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

1 Introduction

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 [4, 2, 7, 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 . 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 [9, 3, 8].

2 Background

2.1 Copying and Dropping

Utlitmaly, 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, , 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, `copy` $: ((x : a) \rightarrow_1 (y : a) \rightarrow_1 ($
This essientally “uses a value twice” in an arbitrary (potentially dependent) function, and an erased proof that `copy` $fz =_? fzz$.

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

[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

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`.

Theorem 2.1 (Finite Initial). *There is a finite list of all the inhabitents of* $\Sigma^1{}_{n:\mathbb{N}}(n \leq v)$, *where* $z : \mathbb{N}$

Proof. TODO □

Conatural Numbers.

2.3 Dependent Pairs

As is common, in Idris, existentials are modeled as a dependent pair [13] [12]. With Idris, however, there is an extra portion to this. Specifically, we can also change the multiplicity of the first and second element of the pair. In addition, we can chose to make these linear, unrestricted, or erased. Because one of the central goals of this is to create a system that does not use unrestricted bindings, we focus only on the linear and erased possibilities.

The possible combinations, and their names in Idris, as well as their constructors, are given below

	0	1
0	Σ^1, `Sigma`, `For`	$\exists^1$, `Exists`, `Given`
1	Σ^0, `Subset`, `Elem`	Σ^1, `Sigma`, `For`

We write all of these with the same "typebind" shorthand, which, when binding application is added, will allow the above to be written as `For`$(x : \mathbb{N})|(x =_? x)$, for instance [14].

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 2.

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

In addition, M has two constructors

Definition 3.2. There are two constructors of M, ⊠ `MZ`, and ⊙, `MS`, Which have the signatures ⊠ : $\mathscr{M}\ n\ t\ w$ and $\odot : (w : t) \rightarrow_1 \mathscr{M}\ n\ t\ w \rightarrow_1 \mathscr{M}$ (`Succ`n) $t\ w$

```
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 2. The definition of M in Idris

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

Remark 3.3. $\mathscr{M}\ 0\ t\ w$ can only be constructed by ⊠.

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 $x\ :\ \mathscr{M}\ 2$ `String` `"value"`, we can only construct this through ⊙, 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 $\mathscr{M}\ 1\ \ String$`"value"`, which we then repeat one more time and get another `"value"` an finally we match $\mathscr{M}\ 0$ `String` `"value"` and get that we must have ⊠.

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

The `w` index or "witness" is the value being copied. Notably, if we remove the `w` index, we would have ⊠ $:\ Mnt$, and ⊙ $:\ t\ \to_1\ Mnt\ \to_1\ M(Sn)t$, 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` $:_0\ \forall(w\ :\ t) \to_0 \mathscr{M}\ n\ t\ w\ \to\ t$. 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 . In addition, we also have `drop` $:\ \mathscr{M}\ 0\ t\ w \to \top$, which allows us to drop a value. Its signature is simple `drop`⊠ $:= ()$.

Finally, we have once, which takes a value of form $M1$ and extracts it into the value itself. It has a signature `once` $:\ \mathscr{M}\ 1\ t\ w\ \to_1 t$, where we have `once`$(x$⊙⊠$)\ := x$

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 $\mathscr{M}\ n\ t\ w$, so long as that type is well formed.

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

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

```
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 3. Proof of 3.5 in Idris

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" .

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

Proof. Induct on n.

- The first case, where x and y are both $\mathscr{M}\ 0\ t\ w$, is trivial because, as per 3.3 they both must be ⊠
- The inductive case, where, from the fact that for any a and b (both of $\mathscr{M}\ n'\ tw$) we have $a =_? b$, we prove that we have, for x and y of $\mathscr{M}\ (Sn')\ t\ w$ that $x =_? y$. We note that we can destruct both of these, with $x_1 : \mathscr{M}\ n'\ tw$ and $y_1 : \mathscr{M}\ n'\ tw$, 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 $\mathscr{M}\ n'\ tw$ has the above property, which is the induction hypothesis. □

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 $\mathscr{M}\ n\ t\ w$ is well-formed, then it must have* exactly *one inhabitant.*

Proof. We know by 3.5 that there is *at most* one inhabitant of $\mathscr{M}\ n\ t\ w$. We then induct on n to show that there must also exist *at least* one inhabitant of the type.

- The first case is that $\mathscr{M}\ 0\ t\ w$ is always constructible, which is trivial, as this is just ⊠.
- The second case, that $\mathscr{M}\ n'\ tw$ provides $\mathscr{M}\ (Sn')\ t\ w$ being constructible is also trivial, namely, if we have the construction on $\mathscr{M}\ n'\ tw$ 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 $\mathscr{M}$. 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.

One important derivation of this is that if $\mathscr{M}ntw$ is well formed, we can *always* construct an inhabitant of it.

Lemma 3.7 (Examples of Mu). *There is a function,* `Example` $: \forall(n : \mathbb{N}) \to_0 \forall(t : *) \to_0 \forall(w : t) \Rightarrow_0 \mathscr{M}\ n\ t\ w$

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

```
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

$$\boxtimes \otimes x \quad := x \qquad (1)$$

$$Z + x \quad := x \qquad (2)$$

$$(a \odot b) \otimes \quad x := a \odot (b \quad \otimes x) \qquad (3)$$

$$(Sn) + \quad x := S(n \quad + x) \qquad (4)$$

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 [11]. It instead does this similarly to how others have embedded QTT in Agda [6, 5].

Namely, rather than viewing $\mathscr{M}\ n\ t\ w$ 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 $\Gamma \vdash \phi \,:\, \mathscr{M}\ r\ a\ x$, 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 3.8. 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.

The fact that these are equivalent becomes even more clear when we create ⊗, `split`, and `join` and expand, which are included in the definition of context concatenation and flattening in Granule [5].

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 $\otimes \,:\, \forall(m \,:\, \mathbb{N}) \to_0 \forall n \to_0 \mathbb{N} \mathscr{M}\ m\ t\ w \to_1 \mathscr{M}\ n\ t\ w \to_1 \mathscr{M}\ (m+n)\ t\ w$, 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 $(Sn) + x \ := S(n + x)$

In addition, using the assumption of 3.7, we can liken the function ⊗ to context concatenation [11]. However, unlike Granule, we define this in the language itself. Per as a matter of fact, it isn't actually possible to construct ⊗in Granule, as it would require a way to reason about type level equality, which isn't possible [**granule**].

Lemma 3.9. *$\otimes$ is commutative*[2]*, that is, $x \otimes y =_? y \otimes x$.*

Proof. These types are the same, and by 3.6, 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 signatureWe actually use this a fair bit more than we use `combine`, as `split` doesn't impose any restrictions on the witness, which is very useful for when we discuss $^\wedge$types.

In a similar manner to how we proved (in 3.8) the commutativity of $\otimes$, we can prove that these are inverses.

Lemma 3.10. *Given that we have f := `split`, and g := `uncurry`$^1(\otimes)$*[3]*, then f and g are inverses.*

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

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

Definition 3.11. We define `join` : $\mathscr{M}\ m\ (\mathscr{M}\ n\ t\ w)\ ?$[4]. Its definition is like that of natural number multiplication, with it defined as follows:

`join`⊠ := ⊠
`join`$(x_0 \odot x_1)$:= $x_0 \otimes$ (`join`x_1)

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

3.4 Applications over Mu

One of the most important operations possible operations on $\mathscr{M}$ is `map`, which takes a given linear function $f\ :\ (x\ :\ t) \to_1 (px)$ and maps it to a function that takes in a type of $\mathscr{M}$ and returns a type on $\mathscr{M}$. This defines a "functor" from the category of linear functions on Idris types to the "category" of $\mathscr{M}$ on its second and third arguement. However, `map` will have a couple cavaets:

- It will have the morphism be external to the linear category, as it will need to use them more than once.
- It has to map over *both* the type and the value inside $\mathscr{M}$.

Lemma 3.12 (Mapping Mu). *The is a function* `map` $:\ (f\ :\ ((x\ :\ t) \to_1 (px))) \to_\omega \mathscr{M}\ n\ t\ w \to_1 \mathscr{M}\ n\ (pw)\ (fw)$

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` $:\ \mathscr{M}\ n\ (t{\times_1}u)\ (w_0{*_1}w_1) \to_1 \mathscr{M}\ n$
and `pull` $:\ \mathscr{M}\ n\ t\ w_0{\times_1}\mathscr{M}\ n\ u\ w_1 \to_1 \mathscr{M}\ n\ (t{\times_1}u)\ (w_0{*_1}w_1)$.

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.

[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

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`.

4 Resource Algebras as Types

In many programming languages, algebras are used as a supplement to the type system to model various concepts [3] [8] [9]. Among these, Granule uses a resource algebra to model multiplicity [11]. However, a number of libraries in Haskell use the type system itself to model algebras .

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, $\texttt{Eval}' : \texttt{Form}'n \rightarrow_1 \texttt{QVect}n\mathbb{N} \rightarrow \mathbb{N}$ Further, we define a function $\texttt{Solve}' : \texttt{Form}'n \rightarrow_1 \mathbb{N} \rightarrow_1 *$, which is defined as $\texttt{Solve}'\phi x := \exists^1_{(x:\texttt{Form}'n)} \texttt{Eval}'\phi x =_? y$. In addition, we define $\texttt{Unify}'\phi\psi := \forall(n : \mathbb{N}) \rightarrow_1 \texttt{Solve}'n\phi$ –

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 $\phi \subseteq \psi := \forall(n : \mathbb{N}) \rightarrow_1 \phi \ni n \rightarrow \psi \ni n$. 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′`

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

4.2 The Core Formulas

The first `Form′` iS these is `FVar′`, which has the type $\texttt{Form}'1$. 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′`.

Lemma 4.2. *`FVar′` $\ni n$ is trivial.*

Proof. This expands to $\exists^1_{(x:\mathbb{N})}(\texttt{Eval}'\texttt{FVar}'x =_? n)$, which, if we have (n, α), where α is $\texttt{EvalFVar}'n =_? n$, which is trivial. □

The next of these, `FVal′`, models the notion of "constants" in formulas. It has the form $\texttt{FVal}' : \mathbb{N} \rightarrow_1 \texttt{Form}'0$, and has the evaluation of $\texttt{Eval}'(\texttt{FVal}'n)[] := n$

4.3 The Binary Constructors

The remaining constructor models the notion of "binary operations" on `Form′`. It allows us to create a very basic tree of quantity expressions, which, combined with `FVal′` and `FVar′`, allow us to model literals, variables, and "applications" of either addition, multiplication, minimims and maximums to those formulas.

It makes use of a enumeration type, `FOp`, which are attatched to each operation on two values.

Definition 4.3. We define $\texttt{FOp} = +|-|min|max$, and also define $\texttt{runOp}$ that has the type $op \rightarrow_1 \mathbb{N} \rightarrow_1 \mathbb{N} \rightarrow_1 \mathbb{N}$, such that we map the operation in $\texttt{FOp}$ to its corresponding two arguement function

$\texttt{FApp}'$ then takes that operations and applies it to two formulas. This allows us to create "quantity expressions".

Definition 4.4. $\texttt{FApp}'$ is of the type $\texttt{FApp}' : (op : \texttt{FOp}) \rightarrow_1 \forall(a : \mathbb{N}) \rightarrow_1 \forall(b : \mathbb{N}) \rightarrow_0 \texttt{Form}'a \rightarrow_1 \texttt{Form}'b \rightarrow_1$(

Note that we *add* the number of variables in the types, and the first number is linear, not erased. This has to do with the system of variables in $\texttt{Form}'$: each variable can only occur once. While this does limit the power of $\texttt{Form}'$, it also makes it substaintally simplifies the solving of $\texttt{Form}'$.

This allows us to reason about formulas easily, because we know that each part of the formula can be solved independtly.

Lemma 4.5. *If $\phi \ni x$, and $\psi \ni y$, then $\texttt{FApp}'op\phi\psi \ni (\texttt{runOp}\ op)xy$.*

While this dosen't look very intuitive, this is simply the fact that $\phi + \psi \ni x + y$ and so on.

Proof. □

4.4 Abstract Forms

While so far we have been dealing with formulas with explicit numbers of variables. However, we almost never actually care about the inputs to a formula, we care only about the outputs. Neither $x \in \phi$ or $\phi \subseteq \psi$ is dependent on the type of the given formulas. So, rather than dealing with the "concrete" types, we instead deal with an abstract type, $\texttt{Form}$, which is a dependent linear pair of the form $\Sigma^1{}_{n:\mathbb{N}}(\texttt{Form}'n)$.

We then apply equivalents to each operation that works on $\texttt{Form}$ instead of $\texttt{Form}'\ n$, and there name is the respective operation, merely with the prime dropped from the end We also define $\texttt{Solve}$ and $\texttt{Unify}$, which are defined likewise. Finally, we use the same notation for $\texttt{Form}$ as $\texttt{Form}'n$, 5 for $\texttt{FVal}\ 5$, $\phi + \psi$ for $\texttt{FAdd}\phi\psi$, and $x \in \phi$ for $\texttt{Solve}$ and $\phi \subseteq \psi$ for $\texttt{Unify}\phi\psi$.

Altogether, the formula language is as follows:

$$\begin{array}{llcll}
 & \text{op} & \boxtimes & + & \\
 & & | & * & \\
 & & | & \text{min} & \\
 & & | & \text{max} & \\
\text{Form} & \phi & \boxtimes & n & \text{Number} \\
 & & | & \_ & \text{Variable} \\
 & & | & \phi \text{ op } \phi & \text{Application}
\end{array}$$

4.5 Decidability of Formulas

One of the reaons this specific set of formula types was chosen is that it allows for $\phi \ni n$ to be provably terminadbly decidable for any formula and natural number. That is, their is a total function $\texttt{DecSolve}$ from a formula and natural number to either $\phi \ni n$ or a proof that it is absurd

We prove this by case anaylisis and induction. Firstly, we have the two base cases:

Lemma 4.6. *$\texttt{FVar}' \ni n$ is decidable.*

Proof. This reduces to $\exists^1_{(x:\mathbb{N})}(\texttt{Eval}'\texttt{FVar}'x =_? n)$, which simply has $(n, \texttt{Refl})$. □

Lemma 4.7. *$\texttt{FVal}'v \ni n$ is decidable.*

Proof. This reduces to $\exists^1_{(x:\mathbb{N})}(\texttt{Eval}'\texttt{FVal}'vx =_? n)$, which further reduces to $\exists^1_{(x:\mathbb{N})}(v =_? n)$, and, as natural number equality is decidable, is decidable □

Their are 4 more inductive cases, one for each operation.

However, first, we must define the induction hypothesis. Because for each case they are exactly the same, we note them all here.

4.6 Formula Syntax Sugar

5 Omega Types

Omega types allow us to generalize mu types to bindings that have mutiple possible values. For instances, in the Granule binding `x : t [2*c]`, the binding has a variable multiplicity given by the effect formula $2 * c$ [5]. This allows for Granule to have, say, a function `mapMaybe` which has the form $(a \rightarrow_1 b) \rightarrow_{0..1} (\texttt{Maybe} a) \rightarrow_1 (\texttt{Maybe} b)$.

Of course, this is just one example of many of the potential utility of such a system, perhaps the most interesting of which is modeling the idea of optional ownership. We propose Ω, which model such bindings of variable multiplicity using a continuations on the exact number of bindings

Ω types allow for bindings that have multiplicity polymorphism. The simplest example of this is a binding that may or may not be used. In Granule such bindings are created by allowing for effect formulas to serve as multiplicities

5.1 Extended Mu

Definition 5.1 (Omega types). Ω is an erased function that takes a `Form` as an arguement, as well as a type, t, and a witness of t, which, altogether, has the singature $\Omega : \texttt{Form} \rightarrow (t : *) \rightarrow w \rightarrow *$. Its definition is $\Omega\ \phi\ t\ w := (n : \mathbb{N}) \rightarrow_1 \forall (n \in \phi) \Rightarrow_0 M\ n\ t\ w$

This is simplest understood by example.

The easiest form of this is $\Omega\, \texttt{FVar}\, t\, w$, which expands to the type $(n : \mathbb{N}) \rightarrow_1 \forall (n \in \texttt{FVar}) \Rightarrow_0 M\ n\ t\ w$. Per 4.2, this becomes simply $(n : \mathbb{N}) \rightarrow_1 \mathscr{M}\ n\ t\ w$. Thereby, this is simply a mapping from any number of bindings to that many bindings of the form $w : t$.

Another simple form of Ω is that where the formula is some `FVal`. This type, given that the specific number is m, expands to $(n : \mathbb{N}) \rightarrow_1 \forall (n \in \texttt{FVal}\ m) \Rightarrow_0 M\ n\ t\ w$. Because $\texttt{FVar} m \ni n$ only exists if $n =_? m$, we know that this will simply be equivalent to $\mathscr{M}\ m\ t\ w$.

5.2 Operations on Omega

Among the more important operations on Ω

Given the fact that Ω attempts to generalize M, it stands to reason that each of the operations on $\mathscr{M}$ have equivalents on Ω. This is, unfourtanetly, only partially true.

While we can create equivalents of `combine` and `join`, we *cannot* create the equivalents of `split` and can only create a specific version expand. The reason for this is as follows

5.3 Infinite Lazy Copies

6 Exponential Types

Linear exponential types have been noted previously to be equivalent to a certain number of bindings of a value. Indeed, this is why they are called *exponential* types, as, say, $\texttt{String}^3$ models $\texttt{String} \times \texttt{String} \times \texttt{String}$. Here, we construct `Exp`, which models exponential types using Ω types abstracted over their witness. This allows us to transform the Ω, which requires a witness of the value, to something more closely resembling the type of graded value *themselves*.

Definition 6.1 (Exponential Types). `Exp` is a type function indexed on a formula, ϕ, and type, t, with the definition of $\texttt{Exp}\phi t := \exists^1_{(w:t)} (\Omega\ \phi\ t\ w)$.

We also, as the name suggests, employ a terse notation using the syntax sugar described for formulas earlier 4.6 to write these more expressivly.

Notation 6.2. We write $t^\wedge\phi$, where ϕ is a formula-like object and t is a type for $\texttt{Exp}\phi t$, and, for an even simpler syntax, t^ϕ.

This syntax allows us to write, say t^2 for $\texttt{Exp}(\texttt{GivenOFVal}'(S(S0)))t$, which is obviously much clearer [5].

6.1 Exponential Types as Graded Values

Exponential types serve the primary purpose of modeling linear values, rather than the bindings of those values. So, for instance, the granule type `String[2]` is modeled by $\texttt{String}^2$ One of the most important facts about exponential types is that they are functorial over *both* their first and second arguements[6].

Lemma 6.3. *Exponential types have a function* $\texttt{map}_{\texttt{Exp}_2} : \forall(p : t \rightarrow_1 u) \rightarrow_0 (\forall(w_t : t) \rightarrow_0 \Omega\ \phi\ t\ w_t \rightarrow_1 \Omega\ \psi\ u\ ($
[7]

Proof. For $\texttt{map}_{\texttt{Exp}_2} f x$, we have $Given(px_1)(\texttt{map}_\Omega f x_2)$ □

Next, and slightly more interesting, is that over the second

Lemma 6.4. *Exponential types have a function* $\texttt{map}_{\texttt{Exp}_1} : \forall(\phi \subseteq \psi) \Rightarrow_0 t^\phi \rightarrow_1 t^\psi$ *(we call this* `weaken`*)*

Proof. We simply have □

6.2 The type of Strings Squared

6.3 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 $^\wedge$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..

[5]Obviously, this example is a little bit ridiculos, as we already have, say, $2 =_? (S(S0))$, but it still is true that this is a more terse syntax
[6]With respect to $\subseteq$
[7]Note that when we want to disambitguate among $\mathcal{M}$ and $\Omega$and Exp types, we use a subscript to differentiate them

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[8]

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] 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.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[8]Some listings may be modified for readability

[10] 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.

[11] 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.

[12] EGBERT RIJKE and BAS SPITTERS. “Sets in homotopy type theory”. In: *Mathematical Structures in Computer Science* 25.5 (Jan. 2015), pp. 1172–1202. ISSN: 1469-8072. DOI: 10.1017/s0960129514000553. URL: http://dx.doi.org/10.1017/S0960129514000553.

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

[14] André Videla. *Binding Application Proposal & Design*. URL: https://github.com/idris-lang/Idris2/issues/3582.