# 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 [10, 4, 9].

[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



### 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 $\omega$ 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 $\Pi$ types prefixed with a $\forall$, so, for instance, we have $\forall 1 \to x : t.u$ as the equivalent of the Idris `{1 x : t} -> u`. We write $\times^C$ for the product construction in $C$, and for a given evidence of that construction we write that with the constructor $*^C$

## 2 Background

While linear values are alone quite useful, there are some times we want to "clone" bindings, in the manner of Rust. However, there is a problem: cloning a value creates a different value.

While in a lnaguage like Rust this isn't a problem, with a language like Idris with dependent types that even if $x$ is a clone of $y$, $px$ and $py$ might not unify.

Here, however, we don't use the `linear` libraries `Duplicate`, as it actually is a little to good at generating values that are the same, such that we can't actually use one without also using the other.

Instead, we use one main solution, representation functions, inspired by Ghost Type Theory [13].

*Ghost Types.* One of the slightly more irratiting things

## 3 $M$ 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 3.1.** $M$ is a polymorphic type with signature $M : (m')n : \mathbb{N}) \to_{(S}(t : *) \to_0 (w : t) \to_0 *$

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 $\odot : (w : t) \to_1 M\ n\ t\ w \to_1 M\ (Sn)\ t\ w$

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* 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, 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` $: M\ 0\ t\ w \to \top$, 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` $: M\ 1\ t\ w \to_1 t$, 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 $x : M(Sn')tw$, 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 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 3.3 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, $x_1 =_? y_1$

```
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 3.5 in Idris

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

Proof. We know by 3.5 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.

### 3.2 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 [12]. 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 `dropwhere` the exact relationship is shown in **??**.

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

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

```
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

$$\diamond \otimes x := x \tag{1}$$
$$Z + x := x \tag{2}$$
$$(a \odot b) \otimes x := a \odot (b \otimes x) \tag{3}$$
$$(Sn) + x := S(n + x) \tag{4}$$

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.

### 3.3 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 3.7, 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 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 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 $^\wedge$types.

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

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

Add something on $M$ metalogical interpretation

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".

[2]You can also prove associativity and related properties, the proof is the same
[3]Given that we have $uncurry^1 : (a \to_1 b \to_1 c) \to (a \times^1 b) \to_1 c$

```
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 3. Definition of map and app

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

$\texttt{join}\diamond := \diamond$

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

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 $M$

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.

So, we define another function, app, which has the type $\mathsf{app} : (f : M\ n\ (t \to_1 u)\ w_f) \to_1 (x : M\ n\ t\ w_x) \to_1 M$ and a defintion given at 3

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

This isn't exactly correct

### 3.5 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 $\texttt{push} : M\ n\ (t \times_1 u)\ (w_0 *_1 w_1) \to_1 M\ n\ t\ w_0 \times_1 M\ n\ u\ w_1$ and $\texttt{pull} : M\ n\ t\ w_0 \times_1 M\ n\ u\ w_1 \to_1 M\ n\ (t \times_1 u)\ (w_0 *_1 w_1)$. Their construction is simple, and may be found in $\texttt{idris} - \texttt{mult8}$.

## 4 Resource Algebras as Types

In the languages of Flix, Koka, and others, the type system is enriched with an effect algebra [10, 9]. In a similar vein, Granule (and Idris to a much more limited extent) use a resource algebra to enrich the type system.

In a language like Haskell however, there has been a question as to whether this can be embed into the language itself. This is the goal of the libraries such as $\texttt{fusted} - \texttt{effect}$ Need citation, which use the constraint and type polymorphism to form a limited algebra.

Check name

In Idris, we can do similar constructions. However, it remains a question as to whether we can construct a type of resource algbebras to enrich Idris types with multiplicitity polymorphism

[4]For simplicity, we infer the second witness

Here, we propose a "simple" system of effect formulas that model functions on natural numbers. We view each possible multiplicity as a "member of" solution set. So, for instance, to get all possible multiplicities of two, we would use the model of $\lambda x.2 * x$.

The problem with this, however, is that solving functions themselves are not decidable.

### 4.1 Formula Types

So, we create a "restricted" effect formula type, called `Form` which represents a function from natural numbers to natural numbers. We formalize this notion of "modeling a given function" by defining a function `Eval` $:_0$ `Form` $\to \mathbb{N} \to \mathbb{N}$. This is the core means through which we interpret a given formula.

In addition, we have two further procedures, `Solve`$\phi x$. Its definition is `Solve`$f y :=_0$ (`Exists`$\mathbb{N}(\lambda x.($`Eval`$f x = y)))$. We further define `Unify`$\phi\psi$, which takes in two formulas and retursn a type, and is defined as $\forall(y : \mathbb{N}) \to y \in \phi \to y \in \psi$

*Notation* 4.1. We write `Solve`$\phi x$ as $x \in \phi$ and `Unify`$\phi\psi$ as $\phi \sqsubseteq \psi$

Intuitively, $y \in \phi$ is "$\phi$ can output $x$, and $\phi \sqsubseteq \psi$ is that if $y$ is a solution of $\phi$ it is also a solution of $\psi$ [12].

Note that $\sqsubseteq$ forms a pre-order on `Form`, that is, it is transitive and reflexive.

**Lemma 4.2.** *$\sqsubseteq$ is reflexive, that is,* $\forall(\phi : \texttt{Form}) \to \phi \sqsubseteq \phi$

Proof. Upon expansion, the result type becomes $\forall(y : \mathbb{N}) \to$ `Eval`$\phi y \to$ `Eval`$\phi y$, which is just identity □

**Lemma 4.3.** *$\sqsubseteq$ is transitive, that is,* $\forall(\phi_0 : \texttt{Form}) \to \forall(\phi_1 : \texttt{Form}) \to \forall(\phi_2 : \texttt{Form}) \to \phi_0 \sqsubseteq \phi_1 \to \phi_1 \sqsubseteq \phi_2$

Proof. □ Proof

### 4.2 The Simple Forms

The most basic constructor of the `Form` type is that of `FVar`, which is a nullary constructor of `Form`. It models the "arguement" in a given formula, and its clause of `Eval` is `EvalFVar`$x := x$.

Note that `FVar` is more general than any other formula, as every single natural number is mapped to, so we have $\phi \sqsubseteq$ `FVar` for each formula $\phi$

Add proof of this

Next up we have nearly as simple of a constructor, `FLit`. It is a unary constructor taking a natural number, which is essientally the "constant" formula. Its branch of `Eval` is `Eval(FLit`$k)x := k$, that is, it ignores its arguement.

### 4.3 Binary Operations

We introduce four constructors of `Form`, all of which are quite similar, all taking in two sub-formulas as arguemetns to the constructors, and all model binary operations.

They are `FAdd` (addition), `FMul` (multiplication), `FMax` (joins), `FMin` (meets). The first two of these are sufficent to define `Form` as an instance of the Idris `Num` class, with the conversion being `FLit`, addition as `FAdd` and multiplication as `FMul`.

We then define the evaluation of all of these. Using `FAdd` as an example, we have `Eval(FAdd`$f g)x :=$ (`Eval`$f x$) + (`Eval`$g x$), and we omit here the rest of the operators because they are quite similar.

The only notable fact about such of these is that they both use the same variable for both of the formulas, so `FAddFVarFVar` is a formula which the same variable is used twice. every instance of `FVar` in $\phi$ with $\phi$

### 4.4 The Extensions

While so far we have defined `Form` to be a rig (ring sans negation), the rest of these forms all do something fairly unique. First among these is `FApp`, which is a binary constructor of a `Form`. Very roughly, $\texttt{FApp}\phi\psi$ represents the "composition" of the formulas. While this sounds complex, it is in praticed quite simple, with us defining the `Eval` branch as $\texttt{Eval}(\texttt{FApp}\phi\psi)x := \texttt{Eval}\phi(\texttt{Eval}\psi x)$.

Apart from being viewed as a composition, we can also view $\texttt{FApp}\phi\psi$ as the result of substituting

**Lemma 4.4.** *`FVar` is the identity formula with respect to `FApp`*

Proof

Proof. □

We also note compostions always are subformulas of their second formula

**Theorem 4.5.** *For any formulas $\phi$ $\psi$, $\texttt{FApp}\phi\psi$ is at most general as $\phi$*

Proof

Proof. □

### 4.5 Completeness of Natural Numbers

There are two much more out of place constructors, `FLeft` and `FRight`, both of which are unary constructors on `Form`. These constructors have a seemingly random definition, but have a very important for allowing polyvariadic formulas.

Firstly, we define a function, `pairing`, which has the form $\mathbb{N} \rightarrow_1 \mathbb{N} \times \mathbb{N}$ it is defined as follows:

- `pairing` 0 is $(0, 0)$
- If $n$ is not of the prime factorization form $2^x 3^y$, then it is equal to $(0, 0)$
- If, for some $x, y : \mathbb{N}$, it is of the form $2^x 3^y$, then it is equal to $(x, y)$

**Corollary 4.6.** *For all $x, y : \mathbb{N}$, $\texttt{pairing}(2^x 3^y) =_? (x, y)$*

Finish

Proof. □

Finish this section

## 5 Ω Types

The principle use of `Form` here is as part of the Ω construction, which uses it to constraint over mutliplicity, thereby giving us an equivalent of Granuleeffect formulas,

### 5.1 Poly-multiplicative Judgments

**Definition 5.1.** Ω is a type indexed by a formula, $\phi$, a type $t$, and a witness of that type, such that $\Omega\ \phi\ t\ w :=_0 (n : \mathbb{N}) \rightarrow_1 M\ (\texttt{Eval}\ \phi\ n)\ t\ w$

That is, an Ω designates a function from a natural number to a certain number of $t$. However, this is often unedsierable, so we instead efine a helper function, reify, of the type $\Omega\ \phi\ t\ w \rightarrow_1 \forall (n : \mathbb{N}) \rightarrow_1 \forall (n \in \phi) \Rightarrow$ which, allows for us to instead consider if the number in is in the "solution set" as opposed to a value projected to.

The most simple form of Ω is that where $\phi$ is `FVar` and thus is simply, $(n : \mathbb{N}) \rightarrow_1 M\ (\texttt{Eval FVar}\ n)\ t\ w$, or simply $(n : \mathbb{N}) \rightarrow_1 M\ n\ t\ w$, which we specifically call $\omega t w$, and $\omega t w :=_0 (n : \mathbb{N}) \rightarrow_1 M\ n\ t\ w$

## 6 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 3.2, 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

```
data Form : Type where
FVar : Form
FVal : (1 n : QNat) -> Form
FApp : (1 g : Form) -> (1 f : Form) ->
↪  Form
FAdd : (1 x : Form) -> (1 y : Form) ->
↪  Form
FMul : (1 x : Form) -> (1 y : Form) ->
↪  Form
FMin : (1 x : Form) -> (1 y : Form) ->
↪  Form
FMax : (1 x : Form) -> (1 y : Form) ->
↪  Form
FLeft : (1 f : Form) -> Form
FRight : (1 f : Form) -> Form

0 Eval : (1 f : Form) -> (1 x : QNat)
↪  -> QNat
```

```
Eval FVar x = x
Eval (FVal n) x = n
Eval (FApp g f) x = Eval g (Eval f x)
Eval (FAdd f g) x = ladd (Eval f x)
↪  (Eval g x)
Eval (FMul f g) x = lmul (Eval f x)
↪  (Eval g x)
Eval (FMin f g) x = lmin (Eval f x)
↪  (Eval g x)
Eval (FMax f g) x = lmax (Eval f x)
↪  (Eval g x)
Eval (FLeft f) x = let
        (y # z) = pairing x
        in Eval f y
Eval (FRight f) x = let
        (y # z) = pairing x
        in Eval f z
```

Listing 4. The definition of formulas and evaluation in `idris` – `mult8`

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

Listing 5. The definition of `LExists`



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 $\omega$, 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].

## 6.1 Existential Crisis (Solution)

This principle issue with using $M$ and $\Omega$ 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

## 7 Using $M$ and Friends

## 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 infered and checked is important. However, in terms of QTT (or even systems outside of 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 seperate 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[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/.

[6]Some listings may be modified for readability

[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] 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] Théo Winterhalter. “Dependent Ghosts Have a Reflection for Free”. In: *Proc. ACM Program. Lang.* 8.ICFP (Aug. 2024). doi: 10.1145/3674647. url: https://doi.org/10.1145/3674647.