Linear Types inside Dependent Type Theory

Maximilian Doré

Department of Computer Science, University of Oxford, United Kingdom
maximilian.dore@cs.ox.ac.uk

Abstract

We propose a novel approach to combining linear and dependent type theory. By deeply embedding the rules of linear logic inside dependent type theory, we obtain a linear type system which is inherently also dependent. Moreover, we can dynamically compute resource notations, which allows us to give precise types to programs which need a number of copies of some input depending on some other input. We demonstrate our approach with an implementation in Cubical Agda that allows us to program in a practical way. We then propose a novel type theory which also has dependent linear function types.

Background Ever since the rise of dependent type theory and linear logic, the prospect of having a type theory that has both predicates and allows for restricting variable use has inspired research into *dependent linear type theories* [2, 3, 12, 4]. More recently, quantitative [5, 1] and graded type theories [6, 9] have been proposed as practical programming languages in which users can specify in the type of a program how often the program uses a given input, we call this the *multiplicity* of this input. In many cases, the multiplicity of some input depends on the value of some other input, consider for example the following Haskell program.

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

The program uses the backup element `y` only in case the given list is empty. However, systems which have static multiplicities such as quantitative and graded type theories [1, 6, 9] do not allow for precisely capturing this in the type system.

We propose a new approach of combining dependent type theory with linear logic that allows for equipping inputs with multiplicities that depend on the values of other inputs. The main idea behind our system is to deeply embed linear logic in dependent type theory and have the structural rules of linear logic apply to terms of the host theory. More precisely, given a context Γ of a standard dependent type theory, we require a symmetric monoidal category Supply_Γ with bifunctor $\otimes$, plus a bit more structure to be made precise below. We call an object Δ of Supply_Γ a *supply*, and its morphisms *productions* where we write $\Delta_0 \triangleright \Delta_1$ for the collection of morphisms between Δ_0 and Δ_1. We impose this linear structure in the host dependent type theory, i.e., each Supply_Γ and $(-) \triangleright (-)$ are themselves types, and its objects/morphisms are terms. Lastly, any term that is derivable in Γ can be considered a singleton supply, i.e., for any given $\Gamma \vdash a : A$ we have a $\iota(a) : \mathsf{Supply}_\Gamma$.

Using this structure, we define linear entailment as the following dependent type.

$$\Delta \Vdash A \coloneqq \Sigma(a : A)(\Delta \triangleright \iota(a))$$

In words, to conclude A from a supply Δ, we need to give a term $a : A$ as well as a production that turns the supply into this term. We can regard our system as having two nested entailments, where $\vdash$ is intuitionistic entailment and $\Vdash$ is linear entailment.

$$\Gamma \vdash \Delta \Vdash A$$

Having the linear judgment as a dependent type in the host theory has two crucial advantages: we can use open terms of the host theory to compute supplies, which allows for dynamic multiplicities; and we can derive linear elimination principles using normal dependent elimination, which simplifies the introduction of data types in our system in contrast to quantitative type theories [7].

We can implement Supply_Γ as the finite multiset of pointed types in Cubical Agda [8, 13], which we will sketch in Section 1. To add function types to our system, we need Supply_Γ to have exponentials and a variable binding principle, which means we cannot define this in Cubical Agda anymore and need to devise a new type theory, which we will outline in Section 2. An experimental implementation of our system is online: `https://github.com/maxdore/dltt`.

The observation that equipping the output of a function with a bag of resources gives rise to dynamic multiplicities is due to Pierre-Marie Pédrot [10, 11]. We show how to implement this idea in Cubical Agda; and build on it to devise a novel dependent linear type theory.

1. Linear Types in Cubical Agda Cubical Agda's higher inductive types allow for defining finite multisets over some type. We define *supplies* as finite multisets of pointed types, which allows us to put any term in a supply.

```
Supply : Type
Supply = FMSet (Σ[ A ∈ Type ] A)
```

We can readily define functions for constructing the supply containing a single term a, written $\iota\ a$, and for joining two supplies Δ_0 and Δ_1, written $\Delta_0 \otimes \Delta_1$. We can compute the supply containing n copies of some supply for a given natural number n with a straightforward recursive definition.

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

Supply can be regarded as a symmetric monoidal category whose laws hold up to propositional equality. However, we will need to add more morphisms between supplies to take into account constructors of data types, which is why we introduce a dedicated type of morphisms, called *productions*. This type will be extended with other constructors, we only give its main constructors here.

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

Every supply can be turned into itself with id (which allows us to lift equalities between supplies to productions), while $\circ$ and $\otimes^f$ give transitivity and congruence principles for productions. We have omitted equality rules such as id being the unit for composition, these follow in a standard way for symmetric monoidal categories.

Using this structure, we can define our linear judgment as a dependent type as follows.

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

We can conveniently program using this notion of linear judgment. For example, we can implement an analogue of `safeHead` from above in Cubical Agda and give it the following type.

$$\mathsf{safeHead} : (xs : \mathsf{List}\ A) \to (y : A) \to \iota\ xs \otimes (\iota\ y)\ \hat{}\ \mathsf{null}\ xs \Vdash A \times \mathsf{List}\ A$$

We need a single instance of xs in our program, while the multiplicity of y depends on whether xs is null, which is a program that returns 1 if the given list is empty and 0 otherwise. To implement safeHead, we need to add more productions to $\_\triangleright\_$, e.g., a rule to remove a cons constructor from a supply $\iota\ (x :: xs) \triangleright \iota\ x \otimes \iota\ xs$. This rule captures that the free variables of a non-empty list are the same as the free variables of head and tail considered separately.

2. Linear dependent functions In order to add function types to our system, we need additional structure which is not present in Cubical Agda. First, we require that our supplies have exponentials, i.e., each Supply_Γ is a symmetric monoidal closed category where we write $[\Delta_0, \Delta_1]$ for the supply which internalises productions between Δ_0 and Δ_1. Second, we need to be able to bind free variables in supplies, i.e., we require a functor

$$\Lambda_{x:A} : \mathsf{Supply}_{\Gamma,x:A} \to \mathsf{Supply}_\Gamma$$

Furthermore, $\Lambda_{x:A}$ has to be right adjoint to context extension of supplies (context extension of dependent type theory entails that any term of Supply_Γ is also a term of $\mathsf{Supply}_{\Gamma,x:A}$). Using this structure we can define dependent linear function types as follows.

$$\begin{aligned}&(-) \multimap (-) : (A : \mathsf{Type}) \to (B : A \to \mathsf{Type}) \to \Sigma(C : \mathsf{Type})(C \to \mathsf{Supply})\\&(x : A) \multimap B(x) = ((x : A) \to B(x))\ ,\ (\lambda f \to \Lambda_{x:A}[\iota(x), \iota(f\ x)])\end{aligned} \tag{1}$$

In words, a dependent linear function is a dependent function f and a production that witnesses that any input $x : A$ represents the same resources as the output of applying f to x. To iterate this function type, we need to slightly generalise the above definition, we refer the interested reader to the formalisation.

We can derive natural introduction and elimination principles for our functions.

$$\frac{\Gamma, x : A \vdash \Delta \otimes \iota(x) \Vdash b : B(x)}{\Gamma \vdash \Delta \Vdash \lambda x.b : (x : A) \multimap B(x)}(x \notin \mathsf{fv}(\Delta)) \qquad \frac{\Gamma \vdash \Delta_0 \Vdash f : (x : A) \multimap B(x) \qquad \Gamma \vdash \Delta_1 \Vdash a : A}{\Gamma \vdash \Delta_0 \otimes \Delta_1 \Vdash f\ a : B(a)}$$

These rules can be generalised to take in n copies of the input for some open term n of the natural numbers, we write $(x : A)^n \multimap B$ for such a function. Using these functions with multiplicities, we can write safeHead from above as a proper linear dependent function.

$$\mathsf{safeHead} : (xs : \mathsf{List}\ A)^1 \multimap (y : A)^{\mathsf{null}\ xs} \multimap A \times \mathsf{List}\ A$$

Our system therefore has both dependent types and *dependent multiplicities*, giving an expressive language to type many programs that cannot be precisely typed otherwise.

Acknowledgments I am indebted to Valeria de Paiva and Pierre-Marie Pédrot for introducing me to the Dialectica construction; and to Nathan Corbyn and Daniel Gratzer for clearing up my type-theoretic confusions. I am grateful for helpful discussions with Thorsten Altenkirch, Pedro H. Azevedo de Amorim, Evan Cavallo, Jeremy Gibbons, Sean Moss and Sam Staton; and for the comments of several anonymous reviewers.

References

[1] Robert Atkey. Syntax and semantics of quantitative type theory. *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, (LICS):56–65, 2018.

[2] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and computation*, 179(1):19–75, 2002.

[3] Neelakantan R Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. *Principles of Programming Languages 2015 (POPL)*, 50(1):17–30, 2015.

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

[5] Conor McBride. I got plenty o'nuttin'. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 207–233, 2016.

[6] Benjamin Moon, Harley Eades III, and Dominic Orchard. Graded modal dependent type theory. In Nobuko Yoshida, editor, *Programming Languages and Systems*, pages 462–490, Cham, 2021. Springer International Publishing.

[7] Georgi Nakov and Fredrik Nordvall Forsberg. Quantitative polynomial functors. In Henning Basold, Jesper Cockx, and Silvia Ghilezan, editors, *27th International Conference on Types for Proofs and Programs (TYPES)*, volume 239 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:22, Dagstuhl, Germany, 2022.

[8] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

[9] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–30, 2019.

[10] Pierre-Marie Pédrot. 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.

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

[12] Matthijs Vákár. A categorical semantics for linear logical frameworks. In Andrew Pitts, editor, *Foundations of Software Science and Computation Structures*, pages 102–116, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[13] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, 31:e8, 2021.