Dependent Multiplicities in Dependent Linear Type Theory

MAXIMILIAN DORÉ, Department of Computer Science, University of Oxford, United Kingdom

We present a novel dependent linear type theory in which the multiplicity of some variable—i.e., the number of times the variable can be used in a program—can depend on other variables. This allows us to give precise resource annotations to many higher-order functions that cannot be adequately typed in any other system. Inspired by the Dialectica translation, our typing discipline is obtained by embedding linear logic into dependent type theory and specifying how the embedded logic interacts with the host theory. We can then use a standard natural numbers type to obtain a quantitative typing system with dependent multiplicities. We characterise the semantics for our theory as a combination of standard models of dependent type theory and linear logic. Our system can be added to any dependently typed language, which we demonstrate with an implementation in Agda.

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

Additional Key Words and Phrases: Linear dependent type theory, Quantitative type theory, Linear higher-order logic

ACM Reference Format:
Maximilian Doré. 2025. Dependent Multiplicities in Dependent Linear Type Theory. 1, 1 (July 2025), 27 pages.

1 Introduction

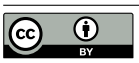
Girard's linear logic [26] has been applied very fruitfully to the design of programming languages in recent years, leading to the development of Linear Haskell [10], Quantitative type theory [5, 34] implemented in Idris [12], and Graded Modal type theory implemented as Granule [37]. The type systems of these languages allow the user to specify how often some variable is to be used in a program, let us call this number the variable's *multiplicity*. Annotating programs with multiplicities significantly reduces the space of possible programs for some given type, guiding both the user in making sure they write correct programs, and the compiler trying to find efficient ways to execute some functional program.

However, the linear typing discipline of these systems has serious limitations when we try to give types to higher-order functions, where it is seldomly possible to assign a static multiplicity to a variable. Consider the following Agda [40] program which folds $z : \mathsf{Maybe}\ A$ by either applying the function j to x in case z is $\mathsf{just}\ x : \mathsf{Maybe}\ A$, or otherwise returning the backup element n.

```
foldMaybe : (z : Maybe A) → (j : A → B) → (n : B) → B
foldMaybe (just x) j n = j x
foldMaybe nothing j n = n
```

The multiplicity of both j and n is either 0 or 1—*depending* on the given z. Existing linear and quantitative type theories, which are based on static multiplicities, cannot represent this use of variables, while systems with a more dynamic notion of resource annotation do not support

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



ACM XXXX-XXXX/2025/7-ART
https://doi.org/

linear higher-order functions [16]. Current programming languages get around this by providing some sort of approximate linear types, either by offering an unrestricted multiplicity or providing bounds for the multiplicity with affine types. However, this limits the space of valid programs not sufficiently and, e.g,. does not rule out a faulty implementation of foldMaybe which always returns $n : B$, no matter the shape of the given z.

Did somebody say a type depends on some value? Enter Martin-Löf's dependent type theory [33]. Dependent type theory equips intuitionistic logic with predicates, which allows for expressing detailed specifications in the type of a program [32]. Indeed, we could use inductive families, a common feature of today's dependently typed languages such as Agda, to devise types which wrap j and n and are inhabited depending on the given Maybe A [22, 35]. We will see in this paper that we can do this systematically and thereby obtain a fully fledged dependent linear type theory in which higher-order functions such as foldMaybe can be precisely typed.

We obtain our principled typing discipline by embedding linear logic into dependent type theory. Our approach is inspired by the Dialectica translation [19, 20, 50], which provides a recipe to turn an intuitionistic calculus into a linear one. Importantly, our work is not about translating intuitionistic terms, rather, we use the *target* of the Dialectica translation as inspiration for a type theory. The observation that Dialectica can provide a way to devise a dependent linear type theory has been made by Pédrot [42, 43], who characterises the Dialectica translation for dependent type theory as giving the "higher-order dependent graded type of a term". In the case of functions, Dialectica amounts to equipping the codomain with copies of the domain of the function, thereby giving a way to specify how often some input is to be used. This idea has been used by Doré [21] to devise a linear typing disciple in Cubical Agda [49] based on finite multisets, the main structure necessary for the approach envisioned by Pédrot.

We take this idea to its completion, adding function types and inductive types to the picture and thereby obtaining a fully fledged *dependent linear type theory* in which the multiplicity of some input variable can *depend* on another variable. We achieve this by embedding not only a symmetric monoidal structure into type theory, but by also *closing* this structure and specifying how the embedded linear logic interacts with the dependent type theory. We also show how to extend our system with a ! (bang) operator, in case we still want to use a variable in an unrestricted fashion (even though the need for such an unrestricted multiplicity greatly reduces in our system).

Since our theory works by adding more structure to a dependent type theory—as opposed to changing its structural rules—we can easily add our theory to existing languages. We demonstrate this with an experimental implementation of our system in Agda, but our construction directly carries over to other dependently typed languages such as Rocq [8] or Lean [18]. We can program in the resulting typed functional language in a practical way, and will for the first time be able to give precise linear types for common higher-order functions. In particular, we will give the following type to foldMaybe, where | isJust z | returns 1 if z is just and 0 otherwise, and | isNothing z | the opposite values.

$$\langle\ z : \mathsf{Maybe}\ A\ \rangle \multimap \langle\ j : \langle\ A\ \rangle \multimap B\ \rangle^{\wedge}\ |\ \mathsf{isJust}\ z\ | \multimap \langle\ n : B\ \rangle^{\wedge}\ |\ \mathsf{isNothing}\ z\ | \multimap B$$

The input j is now a linear function, which we denote with $\langle\ \_\ \rangle \multimap \_$. The whole program itself is a linear function, where function types with domain $\langle\ \_\ \rangle^{\wedge}$ expect some term of the natural numbers as resource annotation, which allows us to give the right type to foldMaybe. In fact, since Agda is total there is only a single program with this type, which means we have fully narrowed down the space of possible implementations with this type.

While foldMaybe only uses non-dependent base types A and B, we can moreover seamlessly program with dependent types since we work in an ambient dependent system. By following

McBride [34] in not caring about the usage of variables in types, but only in programs, we thereby obtain an extremely rich language to specify the behaviour of programs.

Overview of the construction. Let us explain our approach in a bit more detail. We start with dependent type theory [31], called the host theory, and stipulate the existence of two new types, the type of *supplies*, written $\mathsf{Supply} : \mathsf{Type}$, and the type of *productions* $\_\triangleright\_ : \mathsf{Supply} \to \mathsf{Supply} \to \mathsf{Type}$. These types are equipped with the structural rules of linear logic, for example, we can join together two supplies $\Delta_0, \Delta_1 : \mathsf{Supply}$ with an operation $\otimes$, and we have productions which allow us to turn $\Delta_0 \otimes \Delta_1$ into $\Delta_1 \otimes \Delta_0$ and vice versa. Semantically speaking, we equip each context of the host theory with a symmetric monoidal closed category of supplies, with productions capturing the hom-set of each category.

Crucially, we have an inclusion ι which allows us regard any value $a : A$ of the host theory as a supply, i.e., $\iota\ a : \mathsf{Supply}$. In particular, we can put variables into a supply, which motivates us to think of supplies as some sort of "linear contexts"—importantly, these are not contexts since only the host theory introduces *new* variables, but the supplies do specify which variables we are to use in a construction. Accordingly, we define a dependent type $\Delta \Vdash A$ which we read as "linear entailment" of some type A using the resources of the supply Δ. Writing $\Gamma \vdash$ for entailment in the host theory, we therefore obtain a nested calculus $\Gamma \vdash \Delta \Vdash A$ in which the intuitionistic context Γ introduces (possibly dependent) variables, and Δ repeats the variables we are to use to construct A. Crucially, we can specify that we use m many copies of some variable for any (open) term m of the natural numbers, which gives rise to a quantitative system with dependent multiplicities.

By slightly generalising our type $\Vdash$ to account not only for the derivation of intuitionistic types A, but instead *pairs* of an intuitionistic type A and a supply which tells us how an inhabitant of A is to be considered a resource, we can internalise the nested entailment $\_\vdash\_\Vdash$ as a linear function type, which allows us to express and study linear higher-order functions. Crucial for the implementation of linear function types is the addition of a functor Λ on the symmetric monoidal closed structure, which can be understood as a variable abstraction principle for supplies. For example, $\iota\ x \otimes \iota\ x$ is a supply in context $x : A$, whereas $\Lambda_{x:A}\ \iota\ x \otimes \iota\ x$ binds x and is a supply in the empty context. Semantically, Λ is a right-adjoint to the action of *context extension* on supplies. Context extension is the main structural rule of dependent type theory, requiring a closure principle for this for our linear structure suggests that our calculus really is a proper combination of linear and dependent type theory.

We have to do surprisingly little work to also incorporate inductive types into our system. By introducing productions which capture that constructor symbols can always be freely added or removed from a supply as long as the contained resources are maintained, our approach naturally extends to take into account data types such as the already mentioned Maybe, or also recursive types such as lists. Utilising that our supplies can contain arbitrary terms, and not just variables, we can derive linear elimination principles for inductive data types from their usual dependent elimination rules. This makes our theory directly applicable to functional programming.

Implementation. We have implemented our type system in Agda. All example programs in this paper are taken from this implementation where they type-check. However, we want to stress that our construction is not specific to Agda and can be carried out in any dependently typed language. We do not make use of any esoteric type-theoretic constructs (Supply is a just a type with several constructors, and productions $\triangleright$ are a relation on this type). The only Agda-specific feature we use is Agda's reflection capability to alleviate the proof burden of showing that some construction uses the provided resources, but systems such as Rocq [8] or Lean [18] offer similar capabilities.

The implementation and an explanation linking the code in this paper to the code can be found at https://github.com/maxdore/dltt.

Summary of contributions.

- We give syntax for a *dependent linear type theory* which allows for specifying *dependent multiplicities* of variables, thereby giving rise to a quantitative type theory.
- Our system allows us type many higher-order functions which, to the author's knowledge, cannot be typed in any other system.
- We show how our theory can be extended with a ! operator to allow unprincipled use of some variable in a program.
- We specify models for our theory, which are a relatively immediate combination of standard models of dependent type theory and linear logic.
- Our work contributes to the understanding of the Dialectica translation for positive types, and further illuminates the interaction between linear logic and dependent type theories.
- We give a proof-of-concept implementation of our type theory in Agda in which one can program in familiar function fashion.

Outline. We setup our syntax in Section 2, where we also explain our definitions of linear derivability and function types. We then show in Section 3 how our theory naturally affords quantitative types using a standard natural numbers type of the host theory, which allows us to introduce our notion of dependent multiplicities. As an extension to the basic syntax, we present a way to add a ! operator in Section 4, a variable annotated with ! can then be used arbitrarily often in a program. We specify the semantics of our theory Section 5, where we also give two exemplary models of our theory. We compare our approach in more detail with existing (dependent) linear and quantiative type systems in Section 6 before closing with an outlook on possible future lines of work in Section 7.

2 Syntax for embedding linear logic in dependent type theory

We base our development on a standard dependent type theory à la Martin-Löf, which we will often call our *host theory*. We use the basic judgments $\Gamma \vdash A : \mathsf{Type}$ for A being a type in context Γ and $\Gamma \vdash a : A$ for a being a term of type A. Our construction works independently of whether the basic type theory is intensional [31] or extensional [32] since we will never need to make use of an equality proof. Let us hence simply write $\Gamma \vdash a = a' : A$ for the type of equalities between $a, a' : A$, and we are unconcerned if this equality type has rich structure like in homotopy type theory [47], has only unique inhabitants, or even coincides with definitional equality. For legibility we do not worry about different universes, but everything in this paper works fully predicatively.

We have implemented our syntax and type system in the theorem prover Agda [40] and will use its notation in the following, but our constructions can be carried out in any dependent type theory. We will use $A, B, \ldots$ to denote a collection of base types. Functions are introduced as $\lambda x.b$, which are inhabitants of the dependent function type $(x : A) \to B\ x$ for $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma, x : A \vdash B\ x : \mathsf{Type}$. Function application of $f : (x : A) \to B\ x$ to $a : A$ is written as $f\ a : B\ a$. Elements of the dependent pair type $\Sigma[\ x \in A\ ]\ B\ x$ are written as $(x\ ,\ y)$ for $x : A$ and $y : B\ x$, we will also occasionally use projection functions fst and snd. We write the sum of two types $A, B : \mathsf{Type}$ as $A + B$, its inhabitants are $\mathsf{inl}_{A+B}\ x$ and $\mathsf{inr}_{A+B}\ y$ for $x : A$ and $y : B$. The unit type $\top$ comes with a unique inhabitant $\mathsf{tt} : \top$.

Furthermore, we assume that our type theory has recursive types in the form of initial algebras of polynomial endofunctors [1, 24], i.e., we have a type $\mu : (\mathsf{Type} \to \mathsf{Type}) \to \mathsf{Type}$ with a single constructor $\mathsf{init}\ : F(\mu\ F) \to \mu\ F$ for any $F : \mathsf{Type} \to \mathsf{Type}$. We assume that all inductive types have standard dependent elimination principles, which we will not spell out explicitly but only apply in the guise of dependent pattern matching [15].

We will in this section embed linear logic in our host theory, where we proceed in three steps. In Section 2.1, we introduce the basic type formers and constructors that we add to our theory. We

will then describe how we can incorporate inductive types into our system in Section 2.2. Finally, we show how we can add function types in Section 2.3.

2.1 Embedding linear logic in type theory

We add additional types to our host type theory to be able to reason about resources. Our construction can be understood as a deep embedding of linear logic in dependent type theory, where we reuse the values of the host theory as the basic objects of our linear logic.

Supplies.

$$\frac{}{\Gamma \vdash \mathsf{Supply} : \mathsf{Type}} \qquad \frac{}{\Gamma \vdash \diamond : \mathsf{Supply}} \qquad \frac{\Gamma \vdash \Delta_0 : \mathsf{Supply} \qquad \Gamma \vdash \Delta_1 : \mathsf{Supply}}{\Gamma \vdash \Delta_0 \otimes \Delta_1 : \mathsf{Supply}} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \iota\, a : \mathsf{Supply}}$$

Productions. All premises and conclusions below start with $\Gamma \vdash$.

$$\frac{\Delta_0 : \mathsf{Supply} \qquad \Delta_1 : \mathsf{Supply}}{\Delta_0 \rhd \Delta_1 : \mathsf{Type}} \qquad \frac{\Delta : \mathsf{Supply}}{\mathsf{id}_\Delta : \Delta \rhd \Delta} \qquad \frac{\delta_1 : \Delta_1 \rhd \Delta_2 \qquad \delta_0 : \Delta_0 \rhd \Delta_1}{\delta_1 \circ \delta_0 : \Delta_0 \rhd \Delta_2}$$

$$\frac{\delta : \Delta_0 \rhd \Delta_1 \qquad \delta' : \Delta_0' \rhd \Delta_1'}{\delta \otimes^{\mathsf{f}} \delta' : \Delta_0 \otimes \Delta_0' \rhd \Delta_1 \otimes \Delta_1'} \qquad \frac{\Delta_0 : \mathsf{Supply} \qquad \Delta_1 : \mathsf{Supply} \qquad \Delta_2 : \mathsf{Supply}}{\mathsf{assoc}_{\Delta_0,\Delta_1,\Delta_2} : (\Delta_0 \otimes \Delta_1) \otimes \Delta_2 \rhd \Delta_0 \otimes (\Delta_1 \otimes \Delta_2)}$$

$$\frac{\Delta_0 : \mathsf{Supply} \qquad \Delta_1 : \mathsf{Supply}}{\mathsf{swap}_{\Delta_0,\Delta_1} : \Delta_0 \otimes \Delta_1 \rhd \Delta_1 \otimes \Delta_0} \qquad \frac{\Delta : \mathsf{Supply}}{\mathsf{unitr}_\Delta : \Delta \otimes \diamond \bowtie \Delta : \mathsf{unitr}'\Delta}$$

Fig. 1. Basic rules for supplies and productions.

The basic rules that will allow us to reason about resources in type theory are depicted in Figure 1. We introduce a type Supply, which gathers the resources available for a construction. There is always the empty supply $\diamond$, and we can join the resources of two supplies with $\otimes$. We can consider any value $a : A$ as a supply $\iota\, a$. We will see that the power of our system comes from the fact that really *any* term can be part of a supply, as this allows us to derive linear elimination principles for inductive types. On the other hand, the elements of a supply that we actually care about are the *variables*, which explains why we use $\Delta, \Delta_0, \ldots$ etc. for supplies, which is usually used for contexts in the type-theoretic literature. Our supplies are not contexts—variables are only introduced in the host dependent type theory—but we think of supplies as "linear contexts" that tell us which resources we are allowed to use.

Secondly, we introduce a relation between supplies $\rhd$, called the *productions*, which specifies the ways in which a supply can be turned into another supply. Any supply can be turned into itself with id, and two productions can be combined in two different ways: first, they can be composed using $\circ$, and second they can act side-by-side with $\otimes^{\mathsf{f}}$. We furthermore require the productions assoc and swap, which establish that the order of joining up resources with $\otimes$ is associative, and that $\otimes$ is symmetric. The last rule introduces two productions $\mathsf{unitr} : \Delta \otimes \diamond \rhd \Delta$ and $\mathsf{unitr}' : \Delta \rhd \Delta \otimes \diamond$ which establish that $\otimes$ is unital with respect to $\diamond$, where $\bowtie$ is shorthand notation which introduces two productions in opposite directions at once.

Remark 2.1. We will not specify substitutions for our syntax as these are completely standard—note that Supply and $\rhd$ are just types with several constructors. This means that substitutions simply factor through the newly introduced types and terms. Using an explicit substitution calculus

where substitutions γ from Γ_0 to Γ_1 are applied to types and terms as $\_[\gamma]$, we have for instance $\Gamma_1 \vdash \mathsf{Supply}[\gamma] = \mathsf{Supply} : \mathsf{Type}$ and $\Gamma_1 \vdash (\iota\, a)[\gamma] = \iota\,(a[\gamma]) : \mathsf{Supply}$ for any $\Gamma_0 \vdash a : A$. In fact, these are precisely the substitution rules that Agda also derives for us in the implementation of our theory. We will discuss substitutions in more detail when introducing the semantics in Section 5. △

Remark 2.2. The knowledgeable reader will have recognised that our supplies present a symmetric monoidal category with hom-set $\Delta_0 \rhd \Delta_1$. For simplicity, we have deviated from standard presentations of monoidal categories and omitted the inverse of assoc and a left unit law as these can be derived in the presence of swap (details can be found in the artefact). We call these derived productions assoc' and $\mathsf{unitl}/\mathsf{unitl}'$ in the following. △

Remark 2.3. For our semantics in Section 5 to work out, we need to add equalities between productions, even though these will not play a role in our constructions as we are generally only interested in the existence of *some* production, and not how different productions relate to one another. We will in this remark spell out some equalities that hold between the productions, but will in the remainder of this paper refrain from specifying more equalities as these follow immediately from the commutative diagrams of the categorical structures we are embedding.

The combined presentation of the productions for the right unit as $\Delta \otimes \diamond \bowtie \Delta$ suggests that we have $\mathsf{unitr}'_\Delta \circ \mathsf{unitr}_\Delta = \mathsf{id}_{\Delta\otimes\diamond}$ and $\mathsf{unitr}_\Delta \circ \mathsf{unitr}'_\Delta = \mathsf{id}_\Delta$. Whenever we use $\bowtie$ in the following we will assume that both productions are mutually inverse.

Production composition $\circ$ is associative and unital with respect to id. Moreover, $\otimes$ acts as a bifunctor, this entails for instance that we have $(\delta_1 \circ \delta_0) \otimes^{\mathsf{f}} (\delta'_1 \circ \delta'_0) = (\delta_1 \otimes^{\mathsf{f}} \delta'_1) \circ (\delta_0 \otimes^{\mathsf{f}} \delta'_0) : \Delta_0 \otimes \Delta'_0 \rhd \Delta_2 \otimes \Delta'_2$ for any $\delta_1 : \Delta_1 \rhd \Delta_2$, $\delta'_1 : \Delta'_1 \rhd \Delta'_2$, $\delta_0 : \Delta_0 \rhd \Delta_1$ and $\delta'_0 : \Delta'_0 \rhd \Delta'_1$.

The pentagon identity for monoidal categories establishes that it does not matter how we reassociate, which manifests as the following equality in our system.

$$(\mathsf{id}_{\Delta_0} \otimes^{\mathsf{f}} \mathsf{assoc}_{\Delta_1,\Delta_2,\Delta_3}) \circ \mathsf{assoc}_{\Delta_0,\Delta_1\otimes\Delta_2,\Delta_3} \circ (\mathsf{assoc}_{\Delta_0,\Delta_1,\Delta_2} \otimes^{\mathsf{f}} \mathsf{id}_{\Delta_3})$$
$$= \mathsf{assoc}_{\Delta_0,\Delta_1,\Delta_2\otimes\Delta_3} \circ \mathsf{assoc}_{\Delta_0\otimes\Delta_1,\Delta_2,\Delta_3} : ((\Delta_0 \otimes \Delta_1) \otimes \Delta_2) \otimes \Delta_3 \rhd \Delta_0 \otimes (\Delta_1 \otimes (\Delta_2 \otimes \Delta_3))$$

The hexagon equality mediates between swapping and reassociating.

$$(\mathsf{id}_{\Delta_1} \otimes^{\mathsf{f}} \mathsf{swap}_{\Delta_0,\Delta_2}) \circ \mathsf{assoc}_{\Delta_1,\Delta_0,\Delta_2} \circ (\mathsf{swap}_{\Delta_0,\Delta_1} \otimes^{\mathsf{f}} \mathsf{id}_{\Delta_2})$$
$$= \mathsf{assoc}_{\Delta_1,\Delta_2,\Delta_0} \circ \mathsf{swap}_{\Delta_0,\Delta_1\otimes\Delta_2} \circ \mathsf{assoc}_{\Delta_0,\Delta_1,\Delta_2} : (\Delta_0 \otimes \Delta_1) \otimes \Delta_2 \rhd \Delta_1 \otimes (\Delta_2 \otimes \Delta_0)$$

And swapping twice is the same as applying the identity production.

$$\mathsf{swap}_{\Delta_1,\Delta_0} \circ \mathsf{swap}_{\Delta_0,\Delta_1} = \mathsf{id}_{\Delta_0\otimes\Delta_1} : \Delta_0 \otimes \Delta_1 \rhd \Delta_0 \otimes \Delta_1$$

We do not have a triangle equality since we have no primitive left unital production.

△

The—still very straightforward—rules of Figure 1 already allow us to talk about resources in type theory. The basic idea behind our calculus is that we construct a term of the host type theory and at the same time view it as a resource. To this end, we introduce the notion of a *linear type*, LType for short, which is a type and a supply depending on this type.

$$\mathsf{LType} : \mathsf{Type}$$
$$\mathsf{LType} = \Sigma[\, A \in \mathsf{Type} \,]\ (A \to \mathsf{Supply})$$

Inhabitants of a linear type are hence pairs of an intuitionistic (cartesian) term and a supply which says how we should consider it as a resource. For example, if we have some type A, we can always consider the linear type $(A\ ,\ \iota)$, whose inhabitants represent a single copy of an element of

A. The generality of having arbitrary dependent supplies will become relevant in Section 2.3 when we introduce linear function types.

Using linear types we can formulate the main idea behind our calculus: the characterisation of *linear judgments* as the following dependent type:

```
_⊩_ : Supply → LType → Type
Δ ⊩ (A , Θ) = Σ[ a ∈ A ] (Δ ▷ Θ a)
```

To understand our linear judgment, let us first consider the case when the dependent supply Θ is simply ι. In this case our definition says that to derive from a supply Δ a single copy of some A, we need to provide an inhabitant $a : A$ as well as a production which shows that Δ and $\iota\ a$ are made up of the same resources.

Similarly to how we viewed supplies as "linear contexts", we understand $\Vdash$ as "linear entailment", even though it is just a usual dependent type in our host theory. We can hence think of our system as a two-step calculus "$\Gamma \vdash \Delta \Vdash$". In the following, we will sometimes abuse notation and write $\Gamma \vdash \Delta \Vdash a : (A , \Theta)$ for an element $(a , \delta) : \Delta \Vdash (A , \Theta)$ to emphasize this point of view. This notation also indicates that we usually do not care about specific productions, as long as there is some production that establishes that we have used the given resources (in fact, most of the productions will be generated automatically in the following).

To give a simple example of our two-step calculus, consider a context which contains some variable of type A and a supply containing one copy of this variable. This gives us the resources to derive A, a fact we will often need in the following (since we work in Agda, our type-theoretic entailment $\vdash$ is encoded as a dependent function type).

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

The above function can be considered the "variable rule" of our linear calculus—but of course it applies to any term of the host theory. It allows us to consider any intuitionistic term as a linear term.

Let us introduce some abuse of notation to make type signatures involving base types look nicer.

Notation. We will in the following regard our base types $A, B, \ldots$ (and combinations of these such as $A \times B$ etc.) as linear types over ι, i.e., we write A instead of (A , ι).

With this notation and the above defined "variable rule" we can already write Agda code which looks like a linear function.

```
idJ : (x : A) → ι x ⊩ A
idJ x = x⌟
```

We will see that—using some slight annotating—we can program in our system just like in any functional language. The linear judgment will ensure that we construct programs in a linear fashion. Once we have constructed a program in our type system, we can evaluate our programs as usual by projecting out the intuitionistic part of our judgment, e.g., `fst (idJ 4)` will reduce to `4`.

2.2 Adding productions to incorporate inductive types

The first addition we make to our basic syntax is to incorporate inductive data types. Intuitively, we want to regard inductive types as ways in which we can pack up resources, but not as resources themselves. This means variables of the base types $A, B \ldots$ will be considered resources, but several variables packaged up as a list constitute the same resources as having them individually in a supply.

We will integrate inductive types in the guise of least fixpoints of polynomial endofunctors, which provide an alternative formulation of W-types [1, 24, 32]. Remarkably, all we need to add to our syntax are productions which take care of the constructors of the types under consideration. The elimination principles are then derivable using elimination into our dependent type capturing linear judgements.

Units and dependent products.

$$\frac{}{\Gamma \vdash \mathsf{opl}_{\mathsf{tt}} : \iota\ \mathsf{tt} \bowtie \diamond : \mathsf{lax}_{\mathsf{tt}}} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B\ x : \mathsf{Type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B\ a}{\Gamma \vdash \mathsf{opl}, : \iota\ (a\ ,\ b) \bowtie \iota\ a \otimes \iota\ b : \mathsf{lax},}$$

Sums and fixpoints.

$$\frac{\Gamma \vdash A, B : \mathsf{Type} \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{sec}_{\mathsf{inl}} : \iota\ (\mathsf{inl}_{A+B}\ a) \bowtie \iota\ a : \mathsf{ret}_{\mathsf{inl}}} \qquad \frac{\Gamma \vdash A, B : \mathsf{Type} \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{sec}_{\mathsf{inr}} : \iota\ (\mathsf{inr}_{A+B}\ b) \bowtie \iota\ b : \mathsf{ret}_{\mathsf{inr}}}$$

$$\frac{\Gamma, A : \mathsf{Type} \vdash F\ A : \mathsf{Type} \qquad \Gamma \vdash x : F(\mu\ F)}{\mathsf{sec}_{\mathsf{init}} : \iota\ (\mathsf{init}\ x) \bowtie \iota\ x : \mathsf{ret}_{\mathsf{init}}}$$

Fig. 2. Productions for data types.

We extend our production relation from Figure 1 with the rules in Figure 2. For each constructor we give two productions (which are mutually inverse, akin to unitr and unitr′).

For the unit and dependent product type we call the productions oplax and lax, a perspective that will be spelled out in Section 5. The unit type is not considered a resource and its inhabitant can always freely be added or removed from a supply with $\mathsf{opl}_{\mathsf{tt}}$ and $\mathsf{lax}_{\mathsf{tt}}$. The productions opl, and lax, stipulate that it does not matter if we have two terms packaged as a pair, or separately in a supply.

For the sum type and fixpoint type we give productions which allow us to freely add or remove their constructor symbols, which captures that the resources represented are considered invariant no matter if we pack it up as an inhabitant of a sum, respectively fixpoint, or not. We call these productions sections and retractions, as they capture that inl, inr and init act as isomorphisms in the world of supplies.

Our production type is still simple to work with, and the following proposition will mean we do not have to worry about constructing these if they can be constructed using the rules in Figures 1 and 2 (we will introduce additional productions later). Note that the following proposition rests on us not having function symbols in our theory yet (once we introduce inductive functions and have open terms, we have an undecidable problem at hand as we will see in Section 3).

Proposition 2.1. $\Gamma \vdash \Delta_0 \rhd \Delta_1 : \mathsf{Type}$ is decidable for any supplies $\Gamma \vdash \Delta_0, \Delta_1 : \mathsf{Supply}$.

Proof. First, note that $\rhd$ is symmetric. The identity, swap, unital, lax/oplax and section/retraction productions are immediately symmetric, and associativity is so according to Remark 2.2. Production composition $\circ$ as well as $\otimes^{\mathsf{f}}$ can also easily be seen to maintain symmetry: if we have $\delta_1 \circ \delta_0 : \Delta_0 \rhd \Delta_2$, then the symmetric productions δ_0' and δ_1' given by the induction hypothesis yield $\delta_0' \circ \delta_1' : \Delta_2 \rhd \Delta_0$. A similar construction works for $\otimes^{\mathsf{f}}$, only pre- and postcomposing with appropriate swaps.

Second, we can transform any Δ into a normal form $(\iota\ x_1 \otimes (\iota\ x_2 \otimes (\ldots \otimes (\iota\ x_n \otimes \diamond))))$ where the x_i are (not necessarily distinct) variables ordered by their De Bruijn indices. To obtain this

normal form, first remove all constructor symbols using the oplax and section productions. We can then reassociate, remove all $\diamond$ (possibly introducing the last with unitr′) and reorder using swap to obtain this normal form. □

With later extensions to the productions and supplies, the production type is not as easy to decide anymore. However, it turns out that we get surprisingly far with Proposition 2.1 since more complex resource usage—such as consumption of some resource by a function—is usually done explicitly by the user. We will in the following omit any production that can be derived according to Proposition 2.1 (we have implemented an Agda tactic which does exactly this for us, the interested reader can inspect the tactic and the productions it produces in the artefact).

For example, we can devise a program which swaps two elements of a pair as follows.

```
switchJ : (z : A × B) ⟶ ι z ⊩ B × A
switchJ (x , y) = (y , x)ι
```

When we pattern match on the given element of type $A \times B$, the codomain of switch becomes $\iota\ (x\ ,\ y) \Vdash (B\ ,\ A)$. We return an element $(y\ ,\ x)_\iota : \iota\ (y\ ,\ x) \Vdash B \times A$ and let our tactic construct the appropriate production for us, which in this case is $\text{lax}, \circ \text{swap}_{\iota\ x, \iota\ y} \circ \text{opl},$.

Our linear judgment forces us to use exactly the provided resources. Consider the following program which aims to copy a given x.

```
impossible : (x : A) ⟶ ι x ⊩ A × A
impossible x = (x , x) , {! Goal: ι x ▷ ι (x , x)!}
```

We can copy the given x for the intuitionistic part of our judgment $\Vdash$, but we then need to derive a production which turns a single copy of x into two copies, which does not exist (in fact, if we were to apply our Agda tactic, we would get a typing error that $\iota\ x \otimes \diamond\ != \diamond$).

In switchJ we have already tacitly utilised the power of dependent elimination, let us look at another example how elimination into $\Vdash$ gives us linear elimination principles for data types. Consider a Maybe A type, which we define as $A + \top$. Writing just x for inl x and nothing for inr tt, we can have a first attempt at a linear fold for lists as follows.

```
foldMaybe-ish : (z : Maybe A) ⟶ ((x : A) ⟶ ι x ⊩ B ) ⟶ (n : B)
  ⟶ ι z ⊗ (if isJust z then ◇ else ι n) ⊩ B
foldMaybe-ish (just x) j n = j x
foldMaybe-ish nothing j n = nι
```

We are constructing a dependent function which has three arguments, a value z of Maybe A, a dependent function which we can think of as some sort of linear function, and a backup value n of type B. By pattern matching on the given z, the return type turns into $\iota\ (\text{just}\ x) \otimes \diamond \Vdash B$ and $\iota\ \text{nothing} \otimes \iota\ z \Vdash B$, respectively—which means that in each case, we really only have available either the value inside the Maybe A or the backup element, but not both.

Therefore, dependent type theory offers a powerful framework to deal with resources and we can already dynamically compute resources in our linear judgments. However, this is still unsatisfying since foldMaybe−ish is not a proper higher-order linear function. The given j function is just a dependent function returning a linear judgment and not a proper linear function, and we want to specify how often some variable ought to be used as soon as we introduce it. We address the first point right away with the introduction of function types, and will setup our framework for function types with dependent multiplicities in the next Section 3.

2.3 Adding linear dependent function types

We have seen that our embedding of linear logic allowed us to obtain a linear calculus inside dependent type theory, which can be understood as constructing terms in a two-step calculus $\Gamma \vdash \Delta \Vdash a : A$. We will now internalise this two-step entailment and thereby obtain dependent linear function types, which we write as $\langle\, x : A \,\rangle \multimap B\ x$ to distinguish them from the dependent functions of the host theory. In general, the domain and codomain for linear functions will be LTypes, i.e., pairs of a intuitionistic type A and a dependent supply $\Theta : A \to \mathsf{Supply}$, but for now let us focus on the case where Θ is just ι. Using the intuition of our calculus as a two-step intuitionistic/linear calculus, we would expect linear function types to have an introduction and application rule as follows, where we write $\boldsymbol{\lambda}\ x \mapsto b$ for linear function abstraction and $f\ @\ a$ for linear function application.

$$\frac{\Gamma \vdash \Delta : \mathsf{Supply} \qquad \Gamma, x : A \vdash \Delta \otimes \iota\ x \Vdash b : B\ x}{\Gamma \vdash \Delta \Vdash \boldsymbol{\lambda}\ x \mapsto b : \langle\, x : A \,\rangle \multimap B} \qquad \frac{\Gamma \vdash \Delta_0 \Vdash f : \langle\, x : A \,\rangle \multimap B\ x \qquad \Gamma \vdash \Delta_1 \Vdash a : A}{\Gamma \vdash \Delta_0 \otimes \Delta_1 \Vdash f\ @\ a : B\ a}$$

To introduce a linear dependent function from A to a type B depending on A, we need to have derived an inhabitant of $b : B\ x$ in a context extended with a variable $x : A$ and a supply containing a single copy of x and some other supply Δ not mentioning x. The conclusion will live in a context and supply which do not mention x anymore. Conversely, when applying a linear function f from A to B to some inhabitant a of A, we expect that the result will use the combined resources that were used to derive f and a.

Remarkably, both of these rules are admissible in our type system if we add only a little more structure to our embedded linear logic introduced in Figure 1 and 2. Moreover, all the generalisations of this function type—taking arbitrary Θ and not ι, dependent functions with multiplicities as introduced in Section 3.1, and unrestricted functions as introduced in Section 4—are derivable from the structure given in this section. Since our linear judgments live as dependent types in our host theory, we can moreoever directly implement these admissible rules, and using some handy notation and the tactic devised in Proposition 2.1, we will be able to program in our linear type system just like in any functional language.

Internal productions. All premises and conclusions below start with $\Gamma \vdash$.

$$\frac{\Delta_0 : \mathsf{Supply} \qquad \Gamma \vdash \Delta_1 : \mathsf{Supply}}{[\ \Delta_0, \Delta_1\ ] : \mathsf{Supply}} \qquad \frac{\delta : \Delta_0 \otimes \Delta_1 \rhd \Delta_2}{\mathsf{curry}\ \delta : \Delta_0 \rhd [\ \Delta_1, \Delta_2\ ]} \qquad \frac{\delta : \Delta_0 \rhd [\ \Delta_1, \Delta_2\ ]}{\mathsf{uncurry}\ \delta : \Delta_0 \otimes \Delta_1 \rhd \Delta_2}$$

Supply abstraction. Assume $\Gamma \vdash A : \mathsf{Type}$, $\Gamma, x : A \vdash \Theta\ x : \mathsf{Supply}$ and $\Gamma \vdash \Delta : \mathsf{Supply}$.

$$\frac{\Gamma, x : A \vdash \Theta\ x : \mathsf{Supply}}{\Gamma \vdash \Lambda_{x:A}\ \Theta\ x : \mathsf{Supply}} \qquad \frac{\Gamma, x : A \vdash \delta : \Delta \rhd \Theta\ x}{\Gamma \vdash \mathsf{bind}_x\ \delta : \Delta \rhd \Lambda_{x:A}\ \Theta\ x} \qquad \frac{\Gamma \vdash \delta : \Delta \rhd \Lambda_{x:A}\ \Theta\ x}{\Gamma, x : A \vdash \mathsf{free}_x\ \delta : \Delta \rhd \Theta\ x}$$

Fig. 3. Rules for function types.

In Figure 3 we introduce two sets of rules. First, we add a way to internalise a production $\Delta_0 \rhd \Delta_1$ as a supply $[\ \Delta_0, \Delta_1\ ]$. We can introduce internalised productions with currying, and discard by uncurrying. As we will spell out in Section 5, internal productions correspond precisely to *closing* the symmetric monoidal structure that we have embedded in the host theory.

The second set of rules is crucial to making our embedded linear logic work together with the host type theory. We add a variable binding principle for supplies, i.e., if we have a supply $\Theta\ x$ living in a supply extended with some variable x of type A, we can bind x with Λ, i.e., $\Lambda_{x:A}\ \Theta\ x$

lives in context Γ. We will work out in the semantics in Section 5 that Λ corresponds precisely to a right-adjoint to the action on supplies of context extension, which is the central structural rule of dependent type theory. Consequently, we have an isomorphism between hom-sets which manifests itself in our syntax as the bind and free productions. If we have in context $\Gamma, x : A$ a production δ which turns the supply Δ—which does not contain x—into $\Theta\ x$, we can bind variable x on the right-hand side of the production with Λ and thereby obtain a production living in Γ. Conversely, we can remove the variable binding on the right-hand side by applying free.

We will now use the structure introduced in Figure 3 to characterise function types in our type theory. Just like our LTypes were pairs of an intuitionistic type and a supply specifying how inhabitants of the type can be viewed as a resource, our dependent linear functions will be pairs of an intuitionistic function f and an (internalised) production that specifies how f acts on resources. Let us write $A \Rightarrow \mathsf{Type}$ for $\Sigma[\ B \in A \to \mathsf{Type}\ ]\ ((x : A) \to B\ x \to \mathsf{Supply})$, which means an element of $A \Rightarrow \mathsf{Type}$ is a linear type depending on A. This allows us to give the following definition of function types in our system.

$$\langle \_ \rangle \multimap \_ : (A : \mathsf{LType}) \to (A \Rightarrow \mathsf{LType}) \to \mathsf{LType}$$
$$\langle\ A\ ,\ \Theta_0\ \rangle \multimap (B\ ,\ \Theta_1) = ((x : A) \to B\ x)\ ,\ \lambda\ f \to \Lambda_{x:A}\ [\ \Theta_0\ x\ ,\ \Theta_1\ (f\ x)\ ]$$

A function from linear type A to B depending on A (where in the definition we refer with A and B only to their underlying intuitionistic types), is a dependent function f from A to B as well as a supply which establishes that for any input $x : A$, the resources $\Theta_0\ x$ can be turned into $\Theta_1\ (f\ x)$. Note that the dependencies are quite subtle: the supply of the linear type $\langle\ A\ ,\ \Theta_0\ \rangle \multimap (B\ ,\ \Theta_1)$ depends not on A, but on a function f of type $(x : A) \to B\ x$ since the x is bound in the supply using Λ.

The following convenient notation is also part of our artefact using Agda's syntax feature.

Notation. We write $\langle\ x : A\ \rangle \multimap B\ x$ instead of $\langle\ A\ \rangle \multimap \lambda x.B\ x$.

Using the productions for currying and binding a variable, we can derive (a more general version of) the function introduction rule motivated above. Again, Δ is a supply not mentioning x, while we now generalise ι to arbitrary dependent supplies.

$$\Pi\mathsf{I} : ((x : A) \to \Delta \otimes \Theta_0\ x \Vdash B\ x\ ,\ \Theta_1) \to \Delta \Vdash \langle\ x : A\ ,\ \Theta_0\ \rangle \multimap B\ x\ ,\ \Theta_1$$
$$\Pi\mathsf{I}\ f\delta = (\lambda\ x \to \mathsf{fst}\ (f\delta\ x))\ ,\ \mathsf{bind}_x\ (\mathsf{curry}\ (\mathsf{snd}\ (f\delta\ x)))$$

To construct a linear function we have to give some $f\delta$ which establishes that for any $x : A$ we have both an element of $B\ x$ and a production which turns $\Delta \otimes \Theta_0\ x$ into $\Theta_1\ (f\ x)$. The first part gives us the required intuitionistic function $f : (x : A) \to B$, the secondly given production only needs to be curried and it's variable bound to construct the necessary production from Δ to $\Lambda_{x:A}\ [\ \Theta_0\ x, \Theta_1\ (f\ x)\ ]$.

Consequently, in order to construct a linear function, we have to apply $\Pi\mathsf{I}$ and then construct a usual dependent function. Let us again introduce special syntax which makes this convenient.

Notation. We write $\boldsymbol{\lambda}\ x \mapsto b$ for $\Pi\mathsf{I}\ (\lambda x.b)$.

We can derive the linear function application principle as follows.

$$\_@\_ : \Delta_0 \Vdash \langle\ x : A\ ,\ \Theta_0\ \rangle \multimap B\ x\ ,\ \Theta_1 \to ((a\ ,\ \_) : \Delta_1 \Vdash A\ ,\ \Theta_0) \to (\Delta_0 \otimes \Delta_1 \Vdash B\ a\ ,\ \Theta_1)$$
$$(f\ ,\ \delta_0)\ @\ (a\ ,\ \delta_1) = f\ a\ ,\ \mathsf{uncurry}\ (\mathsf{free}_a\ \delta_0) \circ (\mathsf{id}\ \Delta_0 \otimes^f \delta_1)$$

The first linear judgment gives us an intuitionistic function $f : (x : A) \to B\ x$ and a production δ_0 which turns Δ_0 into $\Lambda_{x:A}\ [\ \Theta\ x, \Theta\ (f\ x)\ ]$. We can remove Λ with free and uncurry to get a production from $\Delta_0 \otimes \Theta\ a$ to $\Theta_1\ (f\ a)$. Combining this with the production δ_1 which turned Δ_1 into $\Theta_0\ a$ completes our construction.

Having derived introduction and application principles, we can program in our system just like in any functional language. With the rules added in Figure 3, it is not as straightforward to decide the production type anymore, however, it turns out we can still use Proposition 2.1, where we only dealt with the productions from Figures 1 and 2, since all applications of curry, uncurry, bind and free appear when we introduce or eliminate a linear function as defined above, and are hence implicitly done by the user. We can hence continue to use our tactic for deciding productions, whose notion of normal forms only has to be slightly extended to account for the additional supplies $[\,-,-\,]$ and Λ, and reorder these as necessary.

For a start, let us show that the pair constructor gives rise to a linear function, where we simply return $(x\ ,\ y)$ regarded as a linear turn and let our production solver convince the type-checker that we work linearly.

$$\mathsf{pair} : \diamond \Vdash \langle\ x : A\ \rangle \multimap \langle\ y : B\ x\ \rangle \multimap \Sigma\ A\ B$$
$$\mathsf{pair} = \boldsymbol{\lambda}\ x \mapsto \boldsymbol{\lambda}\ y \mapsto (x\ ,\ y)_\iota$$

Note that pair is a dependent linear higher-order function, since for any input of type A we return the linear function form $B\ x$ to $\Sigma\ A\ B$.

We can reimplement switchJ from Section 2.2 now as a proper linear function by applying pair to the two given elements.

$$\mathsf{switch} : \diamond \Vdash \langle\ A \times B\ \rangle \multimap B \times A$$
$$\mathsf{switch} = \boldsymbol{\lambda}\ (x\ ,\ y) \mapsto \mathsf{pair}\ @\ y_\iota\ @\ x_\iota$$

We can straightforwardly work with functions which take in functions as arguments. For example, we can curry a dependent linear function exactly as we would expect, where we write $\_^\wedge$ to turn a function into the corresponding identity judgment, analogously to $\__\iota$ but for dependent supplies $^\wedge$ more general than ι.

$$\mathsf{curry} : \diamond \Vdash \langle\ \langle\ (x\ ,\ y) : \Sigma\ A\ B\ \rangle \multimap C\ x\ y\ \rangle \multimap \langle\ x : A\ \rangle \multimap \langle\ y : B\ x\ \rangle \multimap C\ x\ y$$
$$\mathsf{curry} = \boldsymbol{\lambda}\ f \mapsto \boldsymbol{\lambda}\ x \mapsto \boldsymbol{\lambda}\ y \mapsto f_\Theta\ @\ (\mathsf{pair}\ @\ x_\iota\ @\ y_\iota)$$

The production that is necessary above identifies supplies that both contain some supply $\Lambda_{A\times B}\ [\ldots]$ corresponding to the linear function f, but involves no currying or binding and can hence be straightforwardly derived using the procedure from Proposition 2.1.

Similarly, we can uncurry a higher-order function f conveniently in our setting.

$$\mathsf{uncurry} : \diamond \Vdash \langle\ \langle\ x : A\ \rangle \multimap \langle\ y : B\ x\ \rangle \multimap C\ x\ y\ \rangle \multimap \langle\ (x\ ,\ y) : \Sigma\ A\ B\ \rangle \multimap C\ x\ y$$
$$\mathsf{uncurry} = \boldsymbol{\lambda}\ f \mapsto \boldsymbol{\lambda}\ (x\ ,\ y) \mapsto f_\Theta\ @\ x_\iota\ @\ y_\iota$$

Thereby we have constructed a dependent linear type system by embedding linear logic in dependent type theory. Most of the structure we had to impose is motivated by linear logic, we only had to give some additional rules which govern the interplay of the linear logic with the host type theory.

3 Quantitative Types with Dependent Multiplicities

We now have at hand a dependent linear type theory—and it turns out to be a very powerful one. Since supplies are values of some type in the host theory, we can compute with them! For example, we can use the natural numbers $\mathbb{N}$ to create copies of some supply, which means we can specify not only that some variable is used once, but m-many times for some $m : \mathbb{N}$. We thereby have at hand a quantitative system, let us call the number of times a variable ought to be used its *multiplicity*. In contrast to existing systems such as Quantitative type theory [5] and Graded Modal type theory [37], our multiplicities do not have to be static, but can be open terms which *depend*

on earlier variables. This allows us to precisely annotate the resource usage of many higher-order functions for the first time.

Notably, we do not have to make any additions to the syntax introduced in the last Section 2, but work entirely within the theory we already defined. We will give a simple recursive definition to take m copies of some supply in Section 3.1, which forms the basis of our quantitative type theory. In particular, we can define function types which consume some multiplicity of their input in a similar fashion as we did in the linear case. We will then reap the fruits of our efforts and program with higher-order functions in Section 3.2, giving for the first time linear types for some programs that cannot precisely be typed in any other setting.

The increase in power of our type system comes inevitably with a price—equality between open terms of the natural numbers is undecidable, which also means that our typing relation cannot be decidable anymore. We will see in Section 3.3 that there are natural programs for which our tactic deciding productions does not work any more. Not all is lost, however: we can still do manual work to show that some function uses the specified resources.

3.1 Adding quantities to our system with a natural resource algebra

We use a standard natural numbers type to introduce quantities in our type system. Let us define $\mathbb{N}$ as the least fixpoint of the functor $X \mapsto \top + X$, and define zero as init (inl tt) and suc n as init (inr n). Since our supplies are themselves standard types in the host theory, we can compute with them, which allows us to copy a given supply m-many times for any $m : \mathbb{N}$.

$$\_\,\hat{}\,\_ : \mathsf{Supply} \to \mathbb{N} \to \mathsf{Supply}$$
$$\Delta\ \hat{}\ \mathsf{zero} = \diamond$$
$$\Delta\ \hat{}\ (\mathsf{suc}\ m) = \Delta \otimes (\Delta\ \hat{}\ m)$$

This simple recursive definition gives rise to quantitative features in our linear type system. For example, we can now conveniently type a program which copies a given element.

$$\mathsf{copyJ} : (x : A) \to \iota\, x\ \hat{}\ 2 \Vdash A \times A$$
$$\mathsf{copyJ}\ x = (x\ ,\ x)_\iota$$

Note that $\iota\, x\ \hat{}\ 2$ reduces simply to $\iota\, x \otimes \iota\, x$, which means we could have written a definitionally equal version to copyJ already with our system from Section 2. However, the multiplicity given to $\hat{}$ need not be in canonical form, and we will see that this is where the power of our quantitative system comes from.

We can internalise our two-step intuitionistic linear calculus in very much the same way as we introduced linear functions in Section 2.3, the only difference being that we now include a natural number parameter in our function types.

$$\langle \_ \rangle\hat{}\ \_ \multimap \_ : (A : \mathsf{LType}) \to \mathbb{N} \to (A \Rightarrow \mathsf{LType}) \to \mathsf{LType}$$
$$\langle\ A\ ,\ \Theta_0\ \rangle\hat{}\ m \multimap (B\ ,\ \Theta_1) = ((x : A) \to B\ x)\ ,\ \lambda\ f \to \Lambda_{x:A}\,[\ \Theta_0\ x\ \hat{}\ m\ ,\ \Theta_1\ (f\ x)\ ]$$

In the internalised production we require a witness that m copies of the input can be turned into one output. In case m is 1, we omit $\hat{}$ from the function type and continue to write $\langle\ x : A\ \rangle \multimap B$.

Abstraction for quantitative functions is of the following form.

$$\Pi\hat{}\mathsf{I} : ((x : A) \to \Delta \otimes \Theta_0\ x\ \hat{}\ m \Vdash B\ x\ ,\ \Theta_1) \to \Delta \Vdash \langle\ x : A\ ,\ \Theta_0\ \rangle\hat{}\ m \multimap B\ x\ ,\ \Theta_1$$

The implementation of abstraction is exactly the same as for linear functions, as we are already given the necessary production which shows that m copies of $\Theta\ x$ are turned into one output.

Notation. We will use the same syntax $\boldsymbol{\lambda}\ x \mapsto b$ as we did for linear functions to derive functions with $\Pi\hat{}\mathsf{I}\ (\lambda x.b)$.

We can thereby internalise our copying judgment from above as follows.

$$\mathsf{copy} : \diamond \Vdash \langle\ A\ \rangle^{\wedge}\ 2 \multimap A \times A$$
$$\mathsf{copy} = \boldsymbol{\lambda}\ x \mapsto (x\ ,\ x)_{\iota}$$

To define function application, we have to do only slightly more work. When applying a function with multiplicity to some term a, we will need to have m copies of a available, which is reflected in the following type.

$$\_@\_ : \Delta_0 \Vdash \langle\ x : A\ ,\ \Theta_0\ \rangle^{\wedge}\ m \multimap B\ x\ ,\ \Theta_1 \to ((a\ ,\ \_) : \Delta_1 \Vdash A\ ,\ \Theta_0) \to (\Delta_0 \otimes \Delta_1\ ^{\wedge}\ m \Vdash B\ a\ ,\ \Theta_1)$$
$$(f\ ,\ \delta_0)\ @\ (a\ ,\ \delta_1) = f\ a\ ,\ \mathsf{uncurry}\ (\mathsf{free}_a\ \delta_0) \circ (\mathsf{id}\ \Delta_0 \otimes^{f} (\triangleright^{\wedge}\ m\ \delta_1))$$

The only difference to the definition of function application for linear functions is in the use of $\triangleright^{\wedge}$, which applies the given $\delta_1 : \Delta_1 \triangleright \Theta_0\ m$ times to obtain a production from $\Delta_1\ ^{\wedge}\ m$ to $\Theta_0\ x\ ^{\wedge}\ m$. The lemma $\triangleright^{\wedge}$ is defined by a simple induction, the interested can find details in the artefact.

Agda's type-checker needs some guidance when applying a quantitative function, and for us it will also be helpful to know how many inputs some function consumes, which is why we introduce the following syntax.

Notation. We write $f \prec m\ @\ a$ for applying a function $f : \langle\ x : A\ \rangle^{\wedge}\ m \multimap B$ to some element $a : B$ (leaving m only implicit in case it is 1).

We thereby have everything at hand to work with functions that have natural number multiplicities. For example, we can create the four-fold product of a given term by applying copy twice.

$$\mathsf{copytwice} : \diamond \Vdash \langle\ x : A\ \rangle^{\wedge}\ 4 \multimap (A \times A) \times (A \times A)$$
$$\mathsf{copytwice} = \boldsymbol{\lambda}\ x \mapsto \mathsf{copy} \prec 2\ @\ (\mathsf{copy} \prec 2\ @\ x_{\iota})$$

Above, we can still use the tactic devised in Proposition 2.1 since by the time we applied both functions, the normal forms of the supplies that we have to turn into each other with a production do not contain $^{\wedge}$ anymore.

Our multiplicities also allow us to capture if a function is considered to *not* consume an input, which is useful when a variable is only relevant for the type signature. Consider function composition between two dependent functions, where the firstly applied f is between A to and B depending on A; and the secondly applied g goes to C depending on $B\ x$ for any $x : A$.

$$\mathsf{compose} : \diamond \Vdash \langle\ g : \langle\ x : A\ \rangle^{\wedge}\ 0 \multimap \langle\ y : B\ x\ \rangle \multimap C\ x\ y\ \rangle \multimap \langle\ f : \langle\ x : A\ \rangle \multimap B\ x\ \rangle$$
$$\multimap \langle\ x : A\ \rangle \multimap C\ x\ (f\ x)$$
$$\mathsf{compose} = \boldsymbol{\lambda}\ g \mapsto \boldsymbol{\lambda}\ f \mapsto \boldsymbol{\lambda}\ x \mapsto (g_{\Theta} \prec 0\ @\ x_{\iota})\ @\ (f_{\Theta}\ @\ x_{\iota})$$

Since g does not consume the given $x : A$, but only the $B\ x$ produced by applying f to x, the composition of both functions only needs a single copy of x.

3.2 Implementing linear higher-order functions with dependent multiplicities

The crucial feature of our quantitative type system lies in the fact that we can give multiplicities which mention variables introduced previously, i.e., the multiplicity of some variable can *depend* on another variable. This allows us to type a large class of higher-order functions that are impossible to precisely type with static quantities.

As a first example, consider the if-then-else function which returns one of two arguments depending on some condition. For the condition, we define Bool as $\top + \top$ and write true for inl tt and false for inr tt. Moreover, we use negation of bools $\neg$ and $|\ b\ |$ to treat b : Bool as a natural number. We can then give an appropriate type to a linear if-then-else as follows.

```
ifthenelse : ◇ ⊩ ⟨ b : Bool ⟩^ 0 ⊸ ⟨ A ⟩^ | b | ⊸ ⟨ A ⟩^ | ¬ b | ⊸ A
ifthenelse = λ b ↦ case b of
  true → λ x ↦ λ _ ↦ x_ι
  false → λ _ ↦ λ y ↦ y_ι
```

The condition b is treated irrelevantly—in fact, it does not matter which multiplicity we assign to b as it consists only of terms which can be discarded with the appropriate productions for the unit and coproduct type. The first input of type A has multiplicity 1 only in case b is true, in which case the second input has multiplicity 0, and vice versa, which means that in each case of the pattern matching on b we really only have available one of both inputs.

In a similar fashion we can implement our foldMaybe function that served as initial motivation in Section 1 and was only sloppily implemented as foldMaybe–ish in Section 2.2. We make use of functions isJust and isNothing which indicate the shape of the given z.

```
foldMaybe : ◇ ⊩ ⟨ z : Maybe A ⟩ ⊸ ⟨ ⟨ A ⟩ ⊸ B ⟩^ | isJust z | ⊸ ⟨ B ⟩^ | isNothing z | ⊸ B
foldMaybe = λ z ↦ case z of
  (just x) → λ j ↦ λ _ ↦ j_Θ @ x_ι
  nothing → λ _ ↦ λ n ↦ n_ι
```

In our definition we have a linear function j, which however is only available if there is an x that we can apply j to. Similarly, the base case n will only be available if we are given nothing.

Our approach also works seamlessly for recursive types such as lists, which for a given type A is defined as the least fixpoint of $X \mapsto \top + (A \times X)$. Let us write nil for init (inl tt) and cons x xs for init (inr (x , xs)). Consider the map on lists, which applies the given function to each element of the list. By using a standard length function we can compute the length of the list and type our program correctly.

```
map : ◇ ⊩ ⟨ xs : List A ⟩ ⊸ ⟨ f : ⟨ x : A ⟩ ⊸ B ⟩^ length xs ⊸ List B
map = λ xs ↦ case xs of
  nil → λ f ↦ nil_ι
  (cons x xs) → λ f ↦ cons @ (f_Θ @ x_ι) @ (map @ xs_ι ≺ length xs @ f_Θ)
```

By performing pattern matching on the given list xs, we know whether we have available some copy of the function f. In case we have not, we can just return nil and are done. In the recursive we use cons, which is the constructor cons considered as a linear function in two arguments. The head of the list will be the result of applying f to x, which makes use of the copy of f we know exists since length (cons x xs) reduces to suc n for some n. We then call map recursively on the rest of the list, using as many copies of the function f as the rest of the list is long. Since Agda's type checker reduces the types appropriately, we can continue to use our tactic implementing Proposition 2.1 to derive the production for us.

3.3 Undecidability of our type system

We have seen that even with recursive data types, our tactic implementing Proposition 2.1 can resolve many productions—as soon as all variables that some multiplicity depends on are appropriately instantiated, dependent multiplicities reduce and we know, e.g., that we have at least one copy of some variable. However, there are naturally occuring programs where we cannot instantiate the multiplicities since we want to stay fully parametric. Our tactic will then not be able to resolve productions, which is expected in a type system which requires checking equality of open terms of the natural numbers, an undecidable problem. However, not all is lost: since our type system lives

first-class in the host theory, the productions can still be constructed manually to establish that some program uses the given resources.

Consider the following generalised function composition, which is parametrised over $m, n : \mathbb{N}$ and composes two functions g and f, where f requires n copies of the input of type A, while g requires m copies of $B\ x$, treating $x : A$ again irrelevantly. We would expect the resulting function to require m times n many inputs, and indeed we can construct such a function.

$$\begin{aligned}&\text{compose'} : \diamond \Vdash \langle\ g : \langle\ x : A\ \rangle^{\wedge}\ 0 \multimap \langle\ y : B\ x\ \rangle^{\wedge}\ m \multimap C\ x\ y\ \rangle \multimap \\ &\quad \langle\ f : \langle\ x : A\ \rangle^{\wedge}\ n \multimap B\ x\ \rangle^{\wedge}\ m \multimap \langle\ x : A\ \rangle^{\wedge}\ (m\ ^{*}\ n) \multimap C\ x\ (f\ x) \\ &\text{compose'} = \boldsymbol{\lambda}\ g \mapsto \boldsymbol{\lambda}\ f \mapsto \boldsymbol{\lambda}\ x \mapsto (g_{\Theta} \prec 0\ @\ x_{\iota}) \prec m\ @\ (f_{\Theta} \prec n\ @\ x_{\iota}) \\ &\quad \text{by unitl}\ \_ \otimes^{f} (\otimes^{\wedge}\text{-distr}\ m\ n) \circ \text{assoc'}\ \_\ \_\ \_\end{aligned}$$

We have used a function by above to give a production which convinces the type-checker that our program uses the specified resources, i.e., by takes a linear judgment $(a\ ,\ \delta_0) : \Delta_0 \Vdash A$ and a production $\delta_1 : \Delta_1 \rhd \Delta_0$ and returns $(a\ ,\ \delta_0\ \circ\ \delta_1) : \Delta_1 \Vdash A$. The production we have constructed does some simple production yoga in order to apply the following lemma.

$$\otimes^{\wedge}\text{-distr} : (m\ n : \mathbb{N}) \to \Delta_0\ ^{\wedge}\ m \otimes \Delta_1\ ^{\wedge}\ (m\ ^{*}\ n) \rhd (\Delta_0 \otimes \Delta_1\ ^{\wedge}\ n)\ ^{\wedge}\ m$$

The proof of $\otimes^{\wedge}-$distr is not difficult, but requires several inductions on the given natural numbers. The interested reader can check the artefact for details.

4 Allowing unrestricted use of variables in our system

We have seen that our system offers a very detailed language for annotating resource usage such that we can precisely type many higher-order functions. However, there are still cases where we might not be able to give precise resource annotations—e.g., if we have mutual dependencies between several input variables—or if we simply do not care about how often some variable is used. For this purpose, linear logic offers the ! (bang) operator to annotate some variable which can be freely be duplicated or dropped . We can use this very idea to also equip our dependent linear type theory with a way to annotate some variable as unrestricted. In fact, we impose exactly the same structure as is done standardly in linear logic [26], this time only for supplies and productions.

Linear exponentials. All premises and conclusions start with $\Gamma \vdash$. Assume $\Gamma \vdash \Delta, \Delta_0, \Delta_1 : \text{Supply}$.

$$\frac{}{!\ \Delta : \text{Supply}} \qquad \frac{\delta : \Delta_0 \rhd \Delta_1}{!^{f}\ \delta : !\ \Delta_0 \rhd !\ \Delta_1} \qquad \frac{}{\text{dupl}_{\Delta} : !\ \Delta \rhd !\ (\Delta \otimes \Delta)} \qquad \frac{}{\text{erase}_{\Delta} : !\ \Delta \rhd \diamond}$$

$$\frac{}{\text{use}_{\Delta} : !\ \Delta \rhd \Delta} \qquad \frac{}{\text{mult}_{\Delta} : !\ \Delta \rhd !\ (!\ \Delta)} \qquad \frac{}{\text{coh}\diamond : \diamond \rhd !\ \diamond} \qquad \frac{}{\text{coh}\otimes_{\Delta_0,\Delta_1} : !\ \Delta_0 \otimes !\ \Delta_1 \rhd !\ (\Delta_0 \otimes \Delta_1)}$$

Fig. 4. Rules for the linear exponential.

The rules in Figure 4 introduce a linear exponential comonad ! in our theory. We can annotate any supply with !, and the functorial action $!^{f}$ allows us to turn any production into a production between supplies annotated with !. We have two productions that capture contraction and weakening for supplies annotated with !, we call these dupl and erase, respectively. The transformations use and mult allow us to make use of something annotated with !, and to introduce additional !'s as necessary. Lastly, the productions coh$\diamond$ and coh$\otimes$ capture that we can always work with the empty supply in an unprincipled way, and join two supplies which are annotated with !.

Remark 4.1. While we have extended our tactic described in Proposition 2.1 to also take into account supplies annotated with ! when comparing normal forms, we have not extended it to construct the productions introduced in Figure 4 for us, and will instead derive these manually. It would not be difficult to do so, but we think using the productions explicitly is instructive in this section, and a more mature implementation of our type theory would probably not construct ! productions, but just remove supplies annotated with ! from resource considerations altogether. △

We can now treat supplies as irrelevant, which allows us to freely use variables. For example, we can now copy a given $x : A$ if it is annotated with !.

```
copyJ : (x : A) → ! (ι x) ⊩ A × A
copyJ x = (x , x)ι by lax, ∘ (use ∘ dupl)
```

We convince the type-checker of the validity of our program my duplicating and then using the given x. Conversely, we can drop a variable annotated with ! and just return the unit inhabitant.

```
drop : (x : A) → ! (ι x) ⊩ ⊤
drop x = tt*ι by laxtt ∘ erase
```

We can internalise derivations which use a supply annotated with ! in the same fashion we internalised linear and quantitative functions, for this we introduce another special function type.

```
!⟨_⟩⊸_ : (A : LType) → (A ⇒ LType) → LType
!⟨ A , Θ₀ ⟩⊸ (B , Θ₁) = ((x : A) → B x) , λ f → Λ_{x:A} [ ! (Θ₀ x) , Θ₁ (f x) ]
```

Function introduction can be derived in exactly the same fashion as before.

```
Π!I : ((x : A) → Δ ⊗ ! (Θ₀ x) ⊩ B x , Θ₁) → Δ ⊩ !⟨ x : A , Θ₀ ⟩ ⊸ B x , Θ₁
```

Notation. We write $x\ !\!\mapsto b$ when we derive a ! function as $\Pi!\mathsf{I}\ (\lambda x.b)$.

When applying a function taking in a variable annotated with ! to some value constructed from some supply Δ_1, this supply might have been used arbitrarily many times by the function, which is why the resulting linear judgment annotates Δ_1 with ! . We derive this principle analogously to application of functions with natural number multiplicities, now making using of the functorial action of !.

```
_!@_ : Δ₀ ⊩ !⟨ x : A , Θ₀ ⟩ ⊸ B x , Θ₁ → ((a , _) : Δ₁ ⊩ A , Θ₀) → (Δ₀ ⊗ ! Δ₁ ⊩ B a , Θ₁)
(f , δ₀) !@ (a , δ₁) = f a , uncurry (free_a δ₀) ∘ (id Δ₀ ⊗^f !^f δ₁)
```

Using our new function types, we can internalise copyJ from above.

```
copy : ◇ ⊩ !⟨ A ⟩ ⊸ A × A
copy = λ x !↦ copyJ x
```

Note that the productions constructed for copyJ where enough for our tactic from Proposition 2.1 to complete the job and convince the type-checker that we have used the right resources.

When applying a function twice, we obtain a supply which has nested !'s, which provides us with an opportunity to apply mult.

```
copytwice : ◇ ⊩ !⟨ A ⟩ ⊸ (A × A) × (A × A)
copytwice = λ x !↦ copy !@ (copy !@ x_ι)
  by unitl' (! (◇ ⊗ ! (ι x))) ∘ !^f (unitl' (! (ι x))) ∘ mult ∘ unitl (! (ι x))
```

We can combine our unprincipled ! functions with linear or multiplicative ones, for example to devise a mapping function for Maybe which does not care about the use of the mapped function, but that the given Maybe A is used exactly once.

```
mapMaybe : ◇ ⊩ !⟨ ⟨ A ⟩ ⊸ B ⟩ ⊸ ⟨ Maybe A ⟩ ⊸ Maybe B
mapMaybe = λ f !↦ λ x ↦ case x of
  (just x) → just @ (f_Θ @ x_ι)
  nothing → nothing_ι
```

We introduce the function with the abstraction principle for ! functions, and x with the usual linear abstraction. By pattern matching on x we know whether we have any value available that we can apply f to and then package up again with just, which is a linear version of just. Note that the application of f to x is just linear application, since f was a linear function. The production necessary here can be derived automatically since it does not require any of the productions introduced in Figure 4. In contrast, the production in the nothing case does require the application of some production from Figure 4, namely erase, but once f is erased from the supply the remaining production is also found by our tactic. The interested reader can consult the artefact for details.

5 Semantics for dependent linear type theory

We will now specify the semantics of the rules we introduced in Figures 1, 2, 3 and 4, which stipulated the existence of some types in a dependent type theory. Our semantics will be setup in much the same way—we start with a model of the host dependent type theory and then embed a linear model into it. As a model of the host theory we use categories with families [23], but it should be straightforward to adapt our account to other notions of models. The objects of the category modeling dependent type theory stand for the contexts of our type theory, to embed our linear logic, we stipulate a functor $\mathcal{S}p$ which assigns to each context a symmetric monoidal closed category (which moreover comes with a linear exponential comonad to interpret !). By requiring a right-adjoint to the action of context extension under $\mathcal{S}p$, we get a straightforward semantic account of our supply abstraction principle. To model our inclusion of terms into supplies, we require a natural transformation ι between the term presheaf of the dependent type theory and $\mathcal{S}p$. How ι acts on the constructors of inductive types captures the remaining productions.

We will introduce the notions of models of dependent type theory and linear logic relevant to us in Section 5.1. We cannot give a full exposition of the categorical semantics of either dependent type theory or linear logic and refer for more details to standard references [3, 4, 23, 27, 36]. We then spell out the semantics for our theory in Section 5.2.

5.1 Background on semantics of dependent type theory and linear logic

As a model for dependent type theory we will use categories with families [23], where for some category $\mathcal{C}$ we will consider presheaves $\mathcal{F} : \mathbf{Psh}(\mathcal{C})$, i.e., contravariant functors into $\mathbf{Set}$. We make use of the category of elements $\int_{\mathcal{C}} \mathcal{F}$, which has objects pairs of $X : \mathcal{C}$ and $x \in \mathcal{F}(X)$; and morphisms $f : X \to Y$ such that $\mathcal{F}(f)(y) = x$.

Also taking into account the inductive types we used in our paper, a model of dependent type theory is the following.

Definition 5.1. A *model of dependent type theory* is given by

- a category $\mathcal{C}x$ (of *contexts* and *substitutions*) with terminal object $\mathbf{1}$;
- presheaves $\mathcal{T}y : \mathbf{Psh}(\mathcal{C}x)$ and $\mathcal{T}m : \mathbf{Psh}(\int_{\mathcal{C}x} \mathcal{T}y)$, where for some $\gamma : \Gamma_1 \to \Gamma$ we write $\_[\gamma]$ for the actions $\mathcal{T}y(\gamma) : \mathcal{T}y(\Gamma) \to \mathcal{T}y(\Gamma_1)$ and $\mathcal{T}m(\gamma, A) : \mathcal{T}m(\Gamma, A) \to \mathcal{T}m(\Gamma_1, A[\gamma])$,
- a context extension operation, i.e., for any $\Gamma : \mathcal{C}x$ and $A \in \mathcal{T}y(\Gamma)$ we have a context $\Gamma.A$, substitution $\mathbf{p}_A : \Gamma.A \to \Gamma$ and term $\mathbf{q}_A \in \mathcal{T}m(\Gamma.A, A[\mathbf{p}_A])$ with the following universal property: for each $\gamma : \Gamma_1 \to \Gamma$ and $a \in \mathcal{T}m(\Gamma, A[\gamma])$ there is a substitution $\gamma.a : \Gamma_1 \to \Gamma.A$ such that $\mathbf{p} \circ (\gamma.a) = \gamma$, $\mathbf{q}[\gamma.a] = a$ and $\gamma = (\mathbf{p} \circ \gamma).\mathbf{q}[\gamma]$.

Our model supports

- dependent functions if for $\Gamma : \mathcal{C}x$ we have an operation $\Pi : (\amalg_{A:\mathcal{T}y(\Gamma)}\mathcal{T}y(\Gamma.A)) \to \mathcal{T}y(\Gamma)$ and $\mathcal{T}m(\Gamma, \Pi\ A\ B) \cong \mathcal{T}m(\Gamma.A, B)$ all natural in Γ,
- dependent pairs if for any $\Gamma : \mathcal{C}x$ we have an operation $\Sigma : (\amalg_{A:\mathcal{T}y(\Gamma)}\mathcal{T}y(\Gamma.A)) \to \mathcal{T}y(\Gamma)$ and $\mathcal{T}m(\Gamma, \Sigma\ A\ B) \cong \amalg_{a:\mathcal{T}m(\Gamma,A)}\mathcal{T}m(\Gamma, B[\mathrm{id}.a])$ all natural in Γ,
- the unit type if for any $\Gamma : \mathcal{C}x$ we have a type $\top \in \mathcal{T}y(\Gamma)$ such that $\mathcal{T}m(\Gamma, \top) \cong \{\star\}$,
- sums if for any $\Gamma : \mathcal{C}x$ we have operations $+ : \mathcal{T}y(\Gamma) \to \mathcal{T}y(\Gamma) \to \mathcal{T}y(\Gamma)$, $\mathsf{inl} : \mathcal{T}m(\Gamma, \Gamma.A) \to \mathcal{T}m(\Gamma, A + B)$ and $\mathsf{inr} : \mathcal{T}m(\Gamma, B) \to \mathcal{T}m(\Gamma, A + B)$ such that for all $\Gamma, A, B \in \mathcal{T}y(\Gamma)$ and $C \in \mathcal{T}y(\Gamma.A + B)$ we have
  $\mathcal{T}m(\Gamma.A + B, C) \cong \mathcal{T}m(\Gamma.A, C[\mathbf{p}.(\mathsf{inl}\ \mathbf{q})]) \times \mathcal{T}m(\Gamma.B, C[\mathbf{p}.(\mathsf{inr}\ \mathbf{q})])$ all natural in Γ,
- initial algebras for fixpoints if we have operations $\mu : (\mathcal{T}y(\Gamma) \to \mathcal{T}y(\Gamma)) \to \mathcal{T}y(\Gamma)$ and $\mathsf{init} : \{F : \mathcal{T}y(\Gamma) \to \mathcal{T}y(\Gamma)\} \to \mathcal{T}m(\Gamma, F(\mu F)) \to \mathcal{T}m(\Gamma, \mu F)$ such that for any $A \in \mathcal{T}y(\Gamma.\mu F)$, $\mathcal{T}m(\Gamma.\mu F, A) \cong \Pi_{x\in\mathcal{T}m(\Gamma,\mu F)}\mathcal{T}m(\Gamma, A[\mathrm{id}.x])$ all natural in Γ.

△

Since Π, Σ and $\top$ internalise structure already present in type theory, we did not have to introduce their constructors explicitly, below we will use the syntactic terms formers to refer to their inhabitants, i.e., tt stands for the unique inhabitant of $\top$, and $(\_\ ,\ \_)$ for inhabitants of the pair type.

To justify our productions for inductive types, we need to study $\mathcal{T}m$ in more detail. Let us characterise at once all terms at some context, for this we define $\widetilde{\mathcal{T}m} : \mathbf{Psh}(\mathcal{C}x)$ as follows.

$$\widetilde{\mathcal{T}m}(\Gamma) = \amalg_{A:\mathcal{T}y(\Gamma)}\mathcal{T}m(\Gamma, A)$$

$\widetilde{\mathcal{T}m}$ is defined componentwise on morphisms. In effect, $\widetilde{\mathcal{T}m}$ gives us the pointed types at some context, and in other accounts of the semantics of type theory such as the natural models of Awodey [6], $\widetilde{\mathcal{T}m}$ instead of $\mathcal{T}m$ is given as the primitive.

Crucially, pointed types at some context give rise to a categorical structure, let us hence regard $\widetilde{\mathcal{T}m}$ as an indexed category, i.e., not as a presheaf but as a functor assigning a category to each context. Morphisms between two objects $(A, a), (B, b) : \widetilde{\mathcal{T}m}(\Gamma)$ are the basepoint preserving maps, and we moreoever have a monoidal structure as follows.

- The unit is $(\top, \mathsf{tt}) : \widetilde{\mathcal{T}m}(\Gamma)$.
- The product $\widetilde{\mathcal{T}m}(\Gamma) \otimes \widetilde{\mathcal{T}m}(\Gamma) \to \widetilde{\mathcal{T}m}(\Gamma)$ is given by the dependent product, i.e., for $A : \mathcal{T}y(\Gamma)$,$B : \mathcal{T}y(\Gamma.A)$, $a \in \mathcal{T}m(\Gamma, A)$ and $b \in \mathcal{T}m(\Gamma, B[\mathrm{id}.a])$ we have $(A, a) \otimes (B[\mathrm{id}.a], b) \mapsto (\Sigma\ A\ B, (a\ ,\ b))$.

One can readily check that the isomorphisms of a monoidal category are satisfied, e.g., we have $(\top, \mathsf{tt}) \times (A, a) \cong (A, a)$ for any $a : A$.

$\widetilde{\mathcal{T}m}(\Gamma)$ has other morphisms of interest to us. For $A, B \in \mathcal{T}y(\Gamma)$, $a \in \mathcal{T}m(\Gamma, A)$ and $b \in \mathcal{T}m(\Gamma, B)$ consider

$$\mathsf{inl}_{A,B,a} : (A, a) \to (A + B, \mathsf{inl}\ a) \quad \text{and} \quad \mathsf{inr}_{A,B,b} : (B, b) \to (A + B, \mathsf{inr}\ b)$$

and for $F : \mathcal{T}y(\Gamma) \to \mathcal{T}y(\Gamma)$ and $x \in \mathcal{T}m(\Gamma, F(\mu F))$

$$\mathsf{init}_{F,x} : (F(\mu F), x) \to (\mu F, \mathsf{init}\ x)$$

These morphisms are crucially not isomorphisms in $\widetilde{\mathcal{T}m}(\Gamma)$—pointed types are only equivalent if their base types are equivalent—but will have to become isomorphisms in the linear world as we will see shortly.

As a second ingredient to our semantics, we will need to have at hand a notion of semantics for intuitionistic linear logic, for which we use a standard account (albeit without products) [27].

Definition 5.2. A *model of linear logic* is given by a symmetric monoidal closed category C with a linear exponential comonad. Write **LinCat** for the category of models of linear logic. △

5.2 Interpreting our syntax

With our notion of semantics for both dependent type theory and linear logic at hand we can characterise models of our theory. The main idea of our construction was to *embed* linear logic in type theory, let us capture in general what it means to embed some other structure into a model of dependent type theory .

Definition 5.3. A model of dependent type theory *embeds* a functor $\mathcal{L} : \mathcal{C}x^{\mathrm{op}} \to \mathbf{Cat}$ if

- for any $\Gamma : \mathcal{C}x$ we have a type $\mathsf{L} \in \mathcal{T}y(\Gamma)$ such that $\mathcal{T}m(\Gamma, \mathsf{L}) \cong \mathrm{ob}(\mathcal{L}(\Gamma))$ and for any $A \in \mathcal{T}y(\Gamma.\mathsf{L})$, $\mathcal{T}m(\Gamma.\mathsf{L}, A) \cong \Pi_{x \in \mathcal{T}m(\Gamma,\mathsf{L})}\mathcal{T}m(\Gamma, A[\mathrm{id}.x])$ all natural in Γ, and
- for any $\Gamma : \mathcal{C}x$ we have an operation $\mathsf{hom} : \mathcal{T}m(\Gamma, \mathsf{L}) \to \mathcal{T}m(\Gamma, \mathsf{L}) \to \mathcal{T}y(\Gamma)$ such that

$$\Pi_{x,y \in \mathcal{T}m(\Gamma,\mathsf{L})} \mathsf{hom}\ x\ y \cong \Pi_{x,y : \mathcal{L}(\Gamma)} \mathsf{hom}_{\mathcal{L}(\Gamma)}(x, y)$$

  and for any $x, y \in \mathcal{T}m(\Gamma, \mathsf{L})$, $A \in \mathcal{T}y(\Gamma.\mathsf{hom}\ x\ y)$,

$$\mathcal{T}m(\Gamma.\mathsf{hom}\ x\ y, A) \cong \Pi_{f \in \mathcal{T}m(\Gamma, \mathsf{hom}\ x\ y)} \mathcal{T}m(\Gamma, A[\mathrm{id}.f])$$

  all natural in Γ.

△

In particular, our notion of embedding works for any indexed category such as functors into **LinCat**. Equipped with this, we have everything at hand to define models for our syntax.

Definition 5.4. A model of dependent type theory is *linear* if we have an embedded functor $\mathcal{S}p : \mathcal{C}x^{\mathrm{op}} \to \mathbf{LinCat}$ and a natural transformation $\iota : \widetilde{\mathcal{T}m} \to \mathcal{S}p$ such that for any $\Gamma : \mathcal{C}x$,

- ι_Γ is a strong monoidal functor,
- $\iota_\Gamma(\mathsf{inl}_{A,B,a})$ and $\iota_\Gamma(\mathsf{inr}_{A,B,b})$ are isomorphisms for $A, B \in \mathcal{T}y(\Gamma)$, $a \in \mathcal{T}m(\Gamma, A)$, $b \in \mathcal{T}m(\Gamma, B)$, and $\iota_\Gamma(\mathsf{init}_{F,x}) : (F(\mu F), x) \to (\mu F, \mathsf{init}\ x)$ is an isomorphism for any $F : \mathcal{T}y(\Gamma) \to \mathcal{T}y(\Gamma)$ and $x \in \mathcal{T}m(\Gamma, F(\mu F))$, and
- there is a right adjoint $\Lambda_A : \mathcal{S}p(\Gamma.A) \to \mathcal{S}p(\Gamma)$ to $\mathcal{S}p(\mathbf{p}_A) : \mathcal{S}p(\Gamma) \to \mathcal{S}p(\Gamma.A)$ for each $A \in \mathcal{T}y(\Gamma)$, natural in Γ.

△

The embedding takes care of most of the rules for supplies and productions, and in particular also justifies the equalities introduced in Remark 2.3. The substitutions rules for ι described in Remark 2.1 follow from naturality, i.e., we have for any substitution $\gamma : \mathsf{hom}_{\mathcal{C}x}(\Gamma_1, \Gamma_0)$ the following commuting square.

$$\begin{array}{ccc} \widetilde{\mathcal{T}m}(\Gamma_0) & \xrightarrow{\iota_{\Gamma_0}} & \mathcal{S}p(\Gamma_0) \\ {\scriptstyle \widetilde{\mathcal{T}m}(\gamma)}\downarrow & & \downarrow{\scriptstyle \mathcal{S}p(\gamma)} \\ \widetilde{\mathcal{T}m}(\Gamma_1) & \xrightarrow[\iota_{\Gamma_1}]{} & \mathcal{S}p(\Gamma_1) \end{array}$$

The fact that ι_Γ is strong monoidal models the productions for the unit and dependent pair type. In effect, we say that ι_Γ should preserve the monoidal structure that we already happen to have for pointed types. In contrast, the rules requiring that the sum and initial algebra constructors are sent to isomorphism represent a conscious choice—we could have also not required our supplies to be invariant under, e.g., inl and inr, which would have meant that our linear annotations would have captured a different understanding of "resource". While we think our choice is quite natural for

inductive types, there are other sensible options if one wants to represent, e.g., runtime of programs or memory allocation, as we will discuss in Section 7.

Requiring a right adjoint Λ_A to the action of context extension on supplies give us a variable abstraction principle for supplies. In particular, we have the following natural isomorphism for any $\Delta_0 : \mathcal{S}p(\Gamma)$ and $\Delta_1 : \mathcal{S}p(\Gamma.A)$.

$$\mathsf{hom}_{\mathcal{S}p(\Gamma.A)}(\mathcal{S}p(\mathbf{p}_A)(\Delta_0), \Delta_1) \cong \mathsf{hom}_{\mathcal{S}p(\Gamma)}(\Delta_0, \Lambda_A(\Delta_1))$$

The maps between both hom-sets are precisely the bind and free productions, and the fact that we have an isomorphism captures that both productions are mutually inverse. Similar to ι, the substitution rules follow from the appropriate naturality squares.

Note that our model works independently of whether the host type theory has dependent functions, as our syntax had no special rules for functions. However, dependent functions internalise context extension, which we do care about in our model, it is hence very natural to assume that our host theory has dependent functions (and necessary to construct our linear function types in Section 2.3).

Let us study some exemplary models of linear dependent type theory.

Example 5.5. Given a model of dependent type theory, we have a *trivial* linear model by setting each $\mathcal{S}p(\Gamma)$ to the trivial category. The inclusion ι sends everything to the unique object of that category, and the axioms all hold trivially. △

Since we have models of dependent type theory, a corollary of Example 5.5 is consistency of our theory—which is hardly surprising as we did not change any structural rules of our dependent type theory, but just stipulated the existence of additional types. More interesting, and indicative of the fact that we have at hand really a *linear* model, is the following.

Example 5.6. The *sets-and-relations* model of linear dependent type theory is given by an extension to the set model of type theory (explained in more detail, e.g., in [4, Section 3.5]), which is defined as follows.

- $\mathcal{C}x$ is the largest Grothendieck universe $\mathcal{U}$,
- each $\Gamma : \mathcal{C}x$ is an element of $\mathcal{U}$, i.e., a set, a type $A \in \mathcal{T}y(\Gamma)$ is interpreted as a function $A : \Gamma \to \mathcal{U}$ and context extension $\Gamma.A$ as $\amalg_{x\in\Gamma}A(x)$,
- the terms $\mathcal{T}m(\Gamma, A)$ are interpreted as indexed products $\Pi_{x\in\Gamma}A(x)$,
- dependent pair, sum and unit are defined as cartesian product, disjoint sum and a one-element set.

Since our functor $\mathcal{S}p$ is embedded as a type into our theory, we also have that $\mathcal{S}p(\Gamma)$ gives rise to some type Supply, which is interpreted as a function $\Gamma \to \mathcal{U}$, and each $\Delta : \mathcal{S}p(\Gamma)$ gives rise to some $\Delta \in \mathcal{T}m(\Gamma, \mathsf{Supply})$ which is interpreted as a product $\Pi_{x\in\Gamma}\mathsf{Supply}(x)$. Similarly, one can characterise the type of productions.

We can regard each set $\mathsf{Supply}(x)$ as a linear category in the standard way, with cartesian product giving us both the monoidal structure and the internal productions, and finite multisets modeling the linear exponential comonad. If we define ι_Γ to send $(\top, \mathsf{tt})$ to $\{()\}$, $(\Sigma\ A\ B, (x\ ,\ y))$ to $\{(x, y)\}$, but discard all constructors inl, inr and init constructors, it is strong monoidal up to equality, and the class of maps that have to be turned into isomorphisms by ι_Γ are sent to maps between equal sets. Lastly, $\Lambda_A : \mathcal{S}p(\Gamma.A) \to \mathcal{S}p(\Gamma)$, where $\mathcal{S}p(\Gamma.A) = \amalg_{x\in\Gamma}A(x) \to \mathcal{U}$, is defined as sending $\phi : \Pi_{f\in\amalg_{x\in\Gamma}A(x)}\mathsf{Supply}(f)$ to $(x \in \Gamma) \mapsto \amalg_{a\in A(x)}\phi(x, a)$. △

6 Related work

Our system is most directly influenced by the interpretation of the Dialectica translation for type theory put forward by Pédrot [42, 43], which has already led to an implementation of a linear

type system in Cubical Agda [21] (which, in contrast to our system, does not have function types). Gödel's Dialectica translation [50] provides a very general recipe to turn an intuitionistic calculus into a linear one [19, 20, 38]. For function types, the Dialectica translation equips the codomain of a function with copies of the input, which is precisely what our linear derivability predicate ⊩ does. Our system can hence be seen as turning the *target* of the Dialectica translation into a fully fledged type theory. We make crucial additions to what the translation would give us by equipping the codomain not only with a symmetric monoidal structure, but a general linear logic with internalised hom and a linear exponential comonad. Moreover, we give an account of how positive types pan out in the Dialectica translation, which was mentioned as an open problem by Pédrot [42], and devise the supply abstraction principle Λ to incorporate linear dependent function types.

Dependent linear type theories. Our approach differs quite a bit from the main strands of work on dependent linear type theories as we embed linear logic into type theory, as opposed to combining two logics as equals. Many approaches have taken insight from linear-non-linear logic [7, 9] which is based on adjunction between models of intuitionistic and linear logic. Consequently, the systems of Cervesato and Pfenning [14] and Vákár [48] distinguish between intuitionistic and linear variables, and only allow dependencies onto the former kind. While this has inherent limitations, the resulting logic can still model interesting relationships and has been fruitfully applied to, e.g., give a proof-theoretic account of imperative programming [28]. Non-dependent linear-nonlinear logic can also be fruitfully applied to write intrinsically verified parsers [45], the last work provides interesting parallels to our works since they embed linear-nonlinear logic into Agda. At a more general level, Licata et al. [30] provide a framework to combine a large class of logics.

While there are some similarities between the semantics of our systems and existing dependent linear systems—e.g., Vákár [48] also has a strong monoidal functor from contexts to symmetric monoidal categories—our approach is quite different since we leave the structural rules of dependent type theory unchanged, which significantly simplifies our system in comparison with systems that have to manage several sets of structural rules at once. In particular, we only have to moderate how the symmetric monoidal structure formed by pointed types interacts with the symmetric monoidal structure of the linear logic we are embedding, but can continue to use structural rules of the host theory such as weakening since our linear "contexts" are comprised of values and do not introduce new variables.

Index terms in dℓPCF. An idea similar to non-static multiplicities has appeared in dℓPCF, which is a type system for PCF devised by Dal Lago and Gaboardi [16]. Also similarly to ours, their theory works by embedding a model of linear logic—this time history-free game semantics [2]—in an intuitionistic calculus. The typing judgment of dℓPCF comes equipped with an *index term* which specifies the cost of a program. This idea can be understood akin to the situation we were in before introducing function types where we defined foldMaybe−ish in Section 2.2 to produce a linear judgment which came with copies of the input variables. Dal Lago and Gaboardi are primarily interested in measuring the runtime of functional programs, which means that cost is defined differently than in our setting, e.g., they consider the cost of doubling a natural number n presented in unary as n since there were n recursive calls, whereas our system treats any function between the natural numbers as not manipulating any resources. These differences are not crucial however, and we could also make our notion of cost more intensional (for instance by omitting productions for inl and inr). An important difference between the calculus of Dal Lago and Gaboardi [16] and ours is that only the latter has function types which allow for internalising the cost-annotated judgment, which is crucial to deriving proper linear higher-order functions such as foldMaybe in Section 2.3. In particular, we can annotate variables directly with costs, which allows us to model

complex relationships between input variables, whereas dℓPCF is limited to a global analysis of cost.

Dal Lago and Gaboardi deal with the issue of a type system having to compare open terms of the natural numbers, which is in general undecidable, by parametrising their syntax over a theory of function symbols that one can use to compute costs. In our system, we have left it to the user to pick the function symbols they want to use in the type system. It is then also up to the user to devise a tactic which helps with establishing typing judgments, or alternatively prove these manually.

Dal Lago and Gaboardi close by showing *completeness* of their system (relative to the class of function symbols one can use to compute costs), which means that any terminating program computing a natural number can be annotated with a cost. We conjecture that there is a similar result for our system, i.e., for any program we can come up with a linear judgment that gathers all the resources used in this program (this linear judgment cannot necessarily be internalised as a linear function, however, since there might be mutual dependencies between the input parameters). More work is necessary to establish such a result, in particular, we would have to study the operational semantics of our system in more detail.

Quantitative and graded type theories. Our work has been greatly inspired by recent work on quantitative and graded type theories. We rely on the insight of McBride [34] that variable use in *types* is of no concern to a linear typing discipline. The calculus of McBride has been further worked out as Quantitative type theory by Atkey [5] and been implemented in Idris by [12]. Similarly, Orchard et al. [41] have worked out a graded modal type theory in which resource usage is modeled as a coeffect, which also supports lightweight dependent types and has been implemented as Granule. Linear and graded type theories have also been fruitfully applied in the non-dependent case with Linear Haskell [10]. All these systems introduce a resource-sensitive typing disciple by changing the structural rules of the original (dependent) type theory, equipping them with multiplicities taken from some semiring $\mathcal{R}$ which is a parameter to the type system. $\mathcal{R}$ could in principle be the (co)natural numbers, but the static nature of the multiplicities mean that in practice mostly a semiring containing only 0, 1 and ω is used. The unrestricted multiplicity ω is used to overapproximate how often some variable is used, mimicking the ! comonad, which allows for giving resource annotations also to higher-order functions. Granule moreover offers affine types to further narrow down how often some variable can be used, but crucially, a program such as foldMaybe cannot be precisely annotated in either of the aforementioned system.

Moreover, our account gives a very clear answer to how inductive types should be considered in a linear setting. We did not have to stipulate linear elimination rules, but could *derive* these from the usual dependent elimination principles, in contrast to the other theories which require careful study of which elimination principles to postulate [39, 41]. We only had to stipulate that coproducts and initial algebras are resource-invariant to take into account recursive types, we expect that we can follow a similar pattern to also incorporate, e.g., coinductive types.

Our very rich language to specify multiplicities came with the caveat that having open terms of the natural numbers in the types makes type-checking undecidable. We were still able to resolve most productions automatically, but sometimes had to do manual work such as when defining the compose' function in Section 3.3. In contrast, Linear Haskell can automatically type-check functions with arithmetic in multiplicity. For our system, one could envision a tactic which automatically applies lemmas to show that functions like compose' are using the specified resources correctly, but more work is necessary to understand if there are inherent advantages of a static semiring over our natural number type multiplicities with respect to type-checking.

Our implementation of the type theory is highly experimental, and the Agda compiler does not utilise our resource annotations in any way (in fact, it does not even know about it). In contrast, Linear Haskell's compiler takes into account linear types to produce more efficient code. More research (and a more refined implementation of our system) is necessary to fully utilise the benefits of our linear typing disciple.

7 Conclusions

We have devised a novel dependent linear type theory by embedding linear logic in dependent type theory. Almost magically, this gives rise to a quantitative system as we can use a standard natural numbers type of the host theory to represent multiplicities, and crucially, these multiplicities can depend on each other. This gives us a very nuanced type system which allows us give precise resource annotations to higher-order functions that cannot be expressed in any other system.

At the same time, our construction is surprisingly simple and can be carried out in any dependently typed language. We do not have to change the structural rules of the host theory, but just stipulate the existence of two types capturing the structural rules of linear logic, which can straightforwardly be done in much the same way we did in, e.g., Rocq [8] or Lean [18]. We have given a proof-of-concept implementation of our system in Agda [40] and can program in the system like in any other functional language.

Future work

We hope that our dependent linear type theory will open up new possibilities for expressing complex properties of programs in other applications of linear logic such as concurrency [13] and quantum programming languages [46]. On a more concrete level, we hope to pursue the following lines of work further.

Advancing the implementation. Our implementation of the type system in Agda is highly experimental and can be improved in many ways. The solver for productions is still relatively bare-bones and could be extended to solve more complex productions for programs involving multiplicity parameters. From a practical point of view, we are actually not interested in the productions as proof objects, so it might in fact be more expedient to not construct productions explicitly, but only compare normal forms (trusting a sound implementation of this procedure).

Our system can easily be added to any dependently typed language, in the absence of dependent types this is more tricky. In principle, we can also define supply and production data types in, e.g., Haskell (making use of heterogeneous collections), but we have crucially relied on dependent elimination to linearise existing data types. Adding our system to a non-dependently typed language hence requires changes to the type-checker, or the explicit stipulation of linear elimination rules for inductive types.

Further logical and semantical analysis. There are several subtleties involved with respect to positive and negative definitions of the same inductive data types that we have not yet studied in more detail—for example, the dependent pair type is thought to differ in a linear setting depending on whether primacy is given to the pair constructor or the projection function. We also believe that our system allows for adding, e.g., linear additive conjunction & if we add productions which only allow one to use one of the elements of a pair, but more work is necessary to better understand how different linear connectives take shape in our system.

We have tried to give concrete and simple semantics for our theory, but the use of indexed categories suggests that there is something 2-categorical going on, it would be interesting to work out our semantics using more abstract models of dependent type theory such as natural models [6].

Besides, the exact relation of our system with the Dialectica translation [19, 20, 38, 42, 50] has to be better understood. We did not "translate" any terms, but instead used as inspiration what we think the target of a Dialectica-style translation could be. Our treatment of positive types and linear function types can perhaps inform our understanding of the Dialectica translation. More work is necessary to see if there is a mechanical translation of intuitionistic terms into our system, which would give rise to an automatic process for annotating programs with their resource use.

Linearising other classes of types. We have so far only added inductive types to our system, but believe that our approach of stipulating certain functions between pointed types as resource-invariant can also be used to incorporate coinductive types and dependent W types [25]. In fact, Doré [21] already incorporates infinite data structures in their system by drawing multiplicities from *co*natural numbers.

More work is necessary to understand if and how our system can be used to linearise univalent type theories [11, 47], possibly giving rise to linear homotopy type theory [44].

Measuring different kinds of cost. We have chosen to make all constructors of inductive types resource-invariant, but could just as well have omitted, e.g., the productions for inl and inr. In the resulting system, natural numbers defined as the fixpoint of $X \mapsto \top + X$ would be relevant resources, and functions on the natural numbers would come at a cost. This would allow us to measure the runtime of, e.g., a doubling function in the spirit of dℓPCF [16], treating each evaluation step as some sort of (co)effect in the spirit of Danielsson [17].

In fact, we think there are many sensible choices for what one should consider a "resource". We can also think of constructors as pointers to memory cells, and a clever use of our framework might allow for a systematic management of memory in the spirit of Rust's ownership types. The dynamic nature of our multiplicities might enhance this typing discipline and recognise the safety of a larger class of memory usage patterns.

Another way to tweak to our system is to add a production that allows one to *drop* resources, thereby giving rise to an affine type system; or conversely a global *duplication* production giving rise to a type system which forces one to use *at least* the provided resources. We could also remove symmetry from our productions altogether to embed ordered logic [29]. The resulting systems could offer interesting new applications of our approach.

Acknowledgments

I am indebted to Valeria de Paiva and Pierre-Marie Pédrot for introducing me to the Dialectica construction and its type-theoretic manifestation. I am grateful for helpful discussions with Pedro H. Azevedo de Amorim, Evan Cavallo, Nathan Corbyn, Junyuan Chen, Jeremy Gibbons, Daniel Gratzer, Jack Liell-Cock, Sean Moss and Sam Staton, and for the helpful comments of several anonymous reviewers.

References

[1] Michael Gordon Abbott. 2003. *Categories of containers.* Ph. D. Dissertation. University of Leicester, UK.

[2] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full Abstraction for Pcf. *Information and Computation* 163, 2 (2000), 409–470. doi:10.1006/inco.2000.2930

[3] Samson Abramsky and Nikos Tzevelekos. 2011. *Introduction To Categories and Categorical Logic.* Springer, 3–94. doi:10.1007/978-3-642-12821-9_1

[4] Carlo Angiuli and Daniel Gratzer. 2025. *Principles of Dependent Type Theory.* in preparation. https://www.danielgratzer.com/papers/type-theory-book.pdf

[5] 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. doi:10.1145/3209108.3209189

[6] Steve Awodey. 2018. Natural Models of Homotopy Type Theory. *Mathematical Structures in Computer Science* 28, 2 (2018), 241–286. doi:10.1017/S0960129516000268

[7] Andrew Barber and Gordon Plotkin. 1996. *Dual intuitionistic linear logic.* Technical Report ECS-LFCS-96-347.

[8] 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

[9] P. N. Benton. 1995. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 121–135.

[10] 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. doi:10.1145/3158093

[11] Marc Bezem, Thierry Coquand, and Simon Huber. 2014. A Model of Type Theory in Cubical Sets. In *19th International Conference on Types for Proofs and Programs (TYPES) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 26)*, Ralph Matthes and Aleksy Schubert (Eds.). 107–128. doi:10.4230/LIPIcs.TYPES.2013.107

[12] 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. doi:10.4230/LIPIcs.ECOOP.2021.9

[13] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–236. doi:10.1007/978-3-642-15375-4_16

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

[15] Thierry Coquand. 1992. Pattern matching with dependent types. In *Informal proceedings of Logical Frameworks*, Vol. 92. Citeseer, 66–79.

[16] 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. doi:10.1109/LICS.2011.22

[17] Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 133–144. doi:10.1145/1328438.1328457

[18] 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. doi:10.1007/978-3-319-21401-6_26

[19] Valeria Correa Vaz de Paiva. 1990. *The Dialectica Categories.* Ph. D. Dissertation. University of Cambridge, UK.

[20] Valeria Correa Vaz de Paiva. 1991. *The Dialectica categories.* Technical Report UCAM-CL-TR-213. University of Cambridge, Computer Laboratory. doi:10.48456/tr-213

[21] Maximilian Doré. 2025. Linear Types With Dynamic Multiplicities in Dependent Type Theory (Functional Pearl). *Proc. ACM Program. Lang.* 9, ICFP, Article 262 (2025), 21 pages. doi:10.1145/3747531 Note: forthcoming, preprint available at https://www.cs.ox.ac.uk/people/maximilian.dore/icfp25.pdf.

[22] Peter Dybjer. 1991. *Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics.* Cambridge University Press, 280–306. doi:10.1017/CBO9780511569807.012

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

[24] Peter Dybjer. 1997. Representing Inductively Defined Sets By Wellorderings in Martin-Löf's Type Theory. *Theoretical Computer Science* 176, 1 (1997), 329–335. doi:10.1016/S0304-3975(96)00145-4

[25] Nicola Gambino and Martin Hyland. 2004. Wellfounded Trees and Dependent Polynomial Functors. In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 210–225. doi:10.1007/978-3-540-24849-1_14

[26] Jean-Yves Girard. 1987. Linear Logic. *Theoretical computer science* 50, 1 (1987), 1–101.

[27] Martin Hyland and Andrea Schalk. 2003. Glueing and Orthogonality for Models of Linear Logic. *Theoretical Computer Science* 294, 1 (2003), 183–231. doi:10.1016/S0304-3975(01)00241-9 Category Theory and Computer Science.

[28] 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. doi:10.1145/2676726.2676969

[29] Joachim Lambek. 1958. The Mathematics of Sentence Structure. *The American Mathematical Monthly* 65, 3 (1958), 154–170.

[30] Daniel R. Licata, Michael Shulman, and Mitchell Riley. 2017. A Fibrational Framework for Substructural and Modal Logics. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 84)*, Dale Miller (Ed.). Dagstuhl, Germany, 25:1–25:22. doi:10.4230/LIPIcs.FSCD.2017.25

[31] 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. doi:10.1016/S0049-237X(08)71945-1

[32] Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. *Studies in Logic and the Foundations of Mathematics* 104 (1982), 153 – 175. https://dl.acm.org/doi/10.5555/3721.3731

[33] Per Martin-Löf. 1984. *Intuitionistic type theory.* Bibliopolis.

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

[35] Conor McBride and James McKinna. 2004. The View From the Left. *Journal of Functional Programming* 14, 1 (2004), 69–111. doi:10.1017/S0956796803004829

[36] Paul-André Mellies. 2009. Categorical Semantics of Linear Logic. *Panoramas et syntheses* 27 (2009), 15–215.

[37] Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. In *Programming Languages and Systems*, Nobuko Yoshida (Ed.). Springer International Publishing, Cham, 462–490. doi:10.1007/978-3-030-72019-3_17

[38] Sean K. Moss and Tamara von Glehn. 2018. Dialectica models of type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 739–748. doi:10.1145/3209108.3209207

[39] Georgi Nakov and Fredrik Nordvall Forsberg. 2022. Quantitative Polynomial Functors. In *27th International Conference on Types for Proofs and Programs (TYPES) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 239)*, Henning Basold, Jesper Cockx, and Silvia Ghilezan (Eds.). Dagstuhl, Germany, 10:1–10:22. doi:10.4230/LIPIcs.TYPES.2021.10

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

[41] 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. doi:10.1145/3341714

[42] 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). doi:10.1145/2603088.2603094

[43] 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.

[44] Mitchell Riley. 2022. *A Bunched Homotopy Type Theory for Synthetic Stable Homotopy Theory.* Ph. D. Dissertation. Wesleyan University.

[45] Steven Schaefer, Nathan Varner, Pedro Henrique Azevedo de Amorim, and Max S. New. 2025. Intrinsic Verification of Parsers and Formal Grammar Theory in Dependent Lambek Calculus. *Proc. ACM Program. Lang.* 9, PLDI, Article 178 (June 2025), 24 pages. doi:10.1145/3729281

[46] Sam Staton. 2015. Algebraic Effects, Linearity, and Quantum Programming Languages. 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, 395–406. doi:10.1145/2676726.2676999

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

[48] 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. doi:10.1007/978-3-662-46678-0_7

[49] 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. doi:10.1017/S0956796821000034

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

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009