Given that we have $uncurry^1 : (a \to_1 b \to_1 c) \to (a \times^1 b) \to_1 c$

[4] For simplicity, we infer the second witness

```
app : Mu n (t -@ u) wf -> Mu n t wx -@
↪  Mu n u (wf wx)
app MZ MZ = MZ
app (MS f fs) (MS x xs) = MS (f x) (app
↪  fs xs)
```

```
map : (f : t -@ u) -> Mu n t w -@ Mu n
↪  u (f w)
map f MZ = MZ
map f (MS x xs) = MS (f x) (map {w=x} f
↪  xs)
```

Listing 5. Definition of map and app

$\texttt{join}\diamond := \diamond$
$\texttt{join}(x_0 \odot x_1) := x_0 \otimes (\texttt{join} x_1)$

This is equivalent to flattening of values, and bears the same name as the equivalent monadic operation, `join`. Just like ⊗had the inverse `split`, `join` has the inverse `expand`

3.4 Applications over Mu

There is still one crucial operation that we have not yet mentioned, and that is application over M. That is, we want a way to be able to lift a linear function into M. We can do this, and we call it `map`, due to its similarity to functorial lifting and its definition may be found **??**.

However, there is something unsatisfying about `map`. Namely, the first argument is unrestricted. However, we *know* exactly how many of the function we need. It will simply be the same as the number of arguments.

So, we define another function, `app`, which has the type $\texttt{app} : (f : M\ n\ (t \to_1 u)\ w_f) \to_1 (x : M\ n\ t\ w_x) \to_1 M$ and a definition given at **??**

Notably, if we define a function $\texttt{genMu} : (x : !_* t) \to_1 \forall(n : \mathbb{N}) \to_1 M\ n\ t\ x.\texttt{unrestricted}$, which is simply defined as $\texttt{genMu}(\texttt{MkBang\_})0 := \diamond$ and $\texttt{genMu}(\texttt{MkBang}x)(Sn) := (x \odot (\texttt{genMu}n(\texttt{MkBang}x)))$, we can define `map`as $\texttt{map}\{n\} f x := \texttt{app}(\texttt{genMu} f n) x$.

This isn't exactly correct

3.5 Applicative Mu

We can use `app` to derive equivalents of the `push` and `pull` methods of Granule, in a similar way to how we can use $< * >$ to define mappings over pairs. In Idris, we define push : $M\ n\ (t \times_1 u)\ (w_0 *_1 w_1) \to_1 M\ n\ t\ w_0 \times_1 M\ n\ u\ w_1$ and `pull` : $M\ n\ t\ w_0 \times_1 M\ n\ u\ w_1 \to_1 M\ n\ (t \times_1 u)\ (w_0 *_1 w_1)$.

Expose

However, while not having a very interesting implementation, it does allow something very interesting to be stated, that morphisms on M types are internal to M. That is, we don't need to use an ω binding to model functions on M, we can instead just use M types themselves. This is very important: we can model linear mapping as a linear map.

3.6 Infinite Co-Mu

We also define a coinductive version of mu that works with conatural numbers (as opposed to just natural numbers). While these "co-mu" types are a extension of mu types, they are much more difficult to work with. This is because matching on them no longer involves matching on the result of a constructor `MS`, but rather on the result of a function.

To make these slightly easier to work with, we define functions from these `CMu` types and `Mu` types, as to allow us to re-use the same functions for `CMu`. Need code

4 Resource Algebras as Types

In many programming languages, algebras are used as a supplement to the type system to model various concepts. Among these, Granule uses a resource algebra to model multiplicity [12]. However, a number of libraries in Haskell use the type system itself to model algebras Need citation.

It stands to reason then that it should be possible, with mu types and Idris' rich type system, to model the resource algebras of Granule. We propose that this is indeed possible with a definition of `Form′` types.

4.1 Formula Language

Definition 4.1. `Form′` is a polymorphic type indexed by a $\mathbb{N}$. We also define a function, $\texttt{Eval}' : \texttt{Form}'n \to_1 \texttt{QVect}n\mathbb{N} \to \mathbb{N}$ Further, we define a function $\texttt{Solve}' : \texttt{Form}'n \to_1 \mathbb{N} \to_1 *$, which is defined as $\texttt{Solve}'\phi x := \exists_1 (x : \texttt{Form}'n)\texttt{Eval}'\phi x =_? y$. In addition, we define $\texttt{Unify}'\phi\psi := \forall(n : \mathbb{N}) \to_1 \texttt{Solve}'n\phi -$

We will write $x \in \phi$ or $\phi \ni x$ for $\texttt{Solve}'\phi x$, and $\phi \subseteq \psi$ $\psi \supseteq \phi$ for $\texttt{Unify}'\phi\psi$. Notably, this means that $\phi \subseteq \psi := \forall(n : \mathbb{N}) \to_1 \phi \ni n \to \psi \ni n$. This allows us to consider formulas as "sets" of natural numbers, those being all their possible outputs. We then say that a given number is "in" the formula if it is possible for it to be output, and a subset if every "element" is in the superset. We define the interpretation of each constructor of `Form′` based off its branch of `Eval′`

Need proof

This means that `Form′` forms a category on $\subseteq$

4.2 The Core Formulas

Out of six total constructors of formulas, four of them are binary constructors modeling addition, multiplication, joins and meets, while the other two model the basic notion of "variable" and "constant" in a formula.

These last two we will discuss first. The first of these is `FVar′`, which has the type `Form′1`. It models the notion of a singular variable in the formula. It evaluates as $\texttt{Eval}'\texttt{FVar}'[x] := x$, notably, however, this is the only branch, as the only possible index that `FVar′` can produce is 1.

Of all the formulas, `FVar′` is the most general. That is to say, it is the terminal object in the category of `Form′`.

Need proof

The next of these, `FVal′`, models the notion of "constants" in formulas. It has the form $\texttt{FVal}' : \mathbb{N} \to_1 \texttt{Form}'0$, and has the evaluation of $\texttt{Eval}'(\texttt{FVal}'n)[] := n$

4.3 The Binary Constructors

The remaining constructors of $\texttt{Form}'n$ all have the exact same form, that of $\forall(a : \mathbb{N}) \to_1 \forall(b : \mathbb{N}) \to_0 \texttt{Form}'a \to_1 \texttt{F}$ b. They are `FAdd′`, `FMul′` `FMax′`, `FMin′`, which model addition multiplication joins and meets. The branch of `Eval′` FAdd' p q n is as follows:

Need code

The other branches are defined similarly, simply switching out the operation of addition for whatever operation is appropriate.

One of the important facts about all of these constructors is that they operate on two formulas *Indepently*. That is, if $y \in \phi$ and $z \in \psi$, then it must also be true that $(y + z) \in (\texttt{FAdd}'\phi\psi)$, and similarly for all the others. So, we can reason about both parts of this completly independetly, which will be useful for creating a decision procedure of `Solve′`.

4.4 Abstract Forms

Almost all of the reasoning we do about `Form'` is about their outputs, but *not* their inputs. Because of this, it would be useful to be able to define an abstract `Form`, that is nullarly and instead abstracts existentially over the $\mathbb{N}$ indice. We define it as

Add def

4.5 Decidability of Solutions

One of the most important properties of the algebra that we have defined here is that is provably decidable, that is, everything can be determined to be a solution of a formula or a contradiction of such by a terminating function.

We prove each part individaully

Lemma 4.2. *$x \in$ `FVal'`n is decidable*

Proof. This boils down to $\exists_1(y : \mathbb{N})(x =_? n)$, and, as equiality of $\mathbb{N}$ is decidable, is itself decidable □

Likewise, variable are also decidable

Lemma 4.3. *$x \in$ `FVar'` is decidable*

Proof. This boils down to $\exists_1(y : \mathbb{N})x =_? y$, which is trivially true (x, x) □

Exampl of formulas

5 Omega Types

While mu types allowed us to generalize linear quanties to graded ones, we have a construction, Ω, that generalizes mu types to arbitary formulas. We do this by specifing a quantity formula for the type, rather than a specific quantity.

Definition 5.1 (Omega definition). Ω is a type indexed on a `Form`, a type, and a witness of that type such that $\Omega\ p\ t\ w := (n : \mathbb{N}) \to_1 \forall n \in p \Rightarrow_0 M\ n\ t\ w$

That is, it maps a number that satisfies the property of being a solution to a formula to that many bindings of that value.

5.1 Basic Properties

One of the most signaficant properties of Ω is that it can be "weakened". That is, if a given formula p is "in" q, then there exists a function from $\Omega\ q\ t\ w$ to $\Omega\ p\ t\ w$, which we call `weaken`. The reason that this must exist is apparent upon expanding the types, whence we get `weaken` : $\forall p \subseteq q \Rightarrow_0 ((m : \mathbb{N}) \to_1 \forall m \in q \Rightarrow_0 M\ m\ t\ w) \to_1 ((n : \mathbb{N}) \to_1 \forall n \in p \Rightarrow_0 M\ n\ t\ w)$, which is trivial

Motavat example

5.2 Operations on Omega

5.3 Completeness and Uniqueness of Omega

There are two very important facts about omega types, specifically with reference to mu types. Namely, they both revolve around the idea that omega types can serve as an exact model for mu types. The first of these is the fact that forall $M\ n\ t\ w$, there is a linearly isomorphic Ω

5.4 Infinite Lazy Copies

6 Exponential Types

6.1 Exponential Types as Graded Values

6.2 Operations on Exponential Types

6.3 The type of Strings Squared

6.4 Inverting Types

7 Using Mu and Friends

7.1 Sources and Factories

8 Conclusion

Related Work

GrTT and QTT. While QTT, in particular as described here, is quite useful, it of course has its limits. In particular, the work of languages like Granuleto create a generalized notion of this in a way that can easily be inferred and checked is important. However, in terms of QTT (or even systems outside it) this is far from complete.

The Syntax. For instance, even with the ^types, the syntax for this, and more generally linearity in general, tends to be a bit hard to use. A question of how to integrate into the source syntax would be quite interesting. Also, in general, one of the advantages of making an algebra part of the core language itself (as opposed to a construction on top of it) is that it makes it easier to create an inference engine for that language..

Bump Allocation. QTT has been discussed as a potential theoretical model for ownership systems. One of the more useful constructs in such a system is bump allocation. With particular use seen in compilers, bump arena allocation, where memory is pre-allocated per phase, helps both separate and simplify memory usage. It is possible that a usage of M types (given the fact that we know exactly how many times we need a value) as a form of modeling of arena allocation might be useful.

Acknowledgements

Thank you to the Idris team for helping provide guidance and review for this. In particular, I would like to thank Constantine for his help in the creation of M types

Artifacts

All Idris code mentioned here is either directly from or derived from the code in the Idris library `idris-mult`, which may be found at its repository[5]

References

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

[2] *base*. URL: https://www.idris-lang.org/Idris2/base/.

[3] Jean-Philippe Bernardy et al. "Linear Haskell: practical linearity in a higher-order polymorphic language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158093. URL: https://doi.org/10.1145/3158093.

[5] Some listings may be modified for readability

[4] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. "Effects as capabilities: effect handlers and lightweight effect polymorphism". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428194. URL: https://doi.org/10.1145/3428194.

[5] Edwin Brady. "Idris 2: Quantitative Type Theory in Practice". In: 2021, pp. 32460–33960. DOI: 10.4230/LIPICS.ECOOP.2021.9.

[6] Maximilian Doré. *Dependent Multiplicities in Dependent Linear Type Theory*. 2025. arXiv: 2507.08759 [cs.PL]. URL: https://arxiv.org/abs/2507.08759.

[7] Maximilian Doré. "Linear Types with Dynamic Multiplicities in Dependent Type Theory (Functional Pearl)". In: *Proc. ACM Program. Lang.* 9.ICFP (Aug. 2025). DOI: 10.1145/3747531. URL: https://doi.org/10.1145/3747531.

[8] Jean-Yves Girard. "Linear logic". In: *Theoretical Computer Science* 50.1 (1987), pp. 1–101. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(87)90045-4. URL: https://www.sciencedirect.com/science/article/pii/0304397587900454.

[9] Daan Leijen. "Koka: Programming with Row Polymorphic Effect Types". In: *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), pp. 100–126. ISSN: 2075-2180. DOI: 10.4204/eptcs.153.8. URL: http://dx.doi.org/10.4204/EPTCS.153.8.

[10] Magnus Madsen and Jaco van de Pol. "Polymorphic types and effects with Boolean unification". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428222. URL: https://doi.org/10.1145/3428222.

[11] Danielle Marshall and Dominic Orchard. "How to Take the Inverse of a Type". In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Ed. by Karim Ali and Jan Vitek. Vol. 222. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 5:1–5:27. ISBN: 978-3-95977-225-9. DOI: 10.4230/LIPIcs.ECOOP.2022.5. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2022.5.

[12] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. "Quantitative program reasoning with graded modal types". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341714. URL: https://doi.org/10.1145/3341714.

[13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.