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

Update everything

1 Introduction

One of the more interesting developments in Programming Language Theory is Quantative 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 foriegn interfaces Need citation. Apart from just the theoretical intrest, QTT has the potential to serve as an underlying logic for languages like Rust, where reasoning about resources takes the forfront.

However, new developments in Graded Modal Type Theory (GrTT), in particular with Granule, serve to create a finer grained[1] notion of usage thatn 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 algbras" 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 [11, 4, 10].

Preliminaries

We write the morphisms in a given category $\to_C$, where C is the category in question. We define 1 to be the category of Idris terms where the morphisms are all linear mappings, 0 to be that where they are all erased, and ω for those that are unrestricted. We write these arrows at $\to_1$, $\to_0$, and $\to_\omega$, respectively.

We also write isomorphisms in a certain category C as $\simeq_C$, and in particular we have $\simeq_1$ as the isomorphism in the linear category. We write implicit Π types prefixed with a $\forall$, so, for instance, we have $\forall(x : t) \to_1 u$ as the equivalent of the Idris `{1 x : t} -> u`. We write $\times^C$ for the product

[1]Hence the name

Author's Contact Information: Asher Frost.

```
data Mu : (n : Nat) -> (t : Type) -> (w : t) -> Type where
        MZ :
                Mu Z t w
        MS :
                (1 w : t) ->
                (1 xs : (Mu n t w)) ->
                Mu (S n) t w
```

Listing 1. The definition of M in Idris

construction in C, and for a given evidence of that construction we write that with the constructor $*^C$

In addition, note that we write $\mathbb{N}$ for the type of *linear* natural numbers Need citation.

2 *M* 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 1.

Definition 2.1. M is a polymorphic type with signature $M : (n : \mathbb{N}) \to_0 (t : *) \to_0 (w : t) \to_0 *$

In addition, M has two constructors

Definition 2.2. There are two constructors of M, $\diamond$ `MZ`, and $\odot$, `MS`, Which have the signatures $\diamond : M\ n\ t\ w$ and $\odot : (w : t) \to_1 M\ n\ t\ w \to_1 M\ (Sn)\ t\ w$

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 succesor of whatever is given in.

Remark 2.3. $M\ 0\ t\ w$ 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, bu 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 windex, we would have $\diamond : Mnt$, and $\odot : t \to_1 Mnt \to_1 M(Sn)t$, 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` $:_0 \forall (w : t) \to_0 M\ n\ t\ w \ \to\ t$. 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 Need citation. In addition, we also have `drop` $: M\ 0\ t\ w \to_1 \top$, which allows us to drop a value. Its signature is simple `drop`$\diamond := ()$.

```
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 2. Proof of 2.5 in Idris

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$(x \odot \diamond) := x$

2.1 Uniqueness

While wserves 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 2.4. *$x : M\ n\ t\ w$ 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 $x : M(Sn')tw$, 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 then 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 2.5. *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 2.3 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 hypthesis, $x_1 =_? y_1$

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

Finish proof

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

Theorem 2.6 (Uniqueness). *If $M\ n\ t\ w$ is well formed, then it must have* exactly *one inhabitent.*

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

```
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

Proof. We know by 2.5 that there is *at most* one inhabitent of $M\ n\ t\ w$. We then induct on n to show that there must also exist *at least* one inhabitent 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 inhabitent, we can prove that there is exactly one inhabitent. □

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.

2.2 Graded Modalities With M

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 [13]. It instead does this in a similar way 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 2.7. 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.

2.3 Operations on M

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

Fix the alignement of these

In addition, using the assumption of 2.7, we can liken the function $\otimes$ to context concatenation [13]. 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.

$$\diamond \otimes x := x \tag{1}$$
$$Z + x := x \tag{2}$$

$$(a \odot b) \otimes x := a \odot (b \otimes x) \tag{3}$$
$$(Sn) + x := S(n + x) \tag{4}$$

```
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 3. Definition of mapin Idris

Lemma 2.8. *$\otimes$ is commutative*[2]*, that is, $x \otimes y =_? y \otimes x$.*

Proof. These types are the same, and by 2.6, 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 signature `split` : $(n : \mathbb{N}) \to_1 (- : M\ (m+n)\ t\ w) \to_1 M\ m\ t\ w \times^1 M\ n\ t\ w$. We 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 ^types.

In a similar manner to how we proved (in 2.8) the commutativity of $\otimes$, we can prove that these are inverses.

Add something on metalogical interpretation

Lemma 2.9. *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 2.10. We define `join` : $M\ m\ (M\ n\ t\ w)$?[4]. Its definition is like that of natural number multiplication, with it defined as follows:

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

2.4 Applications over M

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 3.

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

[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)
```

Listing 4. Definition of app

So, we define another function, app, which has the type app : $(f : M\ n\ (t \to_1 u)\ w_f) \to_1 (x : M\ n\ t\ w_x) \to_1 M$ and a defintion given at 4

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

This isn't exactly correct

2.5 Applicative M

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 : $M\ n\ t\ w_0 \times_1 M\ n\ u\ w_1 \to_1 M\ n\ (t \times_1 u)\ (w_0 *_1 w_1)$. Their construction is simple, and may be found in idris – mult6.

One of the more important constructions, particularly for 3, is that of react. react, which borrows its name from the notion of "statements as chemicals"

3 ω and Ω Types

The ω type and their generalization Ωtypes, serve to model unrestricted multiplicty. An unrestricted multiplicity as one that can create any number of linearly bound terms, which is written by Girard as ! ("of course"), ω in Idris, and t[] in Granule[8, 5, 13]. Further, in other works of embedding a linear logic system with unrestricted multiplicity into a more restrictive system, the system of natural number was extended to conatural numbers, which contain ∞.

Here, we propose Ωtypes, and a more specific form of them, ω types, which are written in idris – mult6as omega. We will first look at ω types, as they are a fair bit simpler than Ω types.

Definition 3.1. ω is a type alias which takes a type and a witness of that type as arguments, we define it as $\omega\ t\ w := (n : \mathbb{N}) \to_1 M\ n\ t\ w$

We can thence, if we have $x : \omega t w$, get any number of bindings of the value w of t. This can then be used to model multipliciites. Firstly, let us note how we might construct these.

Definition 3.2. gen goes from a value in a linear unrestricted multiplicity, $!_*$, and a value of ω. Spefically, gen has the type signature gen : $(x : (!_* t)) \to_1 \omega t$(unrestricted$x$).. It has the definition of Need code.

This also us to unpack a value of unrestricted multiplicity into some number of linear bindings.

Using omega types

3.1 Ω types

Whilst ω types allow us to model general unrestricted multiplicities, they dont allow for for restricted multiplicity polymorphic functions. For instance, consdier the function in GranulemapMaybe , type

We define this with ω and Mtypes as def where we define the erased value MapMaybeW.

However, this would be overly specific. Considering the Granuleequivalent, it is clear what the problem is. Namely, we will use f at most once, so we don't need *any* number of usages of f. We

```
record LExists {ty : Type} (f : (ty -> Type)) where
        constructor LEvidence
        0 fst' : ty
        1 snd' : f fst'
```

Listing 5. The definition of LExists

need at most one instance of f. This has real world consequences. For instance, if we have x : ℕ, we might construct $f : \mathbb{N} \to_1 \mathbb{N}$, $f y := x + y$.

In addition, we can discard this by merely doing fZero, which then has the type ℕ, which we can freely discard. So, we have a way to get either 0 or 1 instances of f, but not, in general, to get any number of these.

This isn't a mu type, as there are two choices, 0 and 1, for the multiplicity. However, it also isn't an ω type, as we need not be able to produce any number of values. To solve this dilemena, we introduce Ω types, a generalization of ω types to require only a subset of ℕ bindings.

Firstly, we note that is Idris, Type and Prop are synonymus. We also note that then a predicate on a type Pred a, is just equivalent to $a \to *$ [9]. We can likewise define a number of relevant operations on Pred a, which are given in more detail in idris − mult6Data.Mu.Maps.

One of the more interesting facts about predicates is that they are contravariant functors. We also define in a slightly less interesting fashioin a number of other general predicate functions.

We in particular focus our attetetion on Pred ℕ, which we can use to define Ωas follows:

Definition 3.3. Ωis a polymorphic type over a predicate on ℕ, p, a given type t, and a witness of that type w, $\Omega : (p : \text{Pred } \mathbb{N}) \to (t : *) \to (w : t) \to *$. We define Ω as $\Omega\ p\ t\ w := (n : \mathbb{N}) \to_1 \forall (\text{prf} : (pn)) \Rightarrow_0 M\ n\ t$

So, to get a certain number of bindings of w, we must provide a proof that the given number of bindings satisfies some arbitrary predicate p.

Finish, among things, work on 3A

4 Exponential and Existential Types

While the M and Ω types bear much intrest theoretically, they, by themselves, have little practical use. This is because as demonstrated in 2.2, these types model the *judgements* about graded modalities, not the graded modalities themselves.

Fortunately, however, we can define a simple abstraction over them that allows them to behave more like true graded modalities at the term level. To do this, we must first introduce a linear existential type. A regular existential type is a dependent pair type that "dosen't care" about its first argument. In Idris 2, we can make the that fact part of the programming by giving the first arguement a multiplicity of zero.

Idris 2 actually defines an existential type, however, it has the second arguement have a multiplicity of ω, while we want it to have a multiplicity one. Fortunately, the modification of Exists to our type, LExists, is quite trivial[5], and we define it in 5

We can also define mapping like how Idris defines the mapping, with the signature described in 6 [2].

4.1 Existential Crisis (Solution)

This principle issue with using M and Ω in practice is that

[5]Idris, however, seems to have trouble correctly linearlizing constructor accessors, so we define the actual fst and snd accessors separately

```
map :
        {0 p : a -> Type} ->
        {0 q : b -> Type} ->
        {0 m : (a -> b)} ->
        (1 f : forall x. p x -@ q (m x)) ->
        (LExists p -@ LExists q)
map f (LEvidence x y) = LEvidence (m x) (f y)
```

Listing 6. Definition of `map` for `LExists`

5 Using M and Friends

6 Conclusion

Related Work

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[6]

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.

[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] Jean-Yves Girard, Paul Taylor, and Yves Lafont. "Proofs and types". In: 1989. URL: https://api.semanticscholar.org/CorpusID:117808896.

[6] Some listings may be modified for readability

[10] 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.

[11] 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.

[12] 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.

[13] 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.