Linear Types with Dynamic Multiplicities in Dependent Type Theory (Functional Pearl)

MAXIMILIAN DORÉ, University of Oxford, United Kingdom

We construct a linear type system inside dependent type theory. For this, we equip the output type of a program with a bag containing copies of each of the input variables. We call the number of copies of an input variable its multiplicity. We then characterise a dependent type which ensures that a program uses exactly the given multiplicity of each input variable. While our system is not closed under linear function types, we can program in the resulting system in a practical way using usual dependent functions, which we demonstrate by constructing standard programs on lists such as folds, unfolds and sorting algorithms. Since our linear type system is deeply embedded in a functional language, we can moreover dynamically compute multiplicities, which allows us to capture that a program uses a varying number of copies of some input depending on the other inputs. We can thereby give precise types to many functional programs that cannot be typed in systems with static multiplicities.

CCS Concepts: • **Theory of computation** → **Type theory**; **Linear logic**; *Logic and verification*; *Programming logic*.

Additional Key Words and Phrases: Linear Dependent Type Theory, Quantitative Type Theory, Graded Types

ACM Reference Format:
Maximilian Doré. 2025. Linear Types with Dynamic Multiplicities in Dependent Type Theory (Functional Pearl). *Proc. ACM Program. Lang.* 9, ICFP, Article 262 (August 2025), 21 pages. https://doi.org/10.1145/3747531

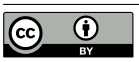
1 Introduction

Linear type systems allow for specifying in the type of a program how often it uses each variable. This gives crucial information to both the programmer, who wants to reason about the correctness of their program, and the compiler, which tries to understand the usage of resource to find an efficient way to run a program. Recent work on integrating linear type systems in functional languages such as Linear Haskell [3], Quantitative Type Theory [1] and Graded Modal Type Theory [21] restrict the typing rules by annotating variables with elements of some sort of resource algebra. Thereby, any variable comes with a *multiplicity* which says how often that variable will be used. Crucially, the multiplicity of each variable needs to be known statically, which makes it impossible to precisely type many programs that occur naturally when programming functionally. Consider, e.g., a safe head function for lists defined as follows in Haskell.

```
safeHead :: [a] -> a -> (a,[a])
safeHead []     y = (y,[])
safeHead (x:xs) y = (x,xs)
```

The function aims to extract the head of the list and, if the given list is empty, resorts to returning the backup element y (it also returns the tail of the list as the second element in the pair). The

Author's Contact Information: Maximilian Doré, University of Oxford, Oxford, United Kingdom, maximilian.dore@cs.ox.ac.uk.



ACM 2475-1421/2025/8-ART262
https://doi.org/10.1145/3747531

element y is therefore used once or not at all, depending on the given list. We cannot express this use of y in either of the systems mentioned above since they require a static multiplicity for y.[1]

We present a solution to this conundrum by deeply embedding a linear type system in a functional language, as opposed to changing the typing rules of the language itself. This makes it possible to dynamically compute the multiplicity of a variable. The functional language we use is dependent type theory [17], which adds predicates to the type system. We deeply embed the rules of linear logic in the host type theory using finite multisets, which can be defined as a higher inductive type [25] in Cubical Agda [28], and then define a dependent type which we understand as a *linear judgment*, i.e., as the statement that some program uses exactly the given resources. We thereby obtain a linear type system inside dependent type theory, and this system ensures that any program whose output type is a linear judgment uses exactly the provided resources.

What is missing from our type system are linear function types, which means we cannot represent linear higher-order functions, in contrast to other systems [1, 21]. We get around this by using the function type of the host dependent type theory. This has limitations, e.g., we cannot capture how often some function variable is used by a program, but we can nevertheless give a precise type to many common functional programs. The programs themselves are implemented as usual in a functional language, we only have to construct linear functions while we implement a program to convince the type-checker that it uses the provided resources. We show how data types can be accommodated in our type system and derive folds and unfolds for lists, which makes it straightforward to construct more complex programs such as sorting algorithms in a linear fashion.

In summary, this paper works out the—in the eyes of the author surprising—finding that in order to make a dependently typed language linear, we do not have to change its rules, but can just deeply embed the rules of linear logic. Not only allows this approach to easily add linear types to an existing language, it also enables us to represent forms of variable use that we could not express if we had equipped the structural rules of type theory with static resource annotations.

Outline of the construction. To make a program of type $(x_1 : A_1) \to \cdots \to (x_n : A_n) \to B$ linear, we equip the output type B with a bag of values Δ, which we call a *supply*. A supply gathers the resources that we can and must use in our program. If Δ contains m copies of variable x_i, we say that m is the multiplicity of x_i. We define a dependent type $\Delta \Vdash B$ which we read as a *linear judgement*. Inhabitants of this type are pairs of some $b : B$ as well as a witness that b is made up of the resources in Δ. We therefore aim to construct programs of the following type.

$$(x_1 : A_1) \to \cdots \to (x_n : A_n) \to \Delta \Vdash B$$

Any program with the above type is guaranteed to consume precisely the resources Δ. For example, we can construct a program with the following type in our system.

$$\mathsf{safeHead} : (xs : \mathsf{List}\ A) \to (y : A) \to (\mathsf{if}\ \mathsf{null}\ xs\ \mathsf{then}\ \iota\ y\ \mathsf{else}\ \diamond) \otimes \iota\ xs \Vdash A \times \mathsf{List}\ A$$

We write $\iota\ \_$ for the supply containing a single value, $\diamond$ for the empty supply, and $\_\otimes\_$ for the union of two supplies. Given an input list xs and a backup element y, the supply in our linear judgment contains the input list xs, and y only in case $\mathsf{null}\ xs$ evaluates to true. We will construct $\mathsf{safeHead}$ by pattern matching on xs, which specializes the supply given above to contain a copy of y only in case xs is the empty list. Our type system hence offers us a very precise language to equip programs with resource annotations.

[1] Some of the theories have *affine* types to give bounds for how often some variable is used, e.g., in Granule [21], which implements graded modal type theory, the backup element is annotated $[0..1]$ to capture that it is used at most once.

All of the examples in this paper have been implemented in the Cubical extension [28] of Agda [20] and can be further explored in the accompanying artefact[2]. We make use of a higher inductive type [25] in our construction, namely finite multisets, sometimes also called bags. While finite multisets are not necessary to add our construction to a dependently typed language—we will explain in Section 6.1 how this can be done in the absence of higher inductive types—it gives us a lot of the necessary structure for our linear type system. We only need to extend our notion of what it means for two supplies to contain the same resources in order to incorporate data types.

Background. The idea at the heart of this paper, namely the equipment of the codomain of a dependent function type with a bag of resources is due to Pédrot [22, 23]. He characterises the Dialectica construction [29] for dependent types as giving the "higher-order dependent graded type of a term", and observes that this allows for dynamically computing multiplicities. We show that this view on linear logic and type theory leads to a practically useful functional programming language, and contribute to the understanding of the Dialectica translation for positive types. Crucial for our system is also the finding that in a linear dependent type theory, we ought to ignore the usage of variables in types, which has been put forward by McBride [18]. Our dynamic multiplicities are reminiscent of the *index terms* of Dal Lago and Gaboardi [8], who equip PCF [24] with a linear dependent type system to capture the complexity of a functional program in its type.

Structure. We start out in Section 2 by giving the basic setup for our linear type system. We define supplies as heterogeneous bags, give a data type of *productions* which captures when two supplies present the same collection of resources, and define our linear judgment $\Delta \Vdash A$.

We use our framework to program with lists in Section 3. We derive linear folds and unfolds for lists and use these to construct list functions in a standard way. In particular, we can devise insertion and selection sort programs whose types ensure that they are linear and therefore do not drop or duplicate any of the input elements.

We then proceed to exploit the full power of dynamically computed multiplicities in Section 3. We derive a fold and an unfold for lists which use resources dynamically, and derive standard list functions such as intersperse and map as instances of these. We show how we can add an infinite multiplicity to our system, which can be used to construct infinite lists. In Section 5, we demonstrate how our linear type system straightforwardly allows us to program with dependent types such as vectors.

We compare our system with other approaches to dependent linear type theory in Section 6 and close with a discussion of future lines of work in Section 7.

2 Equipping types with bags of resources

We carry out our constructions in the Cubical extension of Agda. We write Type for the type of all types[3] and $a : A$ if a is an element of type A. In case B is a type depending on A, we write $(x : A) \to B\ x$ for the dependent function type, and $\Sigma\ A\ B$ for the dependent pair type of A and B. In case B does not depend on A, we write $A \times B$ for the pair type. Elements of the pair type are written $(a\ ,\ b)$.

Cubical Agda implements *higher inductive types* [25], which allow us to introduce equalities in the definition of some inductive type X. More precisely, the target of a constructor of X can be an equality $x \equiv y$ for some $x, y : X$. We will use a higher inductive type to define *bags*, which are lists

[2]See the zip folder accompanying this submission. If this paper is accepted, we will provide a browsable online artefact. Note also that the Agda code in this paper linked, which allows for quickly going to the definition of some expression.
[3]We omit universe levels in the paper to ease readability, but use them in the accompanying artefact.

whose order of elements does not matter (we will explain in Section 6.1 how we can carry out our construction in a dependent type theory without higher inductive types).

2.1 Supplies as bags of values

We gather resources that a program must use in a *bag*, also called finite multiset. In Cubical Agda, we can use a higher inductive type to define bags as lists which are quotiented under reordering.

```
data Bag (A : Type) : Type where
  ◇ : Bag A
  _;_  : A → Bag A → Bag A
  comm : (x y : A) (xs : Bag A) → x ; y ; xs ≡ y ; x ; xs
  trunc : isSet (Bag A)
```

We always have the empty bag ◇ and can prepend elements to a bag with the ; constructor. The higher constructor comm (short for commutative) says that we consider two bags equal if they contain the same two elements in front, but in different order. This constructor can be applied repeatedly to obtain any permutation between two lists of elements. In general, there are many such permutations, the trunc constructor (short for truncation) establishes that we consider all such permutations the same—in other words, we only care about the existence of a permutation. In the lingo of homotopy type theory [25], if all equalities between elements of some type are all equal to one another, this type is a *set*, which explains the name of the predicate isSet. The important point for us is that bags behave just like lists—except that their order of elements does not matter—and possess no higher homotopical structure that we have to worry about.

We can define a function to append one bag to another by recursion on the first bag, just like we would with lists. We just give the type signature here and refer the interested reader to Choudhury and Fiore [7] for further study of bags and an explanation of how to define such a function between higher inductive types.

```
_⊗_ : Bag A → Bag A → Bag A
```

We want to put elements of all types in a single bag. For this, we introduce a type that any value can be made an inhabitant of.[4]

```
Value : Type
Value = Σ[ A ∈ Type ] A
```

A Value is hence a dependent pair of some type A and an inhabitant of A. With this, we have all necessary things at hand to define heterogeneous bags, which act as our notion of resource and will be called *supplies* in the following.

```
Supply : Type
Supply = Bag Value
```

We introduce a function to create the unit supply for a given value a of type A, which simply puts the pair (A , a) in front of the empty bag.

```
ι : A → Supply
ι a = (A , a) ; ◇
```

In the following, we construct supplies using only the unit ι and append $\otimes$ operators, for example, given $a : A$ and $b : B$ we might consider the supply $\iota\ a \otimes \iota\ a \otimes \iota\ b$ of two copies of a and one copy of b.

[4] In the type-theoretic literature, Value is usually called a *pointed type*.

We will use Δ, Δ_0, Δ_1 etc. as names for supplies in the following. The greek letter Δ is usually used for contexts in the literature, but this clash is intended: the elements of a supply that we care about are variables from the host type theory, and the number of times a variable is part of some supply specifies how often we can use this variable.

2.2 Repacking bags

Supplies already satisfy a lot of properties that we would expect from a collection of resources: adding the empty supply $\diamond$ to a supply Δ yields the same supply Δ, we do not care about the order of elements in a supply (i.e., $\otimes$ is commutative), and we do not care about the brackets around sequences of appended supplies (i.e., $\otimes$ is associative). These properties do not need to be postulated as constructors of the Bag type, but can be derived by induction. We refer to the Cubical library[5] for proofs of these properties. However, we need to introduce additional ways in which two supplies are considered to contain the same resources to take into account values of some data type. For example, we want to consider a supply consisting of a pair $\iota\ (a\ ,\ b)$ to be representing the same resources as the supply $\iota\ a \otimes \iota\ b$.

In order to do this, we introduce a data type that captures all the ways in which we consider two supplies to contain the same resources. We call this the type of *productions*, as it allows us to produce one supply from another supply.

```
data _⊸_ : Supply → Supply → Type where
  id : ∀ Δ → Δ ⊸ Δ
  _∘_ : ∀ {Δ₀ Δ₁ Δ₂} → Δ₁ ⊸ Δ₂ → Δ₀ ⊸ Δ₁ → Δ₀ ⊸ Δ₂
  _⊗ᶠ_ : ∀ {Δ₀ Δ₁ Δ₂ Δ₃} → Δ₀ ⊸ Δ₁ → Δ₂ ⊸ Δ₃ → (Δ₀ ⊗ Δ₂) ⊸ (Δ₁ ⊗ Δ₃)
```

The identity production id establishes that any supply can be turned into itself, while the composition production $\circ$ closes productions under transitivity. The constructor $\otimes^f$ acts as a sort of congruence closure of productions with respect to $\otimes$.[6]

The attentive reader will have noticed that there is no symmetry production in our definition. One might conceivably want to add productions that drop resources (to introduce for example additive conjunction), which is why we do not in general make $\multimap$ symmetric. For the developments in this paper, however, $\multimap$ will always be symmetric. We can establish this using swap below, which in turn follows from commutativity of $\otimes$ (any equality between supplies can be turned into a production by substituting one side of the id production). Similarly, we can derive several other structural productions such as the right unital law for $\diamond$.

```
swap : (Δ₀ Δ₁ : Supply) → (Δ₀ ⊗ Δ₁) ⊸ (Δ₁ ⊗ Δ₀)
unitr : (Δ : Supply) → (Δ ⊗ ◇) ⊸ Δ
assoc' : (Δ₀ Δ₁ Δ₂ : Supply) → Δ₀ ⊗ (Δ₁ ⊗ Δ₂) ⊸ (Δ₀ ⊗ Δ₁) ⊗ Δ₂
```

We will extend our type of productions $\multimap$ whenever we want to include a data type in our linear type system. More precisely, we will state rules that allow us to introduce and discard any constructor of this type as necessary, while keeping all resources contained in the respective constructor.

For the dependent pair type, which has a single constructor (_ , _), we introduce productions to capture that having a pair or having both of its elements separately constitutes the same collection of resources. We hence introduce the following productions, for any given types A, B dependent on A and inhabitants $a : A$ and $b : B\ a$.

[5] https://github.com/agda/cubical

[6] More precisely, $\otimes^f$ is the functorial action of $\otimes$ considered as a bifunctor, which explains the superscript in our notation. We spell out this perspective in Section 6.1.

```
data _⊸₀_ : Supply → Supply → Type where
  ...
  opl, : ι (a , b) ⊸₀ (ι a ⊗ ι b)
  lax, : (ι a ⊗ ι b) ⊸₀ ι (a , b)
```

The names stand for oplax and lax, intuition behind these names will be given in Section 6.1.

2.3 The crux: keeping a packing list

With supplies and productions we have all necessary structure to define a *linear judgment* in our system. Saying that something of type A can be constructed using a supply Δ amounts to giving an inhabitant a of A, as well as a recipe which says how a can be obtained using the resources in Δ. This is exactly the following dependent pair type.

```
_⊩_ : Supply → Type → Type
Δ ⊩ A = Σ[ a ∈ A ] (Δ ⊸₀ ι a)
```

When constructing a program with a linear type, we hence have to do two things: we have to construct the output value, and we have to prove that this value is made up of the given resources.

We now introduce several functions which make it convenient to program in our linear type system. The first rule can be considered an analogue to the variable rule for linear logic (but for arbitrary values of the host theory), and states that an $a : A$ is precisely what we need to construct something of type A.

```
. : (a : A) → ι a ⊩ A
. a = a , id (ι a)
```

We resolve two tasks in the definition of . above: we give a witness of type A, which is just the given a, and a recipe how to turn $\iota\ a$ into $\iota\ a$, which is just the identity production.

The second rule that we often use takes a linear judgment over supply Δ_0 and a production from another supply Δ_1 to Δ_0, and yields a linear judgment over Δ_1. In fact, this is just applying composition of productions.

```
_by_ : Δ₀ ⊩ A → (Δ₁ ⊸₀ Δ₀) → Δ₁ ⊩ A
(a , δ₂) by δ₁ = a , δ₂ ∘ δ₁
```

With these two definitions we can conveniently construct linear programs. For example, we can write a function which swaps around the two elements of a non-dependent pair.

```
switch : (z : A × B) → ι z ⊩ B × A
switch (x , y) = . (y , x) by prod where prod : ι (x , y) ⊸₀ ι (y , x)
                                         prod = lax, ∘ swap (ι x) (ι y) ∘ opl,
```

In the definition of switch we pattern match on the given element, which means that the type we need to construct is specialised to $\iota\ (x\ ,\ y) \Vdash B \times A$. By using the basic variable rule, we give the pair $(y\ ,\ x)$ as our element of $B \times A$. What remains to show is that the resources given by $\iota\ (y\ ,\ x)$ are the same as the ones give by $\iota\ (x\ ,\ y)$. For this, we give the packing list which first discards the pair constructor, swaps the elements, and then introduces the pair constructor again.

2.4 Linear functions

The function switch is linear as it consumes the given pair exactly once, we will in this section introduce definitions which make this linearity obvious in the type signature. First, we introduce the following abbreviation for functions which consume their input exactly once.

```
_⊸_ : Type → Type → Type
A ⊸ B = (x : A) → ι x ⊩ B
```

A linear function from A to B is a dependent function from A to $\iota\ x \Vdash B$, i.e., it takes any $x : A$ to the linear judgment that $\iota\ x$ is a supply from which we can produce some B. Using this abbreviation, we can write the type of switch from Section 2.3 simply as $A \times B \multimap B \times A$.

We can apply linear functions to linear judgments, for which we introduce a linear application operator @. We slightly generalise our considerations and define our application operator for linear functions *with leftovers* [16], which will sometimes appear in our developments below. A function with leftovers is a dependent function that takes $x : A$ to the linear judgment that a B can be produced using one copy of x and some supply Δ_0 (which cannot contain other copies of x). We can apply this function to a value $a : A$ which has been derived using some supply Δ_1. The resulting value of type B will then require the joint resources Δ_0 and Δ_1, as we can readily derive in our system.

```
_@_ : ((x : A) → ι x ⊗ Δ₀ ⊩ B) → Δ₁ ⊩ A → Δ₀ ⊗ Δ₁ ⊩ B
f @ (a , δ) = f a by swap Δ₀ (ι a) ∘ (id Δ₀ ⊗ᶠ δ)
```

We apply the linear function f to the given $a : A$, which yields a linear judgment $\iota\ a \otimes \Delta_0 \Vdash B$. The production δ tells us that we can turn Δ_1 into $\iota\ a$, applying this appropriately using the congruence production $\otimes^f$ and swapping around the two supplies establishes that the application $f\ a$ can be produced using Δ_0 and Δ_1.

We will often need to apply functions in two arguments in the following. Since we cannot represent higher-order linear functions in our system, we only consider their uncurried versions. Given one such function $f : A \times B \multimap C$, we will often want to apply f not directly to two intuitionistic terms A and B, but rather to two linear judgments $\Delta_0 \Vdash A$ and $\Delta_1 \Vdash B$. This will yield an element of C made up of the joint resources of Δ_0 and Δ_1. We can turn any 2-ary function into such a function acting on linear judgments as follows.

```
∘⟶⊗ : A × B ⊸ C → Δ₀ ⊩ A → Δ₁ ⊩ B → Δ₀ ⊗ Δ₁ ⊩ C
∘⟶⊗ f (a , δ₀) (b , δ₁) = f @ . (a , b) by lax, ∘ (δ₀ ⊗ᶠ δ₁)
```

We are given a 2-ary linear function f, a value $a : A$ with production δ_0 that turns Δ_0 into $\iota\ a$, and a value $b : B$ with production δ_1 that turns Δ_1 into $\iota\ b$. We evaluate f at (a , b) considered as a linear judgment, which gives us an element of C using resources (a , b). We use δ_0 and δ_1 and the lax production for the pair constructor to establish that these are precisely the joint resources of Δ_0 and Δ_1.

By applying $\circ\!\!\longrightarrow_\otimes$ to the basic judgment . from Section 2.3, we obtain a more general linear pair constructor that we will often use.

```
_,⊗_ : Δ₀ ⊩ A → Δ₁ ⊩ B → Δ₀ ⊗ Δ₁ ⊩ A × B
_,⊗_ = ∘⟶⊗ .
```

For example, we can define switch from Section 2.3 alternatively as follows, joining together two basic judgements for y and x.

```
switch' : A × B ⊸ B × A
switch' (x , y) = (. y ,⊗ . x) by swap (ι x) (ι y) ∘ opl,
```

3 Linear functional programming with lists

We will now do some functional programming in our linear type system. In Section 3.1, we discuss how to make lists amenable to resource-sensitive programming. We then define a linear fold and unfold in Sections 3.2 and 3.3. These can be derived like the usual fold and unfold for lists, except that we have to do a little bit of work to show that these functions use the given resources. Once this is out of the way, we can construct many list programs as the standard folds and unfolds. In particular, we can derive insertion and selection sort as a fold and unfold, respectively, which we will do in Section 3.4. Devising a sorting algorithm in a linear type systems gives us a crucial part of its correctness proof, namely that the algorithm does not drop or duplicate any of the given elements.

In the following, we use a standard type of lists over A, denoted List A, with constructors [] and $x :: xs$ for given $x : A$ and $xs :$ List A.

3.1 Linearising lists

In order to reason about lists as resources, we need to extend our definition of the production type. Similar to what we did with pairs in Section 2.2, we have to introduce productions for any constructor of our list type to establish that we can freely add or remove it, as long as we keep any resources that were kept inside the constructor. In the case of lists, this means that the empty list is not an actual resource that we need to care about; and that the cons constructor can be added or removed as long as we keep track of the head and tail kept inside a cons constructor.

```
data _⊸₀_ : Supply → Supply → Type where
  ...
  opl[] : (ι []) ⊸₀ ◇
  lax[] : ◇ ⊸₀ (ι [])
  opl:: : ι (a :: as) ⊸₀ (ι a ⊗ ι as)
  lax:: : (ι a ⊗ ι as) ⊸₀ ι (a :: as)
```

These productions give us all we need to program with lists in a linear fashion. For example, we can now implement the safe head function for lists mentioned in Section 1.

```
safeHead : (xs : List A) → (y : A) → (if null xs then ι y else ◇) ⊗ ι xs ⊩ A × List A
safeHead [] y = . (y , []) by lax,
safeHead (x :: xs) y = . (x , xs) by lax, ∘ opl::
```

In case the given list xs is empty, we ought to use the backup element y as well as the empty list. In case the list is not null, we do not have y available anymore, we can hence only return the head and tail of the list, which we turn into a pair by first discarding the cons constructor and then introducing the pair constructor.

In the following, we will often want to treat the list constructors as linear functions. The empty list is simply a 0-ary function over the empty supply.

```
[]₀ : ◇ ⊩ List A
[]₀ = . [] by lax[]
```

The cons constructor for lists is a 2-ary function taking an element of A and a list over A to another list over A.

```
_::₀_ : A × List A ⊸ List A
_::₀_ (x , xs) = . (x :: xs) by lax:: ∘ opl,
```

We can apply the function $_{\circ}\!\!\rightarrow_{\otimes}$ from Section 2.4 to turn the linear cons constructor into a more general version that acts on linear judgments, we will often use this version in the following.

```
_::⊗_ : Δ₀ ⊩ A → Δ₁ ⊩ List A → Δ₀ ⊗ Δ₁ ⊩ List A
```

3.2 Linear folding

There are in general many sensible linear folds for a type that are useful, depending on which resources the different functions corresponding to the constructors use [18]. One possible linear fold of lists of type A into some result type B could require the cons function to use both the current head as well as the result linearly, while giving a distinct supply Δ for the nil function. We expect that this fold yields a linear judgment which uses Δ and the list to be folded, and we can indeed derive this fold in our system.

```
foldr₁ : (A × B ⊸ B) → (Δ ⊩ B) → (xs : List A) → Δ ⊗ ι xs ⊩ B
foldr₁ f z [] = z by unitr Δ ∘ (id Δ) ⊗ᶠ opl[]
foldr₁ f z (x :: xs) = f @ (. x ,⊗ foldr₁ f z xs) by prod where
  prod : Δ ⊗ ι (x :: xs) ⊸ ι x ⊗ Δ ⊗ ι xs
  prod = swap Δ (ι x) ⊗ᶠ (id (ι xs)) ∘ assoc' Δ (ι x) (ι xs) ∘ (id Δ) ⊗ᶠ opl::
```

In the nil case we use the linear judgment z, which uses Δ to produce some B. We only have to remove the empty list from the given supply, which stems from the fact that the type we are currently eliminating into is $\Delta \otimes \iota\ [] \Vdash B$. In case we have a head x, we apply f to x and the recursive call of foldr_1. Note how the cons case of foldr_1 leaves the base supply Δ untouched and passes it down until it can be consumed by the nil case.

Defining our fold required a little work as we had to provide some packing lists, but now that we have defined it, we can conveniently construct linear programs. For example, we can append lists linearly as follows, where we use as the base case for our fold the basic judgment for ys.

```
append : List A × List A ⊸ List A
append (xs , ys) = foldr₁ _::∘_ (. ys) xs by swap (ι xs) (ι ys) ∘ opl,
```

We can also program with nested list types in standard fashion, for example, we can construct a linear concat which uses the given 2-dimensional list exactly once.

```
concat : List (List A) ⊸ List A
concat = foldr₁ append []∘
```

3.3 Unfolding lists

After we have deconstructed lists, we will now construct lists in a linear fashion. Similar to how there are different useful linear folds, there are many sensible linear unfolds one might use. In this section, we look at an unfold which is linear in the argument to be unfolded. When unfolding lists, we have no guarantee that we will at some point reach the base case, which means that unfolds are in general partial. We therefore have to turn off Agda's termination checker for the following constructions.

We define our unfold using a standard Maybe type with constructors nothing and just. In order to use the Maybe type linearly, we again stipulate that the constructor symbols can be freely added or removed from supplies, which we do by extending our type of productions with the following rules.

```
data _⟜⟜_ : Supply → Supply → Type where
  ...
  oplnothing : (ι nothing) ⟜⟜ ◇
  laxnothing : ◇ ⟜⟜ (ι nothing)
  opljust : ι (just a) ⟜⟜ ι a
  laxjust : ι a ⟜⟜ ι (just a)
```

Using the lax productions, we can again define linear functions for each constructor, which we call $\text{nothing}_\circ$ and $\text{just}_\circ$.

The unfold that we want to devise uses one copy of the input $x : B$ to produce a list of elements of type A. This means that the recursive go function has to be linear and produce both the head of the list y and the rest $x' : B$ that is still to be unfolded from a given $x : B$. We define our function using Agda's with construct, which can be thought of as Haskell's case.

```
unfold₁ : (B ⊸ Maybe (A × B)) → B ⊸ List A
unfold₁ f x with f x
... | nothing , δ = []∘ by oplnothing ∘ δ
... | (just (y , x')) , δ = . y ::⊗ unfold₁ f x' by opl, ∘ opljust ∘ δ
```

We evaluate f at the given $x : B$. In case we are told to stop, we return the empty list. The production δ given by f says that in this case, x can be turned into nothing, which we then discard using the oplax production for this constructor. If the go function tells us to keep going, we are given an element y that we put in front of the result list as well as the rest x' still to be unfolded. In this case, δ establishes that x can be turned into just (y , x'), we therefore just have to rearrange constructors to complete our construction.

We can use the unfold to define a map function for lists.

```
map : (A ⊸ B) → List A ⊸ List B
map f = unfold₁ go where
  go : List A ⊸ Maybe (B × List A)
  go [] = nothing∘ by opl[]
  go (z :: zs) = just∘ @ (f z ,⊗ . zs) by opl::
```

The go function for map is defined by induction on the given list. In the recursive case with a given head z of the list, we establish that f applied to z should be the head of the list, and that we still need to map the rest of the list zs. We only need to use the oplax productions for lists in our definition, and the other production yoga is taken care of by unfold_1.

3.4 Correct-by-construction sorting algorithms

A linear type system gets particularly useful when we construct more complex programs, as it allows us to be sure that we did not accidentally drop or duplicate any resources. For example, a sorting function should only permute the given list and not forget or copy any of the elements. By constructing sorting functions in a linear type system, we can guarantee this. We demonstrate this by devising insertion and selection sort using the previously defined linear fold and unfold, respectively.

We assume in the following that we have a base type A which comes equipped with a computable total order, and we compare two elements $x, y : A$ by evaluating $x \leq ?\ y$, which returns either inl p (where p is a witness that x is less or equal to y) or inr p (where p is a witness that x is strictly greater than y). In Haskell lingo, our assumptions amount to saying that we are given an instance of the order type class for A.

We can define a linear insert function in a standard fashion. Given an element $x : A$ that we want to insert into a list ys (which we assume is ordered, even though we do not specify this invariant in the type), we perform pattern matching on ys.

```
insert : A × List A ⊸ List A
insert (x , []) = . (x :: []) by lax:: ∘ opl,
insert (x , (y :: ys)) with x ≤? y
... | inl p = . (x :: y :: ys) by lax:: ∘ opl,
... | inr p = . y ::⊗ insert (x , ys)
```

In case the list that we are inserting into is empty, we just give the singleton list of x, which is using the resources $\iota\ [] \otimes \iota\ x$. Otherwise, we compare x with the current head of the list. If it is less or equal, we can simply return the list with x in front. In the other case, we insert y in front of a recursive call to insert x.

With insert being defined linearly, we have done most of the work and can obtain a linear insertion sort as the standard fold of insert onto the empty list.

```
isort : List A ⊸ List A
isort = foldr₁ insert []∘
```

Our type system guarantees that isort produces a list containing the same elements as the input list. What is left to show is that this list is sorted, which we can do in a standard fashion by showing that the first projection of isort xs is sorted for any input xs.

We can use our linear unfold to devise another sorting algorithm, namely selection sort. First, we define a function which takes the minimum of a non-empty list, which is a pair of an A and a list over A, where we make use of a helper function insSnd : $A \multimap A \times \text{List}\ A \multimap A \times \text{List}\ A$ to insert an element into the second spot of a non-empty list.

```
getMin : A × List A ⊸ A × List A
getMin (x , []) = . (x , [])
getMin (x , y :: xs) with y ≤? x
... | inl p = insSnd @ (. x ,⊗ getMin (y , xs)) by id (ι x) ⊗ᶠ (lax, ∘ opl::) ∘ opl,
... | inr p = insSnd @ (. y ,⊗ getMin (x , xs)) by prod
```

We do a case distinction on whether x is not smaller than the current head of the list. If this is the case, we must look for the minimum in the given list and put x after the thereby computed minimum. The other case is symmetric, we have omitted the definition of the production which requires some reassociating.

Having defined getMin linearly, we again have done most of the work and just need to unfold.

```
ssort : List A ⊸ List A
ssort = unfold₁ go where
  go : List A ⊸ Maybe (A × List A)
  go [] = nothing∘ by opl[]
  go (x :: ys) = just∘ @ getMin (x , ys) by lax, ∘ opl::
```

In case the given list is empty, we tell unfold_1 to stop. Otherwise, we specify that the minimum of the list should be in front of the list and that we need to recurse on the remainder list given by getMin.

4 Dynamic multiplicities

So far, we have used our linear type system only to ensure that some function uses a variable once. Resource sensitive type systems offer the even greater promise that we can also precisely type programs which use multiple copies of some variable. Systems like Linear Haskell [3], Quantitative Type Theory [1] and Graded Modal Type Theory [21] introduce a resource algebra to annotate variables with such *multiplicities*. In our approach, we can use the natural number type $\mathbb{N}$ as a resource algebra to obtain quantitative features. Writing zero and suc for its constructors, we can readily define a recursive function which generates a given number of copies of some supply.

```
_^_ : Supply → ℕ → Supply
Δ ^ zero = ◇
Δ ^ (suc n) = Δ ⊗ (Δ ^ n)
```

We will see in this section that this simple definition gives rise to exactly the structure we expect from a type system with multiplicities. What is more, using a resource algebra which is part of the object language offers an advantage over static resource algebras: the natural number argument m in $\Delta \mathbin{\hat{}} m$ is an arbitrary (open) term, which allows us to dynamically compute the multiplicity of some variable depending on other variables in the context. Using this, we can type programs that cannot be adequately typed in the systems mentioned above.

In Section 4.1, we introduce notation to conveniently work with linear functions which use their input several times. We construct several functions which are polymorphic in the multiplicity, similar to what can be written in, e.g., Linear Haskell. We then proceed to consider novel linear types with dynamic multiplicities in Sections 4.2 and 4.3 with the example of linear folds and unfolds. These functions will use some given supply in each iteration, which means we have to compute how many rounds the fold and unfold take until they are finished. We close with introducing an infinite multiplicity by using the conatural numbers as a resource algebra in Section 4.4.

4.1 Multiplicities as parameters

Similar to the development of linear functions in Section 2.4, we develop abbreviations that allow us to conveniently work with functions using arbitrary multiplicities. The following definition captures linear functions that consume m copies of the given input of type A to produce something of type B.

```
_-⟨_⟩⊸_ : Type → ℕ → Type → Type
A -⟨ m ⟩⊸ B = (x : A) → ι x ^ m ⊩ B
```

For example, a copying function requires two instances of the input, which our type system allows us to capture.

```
copy : A -⟨ 2 ⟩⊸ A × A
copy x = . (x , x) by lax,
```

We can also work with functions using multiplicities in higher-order functions such as a map for lists. Given some function that requires m copies of an A to produce a B, we can derive a function that requires m copies of a list over A to produce a list over B.

```
map' : A -⟨ m ⟩⊸ B → List A -⟨ m ⟩⊸ List B
map' f [] = []∘ by ◇^ m ∘ ⊸∘^ m opl[]
map' f (x :: xs) = f x ::⊗ map' f xs by ⊗^ m ∘ ⊸∘^ m opl::
```

The function is defined by pattern matching as usual, in the productions given above we use $\multimap\!\multimap^{\wedge} m$ to apply a given production m times, $\diamond^{\wedge} m$ to discard m copies of the empty bag, and $\otimes^{\wedge} m$ to turn $(\iota\, x \otimes \iota\, xs)^{\wedge} m$ into $(\iota\, x^{\wedge} m) \otimes (\iota\, xs^{\wedge} m)$.

We can apply a function of type $A \text{-}\langle m \rangle\!\multimap B$ to a value derived using a supply Δ, the resulting value of type B will need m copies of Δ. We can establish this similarly to the linear application operator, applying m times the given production δ that turns Δ into some A.[78]

$$\_\ @\langle\ \_\ \rangle\_ : A \text{-}\langle\ m\ \rangle\!\multimap B \to (m : \mathbb{N}) \to \Delta \Vdash A \to \Delta\ ^{\wedge}\ m \Vdash B$$
$$f\ @\langle\ m\ \rangle\ (a\ ,\ \delta) = f\ a \text{ by } \multimap\!\multimap^{\wedge}\ m\ \delta$$

When composing two functions which use n and m copies of their input, respectively, we expect that we need $n \cdot m$ copies of the input, where $\_\cdot\_$ denotes multiplication of natural numbers. Indeed, we can derive this composition principle by applying twice the application operator for functions with multiplicities.

$$\text{compose} : A \text{-}\langle\ n\ \rangle\!\multimap B \to B \text{-}\langle\ m\ \rangle\!\multimap C \to A \text{-}\langle\ n \cdot m\ \rangle\!\multimap C$$
$$\text{compose}\ f\ g\ x = g\ @\langle\ m\ \rangle\ (f\ @\langle\ n\ \rangle\ .\ x) \text{ by explaw } (\iota\ x)\ n\ m$$

The only work necessary to define compose lies in constructing a production that establishes that taking $(m \cdot n)$ copies of some supply Δ is the same as computing $(\Delta^{\wedge} m)^{\wedge} n$. The proof is not essentially difficult, but requires multiple inductions, we refer the interested reader to the artefact for details.

We can use compose to apply our copy function twice to some given value.

$$\text{copytwice} : A \text{-}\langle\ 4\ \rangle\!\multimap (A \times A) \times (A \times A)$$
$$\text{copytwice} = \text{compose}\ 2\ 2\ \text{copy copy}$$

We do not have to do any work to establish that copytwice uses literally 4 elements of the input, as Agda reduces $2 \cdot 2$ to its canonical form for us.

4.2 A dynamic fold for lists

The fold for lists we introduced in Section 3.2 is only one of several useful linear folds. Another fold allows the recursive case to also consume some supply Δ. Consequently, we expect that using such a kind of fold yields a linear judgment which needs Δ as many times as the folded list is long. We can derive such a fold similar to foldr_1 where we make use of a standard length function $\text{len} : \text{List}\ A \to \mathbb{N}$.

$$\text{foldr}_2 : ((x : A) \to (b : B) \to \iota\ b \otimes \iota\ x \otimes \Delta \Vdash B) \to \diamond \Vdash B \to (xs : \text{List}\ A) \to \Delta\ ^{\wedge}\ (\text{len}\ xs) \otimes \iota\ xs \Vdash B$$
$$\text{foldr}_2\ f\ z\ [] = z \text{ by opl[]}$$
$$\text{foldr}_2\ f\ z\ (x :: xs) = f\ x\ @\ \text{foldr}_2\ f\ z\ xs$$

The definition of foldr_2 is similar to the definition of foldr_1, we only neeed to use a different production to reorder the elements (we have omitted its definition as it requires reassociating, and refer the interested reader to the artefact). We have assumed that the base element z requires no resources, we could also pass down a supply to the base case like we did in foldr_1 and thereby obtain a more fold generalising both foldr_1 and foldr_2.

[7]The function argument f and multiplicity m are of course the wrong way around in the type signature presented here and are in opposite order in the artefact, where we make use Agda's syntax feature to be able to write function application in this way.

[8]The attentive reader will have spotted that application for functions with multiplicities does not allow for functions with leftovers. It is not difficult to derive such a more general application principle, but using this requires more guidance for the type-checker. To avoid having to write supply annotations explicitly, we only work with the simpler application principle in the following.

We can use foldr_2 to give an exact type for a function that interleaves an element x with the elements of some list ys.

```
intersperse : (x : A) (ys : List A) ⟶ ι x ^ (len ys) ⊗ ι ys ⊩ List A
intersperse x = foldr₂ (λ y xys ⟶ . x ::⊗ . y ::⊗ . xys by prod y xys) []∘ where
  prod : (y : A) (xys : List A) ⟶ ι xys ⊗ ι y ⊗ ι x ⊸ ι x ⊗ ι y ⊗ ι xys
```

The linear intersperse gives as the base case an empty list. Otherwise, we are given the current head of the list y and the list xys which has x interspersed throughout the tail of the list, and put x and y in front of that list. We hence use one copy of $\iota\, x$ in each step of the fold, which is precisely what foldr_2 allows us to do.

The above intersperse is not exactly the same as Haskell's `intersperse` from `Data.List` since we also put x in front of the list. If we want to have a proper interspersing function, we have to do three-way pattern matching. Using the predecessor function predℕ (which sends zero to zero), we can derive this interspersing function.

```
intersperse' : (x : A) (ys : List A) ⟶ ι x ^ predℕ (len ys) ⊗ ι ys ⊩ List A
intersperse' x [] = . []
intersperse' x (y :: []) = . (y :: [])
intersperse' x (y :: y' :: ys) = . x ::⊗ . y ::⊗ intersperse' x (y' :: ys)
```

In the first two cases of intersperse', the proof goal normalises such that it is immediate that we do not have available any x. We can hence directly return the given input list. Only in the third case we have available an x which we put in front y and the rest of the interspersed list.

4.3 Unfolding lists, or: counting the rounds

With our dynamic multiplicities can also devise more complex unfolds, for instance one which consumes a supply Δ each time it creates an element of the result list. To give an adequate type signature for such an unfold, we need to know the number of rounds that a given go function $f : B \longrightarrow \mathsf{Maybe}\ ((\Delta \vdash A) \times B))$ takes takes until it returns nothing. Here we assume that the result type B is not a relevant resource; again, this is not a crucial constraint, and we could combine unfold_1 and unfold_2 into a single, more general unfold. The function f returns a pair of a linear judgment turning Δ into A, as well as the next element to be unfolded separately outside of the linear judgment. We compute the number of rounds until f returns nothing by repeatedly applying it to the input.

```
rounds : (B ⟶ Maybe ((Δ ⊩ A) × B)) ⟶ B ⟶ ℕ
rounds f x with f x
... | nothing = zero
... | just (_ , x') = suc (rounds f x')
```

Using the rounds function, we can dynamically compute the number of copies we need of the supply Δ, which allows us to give the correct type for our unfold_2.

```
unfold₂ : (f : B ⟶ Maybe ((Δ ⊩ A) × B)) ⟶ (x : B) ⟶ Δ ^ rounds f x ⊩ List A
unfold₂ f x with f x
... | nothing = []∘
... | just ((y , δ) , x') = . y ::⊗ unfold₂ f x' by δ ⊗[f] id (Δ ^ rounds f x')
```

The definition of unfold_2 is similar to unfold_1, Agda's type-checker realises that since $f\ x$ did not return nothing, our counting function rounds $f\ x$ will return a number greater than zero, which means that we have available a copy of Δ which we can turn into y using the given δ.

An important example of a type B whose values are not resources from the point of view of linear logic is the natural number type. We can intuit this by observing that any natural number consists only of constructor symbols (namely zero and suc), and therefore contains no resources. A natural numbers argument m is unfolded in the replicate function to put m copies of an element x into a list.

```
replicate : (m : ℕ) → A -⟨ m ⟩⊸ List A
replicate m x = unfold₂ go m by prod where
  go : ℕ → Maybe ((ι x ⊩ A) × ℕ)
  go zero = nothing
  go (suc m) = just (. x , m)
  prod : ι x ^ m ⊶ ι x ^ rounds go m
```

The definition of the go function is straightforward. To convince the type-checker that our construction is linear, we only have to prove that it takes exactly m rounds to turn m into zero, the inductive construction of prod follows directly via pattern matching and recursion.

4.4 Using a variable infinitely often

We can also draw multiplicities from other types than the natural numbers. To reflect infinite use of some variable, we will in this section adapt the definition of supply exponentiation _^_ and linear function types to take a conatural number as multiplicity. The conatural numbers ℕ∞ can be defined in Cubical Agda using guarded corecursion [4, 19]. Crucially, the conatural numbers have a value ∞ : ℕ∞ which is equal to its successor. This property allows us to derive the following production, which captures that we can always take another copy of a supply with multiplicity ∞.

```
∞prod : (Δ : Supply) → (Δ ^ ∞) ⊶ (Δ ⊗ (Δ ^ ∞))
```

Using our ∞ multiplicity, we can adequately give a linear type to a repeat function which produces an infinite list from a given element $x : A$.[9]

```
repeat : A -⟨ ∞ ⟩⊸ List A
repeat x = . x ::⊗ repeat x by ∞prod (ι x)
```

The repeat function is defined like the usual `repeat` of Haskell, we only need to convince the type-checker that we are using the specified resources using the derived production for ∞. Similarly, we can iterate a linear function on a given element x, making again use of ∞-many copies of x.

```
iterate : (A ⊸ A) → A -⟨ ∞ ⟩⊸ List A
iterate f x = . x ::⊗ (iterate f @⟨ ∞ ⟩ (f @ . x)) by ∞prod (ι x)
```

Note that using a variable an infinite number of times is different from using a variable without any resource annotation (which we can do in our setting by not including it in our linear judgment, as we have done for type B in unfold₂). The multiplicity ∞ requires that some variable really is used infinitely often in a program.

5 Linear dependent programming

While we have made use of dependent types to devise our linear type systems, we have so far not actually programmed *with* dependent types. We will demonstrate in this section that we can seamlessly do this in our system. Since our linear type system lives in an ambient dependent type theory, we actually have no problems to integrate dependent types, and can linearise these just like

[9] If we were working in a lazy language in which ∞ is represented by a non-terminating program yielding suc (suc (. . .)), we could also write repeat and iterate as instances of unfold₂ as the rounds function would then compute ∞ for us.

we linearised non-dependent data types. In particular, all definitions and constructions for linear functions introduced in Sections 2.4 and 4.1 generalise to dependently typed functions. For example, we can derive an application operator for linear functions from type A to a type B dependent on A in exactly the same way as the non-dependent operator.

```
_@_ : ((x : A) → ι x ⊗ Δ₀ ⊩ B x) → ((a , _) : Δ₁ ⊩ A) → Δ₀ ⊗ Δ₁ ⊩ B a
f @ (a , δ) = f a by swap Δ₀ (ι a) ∘ id Δ₀ ⊗ᶠ δ
```

Using dependent types when programming allows us to guarantee that certain programs are linear, in contrast to their non-dependent relatives. Consider for example the usual `zip` function for lists, which might drop elements if the input lists have differing lengths. This case can be excluded by using a dependent type to capture length-indexed lists, usually called vectors or Vec in short. We can work with vectors in the same way as lists, and after imposing productions similar to the ones for lists in Section 3.1, we can define a linear zip in the usual way.

```
zip : {n : ℕ} (xs : Vec A n) → (ys : Vec B n) → ι ys ⊗ ι xs ⊩ Vec (A × B) n
zip {zero} [] [] = []∘ by opl[]' ⊗ᶠ opl[]'
zip {suc n} (x :: xs) (y :: ys) = (. x ,⊗ . y) ::⊗ zip xs ys
```

6 Discussion & related work

We will in the following recapitulate our approach and situate it in the rich landscape of dependent linear type theories. We will sketch the semantics of our theory (without going into any detail) as this will make it easier to explain how the construction can be added in the absence of higher inductive types in Section 6.1, and to compare it with related work in Section 6.2.

6.1 Sketching the semantics

The basic idea behind our linear type system was to deeply embed the rules of linear logic into dependent type theory. Semantically, we can trace this as follows: given a model of dependent type theory (for example categories with families [12]), we equip each context Γ with a symmetric monoidal category Supply_Γ whose hom-set is given by the productions _⊸_. The monoidal unit of Supply_Γ is given by $\diamond$, and _⊗_ acts as the bifunctor of Γ with its action on morphisms captured by $\otimes^f$. We then embed any term of the host theory into our linear logic using a natural transformation ι, which is moreover strong monoidal with respect to products of types in the host theory and Supply_Γ. The structural morphisms of a strong monoidal functor are called lax and oplax, which explains the names we gave to the productions corresponding to each constructor of a data type.

This semantic picture of course requires that we equate certain productions, for example, that composition of productions is associative. However, in practice we do not have to worry about equalities between productions, since we only care about the existence of *some* production to derive a linear judgment. This means we can simply stipulate the construction rules for supplies and productions using a data type. For example, in standard Agda we can define supplies with three constructors for the empty supply, the singleton supply and the union of two supplies.

```
data Supply : Type where
  ◇ : Supply
  ι : {A : Type} (a : A) → Supply
  _⊗_ : Supply → Supply → Supply
```

For the productions, we have to introduce additional constructors on top of the ones of the data type given in Section 2.2 since the stipulated union of supplies ⊗ does not satisfy commutativity, associativity and unital laws any more. The following set of rules is sufficient to characterise

the structural productions used in this paper, and the productions for specific data types can be appended to the following data type as before.

```
data _⊸_ : Supply → Supply → Type where
  id : ∀ Δ → Δ ⊸ Δ
  _∘_ : ∀ {Δ₀ Δ₁ Δ₂} → Δ₁ ⊸ Δ₂ → Δ₀ ⊸ Δ₁ → Δ₀ ⊸ Δ₂
  _⊗ᶠ_ : ∀ {Δ₀ Δ₁ Δ₂ Δ₃} → Δ₀ ⊸ Δ₁ → Δ₂ ⊸ Δ₃ → (Δ₀ ⊗ Δ₂) ⊸ (Δ₁ ⊗ Δ₃)
  unitr : ∀ Δ → Δ ⊗ ⋄ ⊸ Δ
  unitr' : ∀ Δ → Δ ⊸ Δ ⊗ ⋄
  swap : ∀ Δ₀ Δ₁ → Δ₀ ⊗ Δ₁ ⊸ Δ₁ ⊗ Δ₀
  assoc : ∀ Δ₀ Δ₁ Δ₂ → (Δ₀ ⊗ Δ₁) ⊗ Δ₂ ⊸ Δ₀ ⊗ (Δ₁ ⊗ Δ₂)
```

The types Supply and _⊸_ can similarly be added to other dependently typed systems such as Rocq [2] or Lean [11], and thereby enable linear reasoning in these systems.

Our approach crucially relies on dependent pattern matching whenever we derive some witness of the linear judgment $\Delta \Vdash A$. Adding our system to, e.g., Haskell means we either have to stipulate linear elimination rules, or make changes to the type-checker.

6.2 Related work

There has been a wide range of work combining linear with dependent type theories, in the following we compare our approach with the main strands of work.

Linear dependent type theories. Our theory follows the insight of McBride [18] that we need not be concerned about the occurrence of terms in *types* when tracking variable use. This makes it easier to combine dependent and linear types, and is more expressive than other theories which distinguish between intuitionistic (i.e., dependent) and linear variables, where dependent types cannot mention linear variables [6, 14, 26]. Our system is hence more similar to theories which ignore variable use in types such as Quantitative Type Theory (QTT) of Atkey [1], implemented as Idris by Brady [5], or Graded Modal Type Theory (GMTT) implemented as Granule by Orchard et al. [21]. These systems equip the structural rules of dependent type theory with multiplicities, which are elements of some *resource algebra*. In our theory, we follow the insight of Pédrot [22, 23] that we can introduce linearity in dependent type theory through the Dialectica construction, which in the case of function types equips the output with copies of the inputs. This allows us to use types of the underlying theory such as the natural number type as a resource algebra, which means we can dynamically compute multiplicities to capture if the use of some input depends on another input.

On the other hand, our system does not offer an unrestricted multiplicity akin to, e.g., the multiplicity ω in Idris. In our system, we always have to precisely state how often some variable is to be used. Furthermore, our system does not offer linear function types, in contrast to QTT and GMTT. We work around this fact by using functions of the host theory to capture higher-order functions, which is surprisingly practical as we have seen in our programming examples. However, this cannot resolve the fact that we do not have linear higher-order functions and cannot reason about *function variables* as resources—e.g., in the map function from Section 3.3, we might want to represent how often the mapped function is used, but this is not possible in our present system. In summary, our system offers some features that QTT and GMTT do not possess, but is more limited in other ways.

Dynamic multiplicities. Similar features to our dynamic multiplicities are offered by the *index terms* of dℓPCF developed by Dal Lago and Gaboardi [8], which is a dependent linear type system for PCF [24]. Index terms can be build using (partial) functions and can thereby compute how often

a program uses some argument. The set of functions that can be used to compute an index term is a parameter to their type system to take into account a trade-off between expressivity of index terms and undecidability of the typing relation. Something similar emerged in our setting: when deriving compose in Section 4.1, we had to give an explicit proof of an exponentiation law for _^_. Since equality of open terms of the natural numbers is undecidable, we also have in this sense an undecidable type system. We can however always carry out manual proof work to establish that some program is of a given type.

The type system of Dal Lago and Gaboardi [8] was developed to capture computational complexity, which is a different objective from ours. For example, the types of dℓPCF allow for specifying the number of evaluation steps of a function between the natural numbers. In our system, the natural numbers are not considered resources since they only consist of constructor symbols. Instead, we are concerned with the uses of variables of some collection of base types. Furthermore, our base theory is a (total) dependent type theory with data types such as lists, and our productions explain how such data types can be used in a linear setting.

Dal Lago and Gaboardi prove *relative completeness* of their type system, which states that any reducing term t can be typed in their system with some index term which is smaller than the number of reductions used for t by Krivine's machine [15]. We expect that in our system, it is also the case that any program can be given a linear type since the language we use to compute multiplicities is the same language in which we program, but more work is necessary to establish such a result. In general, more research is necessary to understand the precise relationship between dℓPCF and our system, which is blurred by the fact that both type systems have been developed on top of different programming languages.

7 Conclusions & future work

We have added a linear type system to dependent type theory by leaving its structural rules unchanged, but instead deeply embedding the rules of linear logic. Programming in this system is practical, and we can devise programs the same way as we would in any other functional language. Like systems which make their typing rules substructural using a resource algebra—such as Linear Haskell [3], Quantitative Type Theory [1] or Graded Modal Type Theory [21]—we can write functions that are polymorphic in the multiplicity of some variable. But in contrast to these systems, our system has the advantage that we can compute multiplicities dynamically, which enables precise typing of functions which require some number of copies of a variable depending on some other variables.

Our approach has limitations, however: the production type has to be extended for any data type that we are using; productions which justified linearity of our programs had to be constructed manually; and we do not have linear dependent function types and a multiplicity for unrestricted use. In the following we discuss how these issues can be addressed, and point to some other lines of future work.

Deriving productions automatically. While we could program in our system in a usual functional way, we had to manually construct productions to convince the type-checker that we have indeed used the presumed resources. Most of the productions are straightforward to derive as they were just chaining together constructor productions, unital and commutativity laws and reassociations. It is hence natural to devise a tactic using Agda meta-programming which constructs such productions automatically. A simple ring solver which normalises supplies to lists of variables, using oplax productions for term constructors where necessary, would be enough to find that most of the programs in this paper are linear. Further study is needed to see which class of productions is decidable, and if we have a practical tactic to also derive productions involving the congruence $\otimes^f$.

Furthermore, it is tedious to extend the type of productions with constructors for any data type that one introduces. It would be convenient if this can be carried out automatically using Agda meta-programming.

Dependent function spaces and unrestricted variable use. The system we have presented has no support for linear higher-order functions. In order to add linear function spaces to our theory, we need to *close* the symmetric monoidal categories that we equip our type theory with. Furthermore, we need a variable abstraction principle for supplies, which requires changing the host dependent type theory. The author has work in progress to this effect. One prospect of introducing linear function spaces is that we can also keep track of how often some function variable is used. For example, we could represent in the type of map from Section 3.3 that the mapped function is used as often as the given list is long.

We can also add an exponential comonad to our categories, which would allow us to freely drop and duplicate supplies which are annotated with a bang ! operator. This would give us an equivalent of the multiplicity ω from Idris [5].

Integrating other substructural logics. We expect that the approach of deeply embedding the structural rules of some logic in dependent type theory is useful beyond linear logic. We can for instance have an affine dependent type system by adding contraction to our production data type. Further study is needed to study if all substructural logics can be integrated into type theory with the present approach.

Dynamic multiplicities in the compiler. Another big prospect of linear type systems is that they make the compiler's life easier to find efficient ways to execute a program. This is not at all leveraged by the present approach since the compiler of Agda does not know about the resource annotations that we have made. Further research is needed to understand how a compiler can make use of dynamic multiplicities when constructing an executable.

Complexity analysis. One intriguing application of linear type theories lies in analysing the complexity of programs inside type theory. For instance, we can make the comparison function used in our sorting algorithms in Section 3.4 resource-relevant, which means that we need to give a sufficient supply of comparison operations to type a sorting program. This would give us a way to reason about the complexity of different algorithms. Further study is needed to see if this approach is practical to analyse the complexity of programs in the spirit of implicit complexity theory [9, 10, 13, 27]. This would bring our approach also closer to the work of Dal Lago and Gaboardi [8].

Acknowledgments

I am indebted to Valeria de Paiva and Pierre-Marie Pédrot for introducing me to the Dialectica construction; and to Nathan Corbyn and Daniel Gratzer for clearing up my type-theoretic confusions. I am grateful for helpful discussions with Pedro H. Azevedo de Amorim, Evan Cavallo, Sean Moss and Sam Staton; and for insightful comments of several anonymous reviewers. The suggestions of one reviewer in particular greatly helped to improve the presentation of this Pearl.

References

[1] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* LICS (2018), 56–65. https://doi.org/10.1145/3209108.3209189

[2] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. 1997. *The Coq Proof Assistant Reference Manual : Version 6.1.* Research Report RT-0203. INRIA. 214 pages. https://hal.inria.fr/inria-00069968

[3] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158093

[4] Lars Birkedal and Rasmus Ejlers Møgelberg. 2013. Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes. LICS (2013), 213–222. https://doi.org/10.1109/LICS.2013.27

[5] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

[6] Iliano Cervesato and Frank Pfenning. 2002. A linear logical framework. *Inforation and Computation* 179, 1 (Nov. 2002), 19–75. https://doi.org/10.1006/inco.2001.2951

[7] Vikraman Choudhury and Marcelo Fiore. 2023. Free Commutative Monoids in Homotopy Type Theory. *Proceedings of the 38th International Conference on Mathematical Foundations of Programming Semantics* (2023). https://doi.org/10.46298/entics.10492

[8] Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. *Proceedings of the 26th Annual ACM/IEEE Symposium on Logic in Computer Science* LICS (2011), 133–142. https://doi.org/10.1109/LICS.2011.22

[9] Ugo Dal Lago and Martin Hofmann. 2011. Realizability Models and Implicit Complexity. *Theoretical Computer Science* 412, 20 (2011), 2029–2047. https://doi.org/10.1016/j.tcs.2010.12.025 Girard's Festschrift.

[10] Ugo Dal lago and Barbara Petit. 2013. The geometry of types. *SIGPLAN Not.* 48, 1 (Jan. 2013), 167–178. https://doi.org/10.1145/2480359.2429090

[11] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). CADE (2015), 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[12] Peter Dybjer. 1996. Internal type theory. In *Types for Proofs and Programs (TYPES) 1995*, Stefano Berardi and Mario Coppo (Eds.). Springer, 120–134. https://doi.org/10.1007/3-540-61780-9_66

[13] Harrison Grodin and Robert Harper. 2024. Amortized Analysis via Coalgebra. *Electronic Notes in Theoretical Informatics and Computer Science* Volume 4 - Proceedings of MFPS XL, Article 10 (Dec 2024). https://doi.org/10.46298/entics.14797

[14] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/2676726.2676969

[15] Jean-Louis Krivine. 2007. A Call-By-Name Lambda-Calculus machine. *Higher Order Symbolic Computation* 20 (2007), 199–207. https://doi.org/10.1007/s10990-007-9018-9

[16] Ian Mackie. 1994. Lilac: a Functional Programming Language Based on Linear Logic. *Journal of Functional Programming* 4, 4 (1994), 395–433. https://doi.org/10.1017/S0956796800001131

[17] Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 73 – 118. https://doi.org/10.1016/S0049-237X(08)71945-1

[18] Conor McBride. 2016. *I Got Plenty o' Nuttin'*. Springer International Publishing, Cham, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

[19] Hiroshi Nakano. 2000. A Modality for Recursion. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS '00)*. IEEE Computer Society, USA, 255.

[20] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph. D. Dissertation. Chalmers University of Technology and Göteborg University.

[21] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (July 2019), 30 pages. https://doi.org/10.1145/3341714

[22] Pierre-Marie Pédrot. 2014. A functional functional interpretation. *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science* LICS/CSL (2014). https://doi.org/10.1145/2603088.2603094

[23] Pierre-Marie Pédrot. 2024. Dialectica the Ultimate. (2024). https://www.p%C3%A9drot.fr/slides/tlla-07-24.pdf Talk at Trends in Linear Logic and Applications.

[24] Gordon D. Plotkin. 1977. LCF Considered As a Programming Language. *Theoretical computer science* 5, 3 (1977), 223–255. https://doi.org/10.1016/0304-3975(77)90044-5

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

[26] Matthijs Vákár. 2015. A Categorical Semantics for Linear Logical Frameworks. In *Foundations of Software Science and Computation Structures*, Andrew Pitts (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 102–116. https://doi.org/10.1007/978-3-662-46678-0_7

[27] Jan van Brügge. 2024. Liquid Amortization: Proving Amortized Complexity with LiquidHaskell (Functional Pearl). *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium* Haskell (2024), 97–108. https://doi.org/10.1145/3677999.3678282

[28] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2021. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming* 31 (2021), e8. https://doi.org/10.1017/S0956796821000034

[29] Kurt Von Gödel. 1958. Über Eine Bisher Noch Nicht benützte Erweiterung Des Finiten Standpunktes. *Dialectica* 12, 3-4 (1958), 280–287. https://doi.org/10.1111/j.1746-8361.1958.tb01464.x

Received 2025-02-27; accepted 2025-06-27