μ Types for a Multitude of Multiplicities

Asher Frost

September 26, 2025

Abstract

Idris2 uses Quantitative Type Theory (QTT) in a way that is quite practical and elegant. Described here is a way to model ω types using only linear ones, while obtaining multiplicity polymorphism, without extending the core language.

Todo list

Contents

1 Introduction

Idris shows the practical applications of QTT, a branch of type theory based around the notion of *usage*[3]. In particular, the ability for multiplicities to model both modality in logic and resource usage in programs adds yet another way to reason about programs, and is utilized by Linear Haskell (as an extension to the language) and as part of Idris 2 (as a core language) [3, 2]

Idris 2's use of QTT as a core language has been one of its strong selling points. There are quite a few things that can be easily done with restricted bindings that cannot be done any other way. For instance, they provide a good model for type variables, proof irrelevance at runtime, a less magical system of IO, among other properties.

Recently, however, there has been interest in GrTT, which allows for more dynamic restrictions on the usage of types, particularly allowing for usages greater than 1 [6]. Rather than arguing for the use of GrTT in Idris (which would add complexity), we instead propose a construction, M, that can model graded quantities using only linear types.

Furthermore, a system is described that *only* relies on types with multiplicities 0 and 1 to build a system (almost) as powerful as such as system with ω. We do this by using the exponential construction, here called M, to model arbitrary multiplicities.

2 M types

Mtypes aim to serve as a model for multiplicity polymorphism, without extending the core language of Idris. Moreover, a construction on M, Ω, can be used to model a form of unrestricted binding. The Mtype is indexed on three arguments, a "size", $n : \mathbb{N}$, a type $t : *$, and a witness of that type, $w : t$

Definition 2.1. The definition of $M$and its constructors are:
$M : (n :_0 \mathbb{N}) \to (t :_0 *) \to (w :_0 t) \to *$
$\diamond : \forall_* t \forall_t w (M\ 0\ t\ w)$
$\odot : \forall_* t \forall_{\mathbb{N}} n ((w : t) \multimap (M\ n\ t\ w) \multimap (M\ (S\ n)\ t\ w))$

```
data M : (n : Nat) -> (t : Type) -> {w : t} -> Type where
        MZ :
                {0 t : Type} ->
                {0 w : t} ->
                M Z t {w}
        MS :
                {0 t : Type} ->
                {0 n : Nat} ->
                {0 w : t} ->
                (1 w : t) ->
                (1 xs : Lazy (M n t {w})) ->
                M (S n) t {w}
```

Figure 1: Idris definition of M, as found in `Data.Mu.Types`

2.1 The Uniqueness Theorem

Theorem 2.2 (Uniqueness of M)**.** *If Δ_0 and Δ_1 both have type $M\ n\ t\ w$, and that type is constructible, then $\Delta_0 =_? \Delta_1$*

Proof. Let us induce on n.

The base case $n \equiv 0$, thereby Δ_0 and Δ_1 are of type $M\ 0\ t\ w$, which has only one constructor, $\diamond$, which is equal to itself.

The inductive case, then involves a proof that $a \odot \Delta_0' =_? b \odot \Delta_1'$, where $\Delta_0' : M\ n\ t\ w$, $\Delta_1' : M\ n\ t\ w$.

We note the type of $\odot : (w : t) \multimap M\ n\ t\ w \multimap M\ (Sn)\ t\ w$, and attempt to unify $\Delta_0' : M\ n\ t\ w$ with $M\ n\ t\ w$, and then infer the first argument must be of the form w. Thereby, $a =_? w$.

Do similar on b and Δ_1', and we also get $b =_? w$. We then rewrite, using both these two results $w \odot \Delta_0' =_? w \odot \Delta_1'$, then, we have $\Delta_0' =_? \Delta_1'$. This is the induction hypothesis. □

That is, given the fact that $M\ n\ t\ w$ is constructable, there must be exactly one inhabitant.

We can establish a number of interesting corollaries, including that products are unique:

Corollary 2.3. *If* Δ_0 *and* Δ_1 *both have type* $(M\ m\ t\ w_0 \times^1 M\ n\ u\ w_1)$*, and that type is constructible, then* $\Delta_0 =_? \Delta_1$

Proof. □ Add proof

2.2 Pushing and Pulling

Of course, at this point, we still can do almost nothing with the Mtype. To use it, we will need to define functions to help with its usages, we will start with the `push` and `pull` functions [6]. Specifically, we want two functions, `pushM` : $(M\ n\ (t \times^1 u)\ (w_0 *^1 w_1)) \multimap ((M\ n\ t\ w_0) \times^1 (M\ n\ u\ w_1))$, and `pullM` : $((M\ n\ t\ w_0) \times^1 (M\ n\ u\ w_1)) \multimap (M\ n\ (t \times^1 u)\ (w_0 *^1 w_1))$. For the ease of understanding, in Granule syntax, these are `(a, b) [c] → (a [c], b [c])` (for `push`) and `(a [c], b [c]) → (a, b) [c]` (for `pull`) [6]. Essentially, these two functions allow us to "distribute" a modality over a linear pair, or invert this "distribution".

The definitions (in Idris) are as follows

```
pushM MZ = MZ # MZ
pushM (MS (x # y) z) =
        let (xs # ys) = pushM z
        in (MS x {w=x} xs # MS y {w=y} ys)
```

and

```
pullM (MZ # MZ) = MZ
pullM (MS x xs # MS y ys) =
        MS (x # y) {w=(x # y)} (pullM (xs # ys))
```

Note the simultaneous mapping of the values at runtime and the witnesses at type-level, in addition, note that we manipulate the witnesses like what they represent at runtime.

Naturally, these should form an isomorphism.

Lemma 2.4. `pushM` *and* `pullM`*, form a linear isomorphism between* $(M\ n\ (t \times^1 u)\ (w_0 *^1 w_1))$ *and* $((M\ n\ t\ w_0) \times^1 (M\ n\ u\ w_1))$

Proof. It is trivial that the types are indeed those of their respective functions. Then, because the argument type of `pushM` is the result type of `pullM`, and vice versa, we can use 2.2 to determine that this must map to and from the same object (there is only one possible) □

2.3 Mapping Linearly

Next, we define a way to define a `Functor` like `map` on M, namely, `mapM` : $(f : t \multimap u) \to M\ n\ t\ w \multimap M\ n\ u\ (f\ w)$ Idris definition

2.3.1 Applying M to M

Now, for the ability to apply M types. Firstly, note that in general, in both QTT and GrTT, we need to have an equal number of functions and arguments [3, 6] We can construct a function `applyLinear` : $(a \multimap b) \multimap a \multimap b$, whose term is just `applyLinear`$fx \equiv fx$. We can then curry this function, getting `applyPair` : $((a \multimap b) \times^1 a) \multimap b$.

Cite and maybe linear currying?

In addition two values $x : Mn(t \multimap u)f$ and $y : Mntw$, we can then construct the linear pair, $(x *^1 y) : Mn(t \multimap u)f \times^1 Mntw$. We can then apply `pullM` to the pair, thereby getting a value of the form $z : Mn(t \multimap u \times^1 t)(f *^1 w)$, and we can then use `mapM applyPair` to get $z : Mnu($`applyPair`$(f *^1 w))$. Simplify, and we get $z : Mnu(fw)$

2.4 combining and Spiting

If we view Mas "vectors of linear values", then we should have an equivalent to vector concactination, `++` [4]. These correspond with the GrTT notion of context combination [6]. For M, we call these $\otimes$, (in Idris `combine`)

It is defined as follows:

```
combine :
        forall t, n, m.
        {0 w : t} ->
        M n t w -@
        M m t w -@
        M (n + m) t w
combine {w} MZ ys = ys
combine {w=x} (MS x xs) ys = MS x {w=x} (combine xs ys)
```

This simply combines all the elements in a similar way to a list. In addition, we also have the inverse of `combine`, `split`[1]:

```
split' : forall t, w, n. {1 m : Nat} -> M (m + n) t w -@ LPair (M
↪  m t w) (M n t w)
split' {m=Z} xs = MZ # xs
split' {m=S m} (MS x xs) = let (ys # zs) = split' {m=m} ( xs) in
↪  (MS x {w=x} ys # zs)
```

Lemma 2.5. *split combine inverses*

Finish

2.5 Squashing and Expanding

Write section

2.6 Associativity, Commutativity

Write section

[1] `split'` is described here. `split` essentially leaves the proof that the source does indeed have a index of $n + m$ up to the user with an `auto` implicit

2.7 Conversing Resources

Write section

3 Linear Exponential Types

While M types are themselves interesting, they are not, by themselves, very useful. This is because they don't actually serve as a model for graded modalities (as they appear in Granule) [6]. These require a way to abstract *only* on the type and multiplicity, but *not* to require a given instance.

For this, we need to extend the type system with linear exponential types, written a^n, which allow us to abstract over the witness

3.1 Linear Existentialal Types

First, however, we must define a Linear Existential Type. For a dependent function ($\prod$), with Idris 2's three multiplicities, there are three functions, one for each multiplicity. For a product, however, there are *9* types, one for each *pair* of multiplicities.

The one we need is not one of the many defined in `prelude`, `base`, and `linear`. What we need is a *linear existential*, that is, a dependent product type where the first arguement is erased, and the second depends on the first. We can define such a thing in Idris as:

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

This is very much like the `base`'s `Exists`. However, that is unrestricted on its second field, which is most certainly not what we want.

3.2 Construction of Exponentials

We define the exponent, written explicitly as $\wedge$ (both here and in the source), as being a function on types. Specifically, it is erased (type level function) that takes in a natural number, and a type, and returns a type, and is of form

```
0 (^) : (t : Type) -> (n : Nat) -> Type
(^) t n = LExists (M n t)
```

After binding application [8] is implemented, the will look like

```
0 (^) : (t : Type) -> (n : Nat) -> Type
(^) t n = LExists (w : t) | (M n t w)
```

That is, it is a witness of a given type at the *value* level and a given value with the witness at the *type* level. The important distinction between the exponent type and M type is that M types are indexed over their witness. This is a very useful property, indeed, this is 2.2. However, for a modality, this isn't true, $x : t[2]$ and $y : t[2]$ have the same type, no mater what their actual values are.

Probably should be more here

3.3 From M to Exponentials and Back

As mentioned previously, exponentials don't fundamentally include any different information then M types. They merely contain it with a differing place hence differing variance, M at the type level, $\wedge$ at the value level. Moreover, the type level information in M corresponds to the erased $\wedge$ level information (the first element of the pair).

Because Idris 2 is also dependently typed, this makes the transformation from term level erased information and type relevant[2] information trivial, we just have to extract the specific values. We define `inType` (for going to M) and `inValue` (for going to $\wedge$) as follows:

```
inType : {0 a : Type} -> (1 ex : (a ^ n)) -> M n a ex.fst
inType {a=a} ex = sndL ex

inValue : {0 a : Type} -> {0 w : a} -> M n a w -@ (a ^ n)
inValue {a} {w} x = LEvidence {fst'=w} {snd'=x}
```

These two operations are inverses of each other .

Maybe prove this?

Because of this, any function that takes or gives one of these types can be made to take or give the other. So, for instance, to go from $f : Mmtw_0 \multimap Mnuv_0$ to $g : t^m \multimap u^n$, all we must do is $g \equiv \texttt{inValue}.f.\texttt{inType}$

3.4 The Numerical Properities of Exponentitation

We can now get to the linear exponentails numeric like properites. Note that much of what is descirbed here is discussed at length by Marshall and Orchard.

We already have a couple useful properties. Firstly, for this discussion, we think of a^n like the exponent operator of arithmetic and $a \otimes b$ as the multiplication operator. To go over laws we have already shown:

Enumerate the laws

3.5 Repeated Application

4 ω and Ω Types

While all this is useful, we still need to be able to incorperate the idea of "unrestricted" bindings into our set of constructs. This is the goal of the ω and Ω constructions, to provide model for almost all multiplicities, without extending the core language.

4.1 ω

The ω construction is quite simple. It takes advantage of the fact that there isn't a distinction between a value and a way to generate it. So, we don't need

[2] But still ultimitly runtime erased

an actual variable number of bindings, we merely need a way to generate for any n some n bindings. This is all ω is, a continuation on the number of bindings.

So, $\omega tw \equiv ((n : \mathbb{N}) \to Mntw)$ add more info on this

4.2 ω as $!_*$

4.3 Ω as a generalization of ω

4.4 Ω as a generalization of M

4.5 M to Ω

5 Using M and Friends

5.1 Fully General Product Types

5.2 Graded Modalities

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

A Figures

List of Listings

Glossary

$=_?$ Propositional (type-level) equality 3, 4

$\diamond$ All copies have been exhausted 3

$\forall$ Erased function, or equivalently universal quantitfication, ($\forall_a bc$ is equivalent to $b :_0 a \to c$) 3

$\mathcal{L}$ The category of linear values The linear mapping, written in Idris as `-@` 3–5, 7.

$\odot$ Provide a single instance for the copy 3

`base` The Idris 2 base library[1] 6

`linear` The Idris 2 linear library[4] 6

`prelude` The Idris 2 Prelude[7] 6

GrTT Graded Modal Type Theory 2, 5

Ω The Omega Ω type, which is a certain number of copies of a type, combined with a witness for that, that is used to model multiplicities 2, 3, 7, 8

M The Mu (M) type, which is a certain number of copies of a type, combined with a witness for that, that is used to model multiplicities 1–5

$\mathbb{N}$ The set of all natural numbers 3, 8

QTT Quantative Type Theory 2, 5

$*$ The type of all types, also called `Set`, $\mathcal{U}$, or $\mathcal{U}^{\infty}$ 3

References

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

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

[4] *linear*. URL: https://www.idris-lang.org/Idris2/linear/.

[5] 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.

[6] 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.

[7] *prelude*. URL: https://www.idris-lang.org/Idris2/prelude/.

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