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

ICFP 2025, National University of Singapore
15 October 2025

Maximilian Doré, University of Oxford
maximilian.dore@cs.ox.ac.uk

Type systems with multiplicities

Linear logic: Don't drop or duplicate variables. $A \otimes B \not\multimap A$ $A \not\multimap A \otimes A$

Useful for programming: all programs of type `List A ⊸ List A` are permutations.

Natural extension: quantitative types.
(Quantitative TT, Granule, Linear Haskell, …)

```
copy : (x : A) →² A × A
```

called *multiplicity* of x

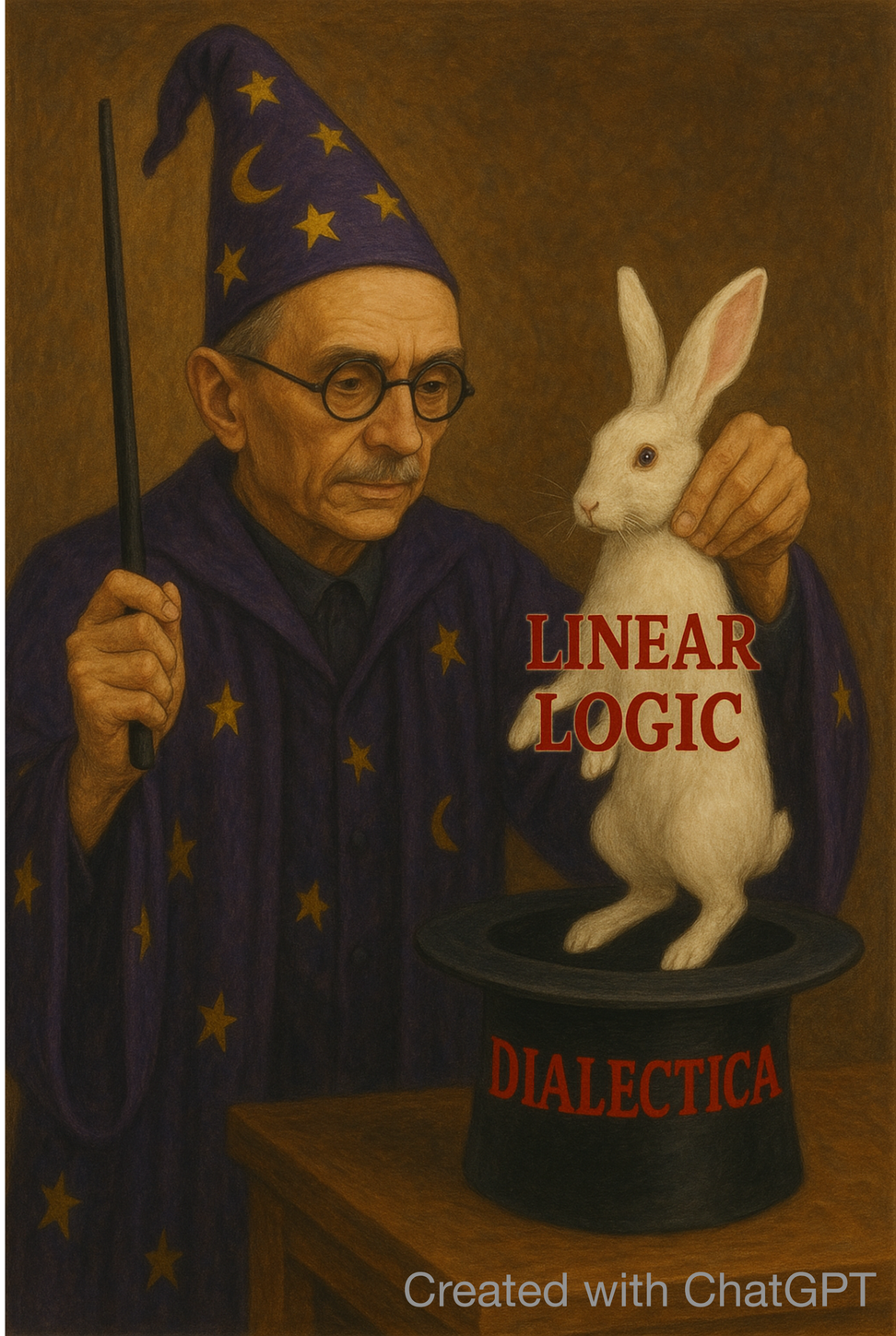
What's the type of

```
safeHead : (xs : List A) →¹ (y : A) →? A × List A
safeHead []       y = (y , [])
safeHead (x :: xs) _ = (x , xs)
```

multiplicity of y depends on whether xs is empty

Add **dynamic** multiplicities to a **dependent type theory**, inspired by *Dialectica*
(carried out in Cubical Agda, but works for any DTT)

The construction



The `Supply` as a bag of values

Cubical Agda supports `Bag`s, ie lists quotiented by reordering:

```
◇   : Bag A
_;_ : A → Bag A → Bag A
_⊗_ : Bag A → Bag A → Bag A
```

Let's capture collections of resources:

```
Supply = Bag (Σ[ A ∈ Type ] A)
```

We can put any value into a supply:

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

Eg, given `x : A` and `xs : List A`:

```
ι x ⊗ ι xs : Supply
```

Intuition: supplies are linear „contexts“ which say which resources we can use

Rearranging supplies with productions ⊸

We can show that `ι x ⊗ ι xs ≡ ι xs ⊗ ι x`. But what about `ι (x :: xs)`?

Productions `_⊸_ : Supply → Supply → Type` capture when supplies are the „same“

Most of the productions `_⊸_` are straightforward (it’s a *hom-set* for supplies);
we introduce additional productions to freely add and remove constructor symbols:

```
_⊸_ : Supply → Supply → Type
  …
  opl[] :         ι [] ⊸ ◇              : lax[]
  opl::  : ι (x :: xs) ⊸ (ι x ⊗ ι xs) : lax::      for x : A , xs : List A
```

(`ι` is *strong monoidal*)

The type of linear judgments

Let's introduce a dependent type to capture when something of type A can be constructed using the resources represented by some supply Δ.

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

This is all we need to obtain a linear typing discipline in Agda!

```
safeHead : (xs : List A) → (y : A) → (if null xs then ι y else ◇) ⊗ ι xs ⊩ A × List A
safeHead []       y = (y , []) , lax, {Goal: (ι y ⊗ ι []) ⊸ ι (y , []) }
safeHead (x :: xs) y = (x , xs) , lax, {Goal: (ι y ⊗ ι (x :: xs)) ⊸ ι (x , xs) }
                                  opl::
```

```
_⊸_ : Supply → Supply → Type
…
opl, :  ι (x , y) ⊸ (ι x ⊗ ι y)  : lax,
opl[]:       ι [] ⊸ ◇            : lax[]
opl:: : ι (x :: xs) ⊸ (ι x ⊗ ι xs) : lax::
```

Programming with Dialectica

Deriving linear elimination principles

We'll write „function spaces“ like so:

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

We can pretend that these are functions:

```
_@_ : (A ⊸ B) → (Δ ⊩ A) → (Δ ⊩ B)
```

```
foldr₁ : (A × B ⊸ B) → (Δ ⊩ B) → (xs : List A) → Δ ⊗ ι xs ⊩ B
foldr₁ f z []       = z by …
foldr₁ f z (x :: xs) = f @ (.  x ,∘ foldr₁ f z xs) by …
```

} we have to construct some productions…

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

… but this is verbose from the artefact!

Linear elimination principles are special cases of dependent elimination!

Dynamic multiplicities

We can use the natural numbers ℕ of Agda to represent multiplicities (in place of semiring built into QTT, Granule or Linear Haskell)

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

```
foldr₂ : ((x : A) → (b : B) → ι b ⊗ ι x ⊗ Δ₁ ⊩ B) → Δ₀ ⊩ B
    → (xs : List A) → Δ₀ ⊗ Δ₁ ^ (length xs) ⊗ ι xs ⊩ B
foldr₂ f z []       = z by {Goal: Δ₀ ⊗ Δ₁ ^ length [] ⊗ ι [] ⊩ B }
foldr₂ f z (x :: xs) = f x @ foldr₂ f z xs by …

intersperse : (x : A) (ys : List A) → ι x ^ (length ys) ⊗ ι ys ⊩ List A
intersperse x = foldr₂ (λ y xys → .  x ::∘ .  y ::∘ .  xys by …) []∘
```

The ∞ multiplicity

We can also draw multiplicities from the conatural numbers ℕ∞ which have ∞ : ℕ∞

```
_^_ : Supply → ℕ∞ → Supply
```

Let's have „linear functions“ taking in a conatural multiplicity:

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

Thereby we can type functions which use a variable infinitely often:

```
iterate : (A ⊸ A) → A -⟨ ∞ ⟩⊸ List A
```

Wrapping up

Summary

We've seen a simple recipe to add a linear typing discipline to an existing dependently typed language, inspired by the *Dialectica* construction.

- Linear elimination principles can be derived using dependent elimination.
- No function types, but it's still practical for writing many functional programs.
- Ability to *compute* multiplicities gives very powerful typing discipline, allowing us to capture resource usage which has to be approximated in systems with static multiplicities (such as QTT, Granule and Linear Haskell).

Based on ideas from Pédrot (Dialectica the Ultimate, TLLA 2024), similar idea to ⊩ used by Atkey (Polynomial Time and Dependent Types, POPL 2024)

What's next

- Dependent linear function types require more structure
- Productions should be constructed automatically

} arXiv:2507.08759
Dependent Multiplicities in Dependent Linear Type Theory

- Adding the construction to other theories: straightforward for Rocq, Lean, … (use of bags not crucial, we can instead add symmetry to productions)
- Utilise dynamic multiplicities for more *efficient* compilation.

Thanks to Pierre-Marie Pédrot, Valeria de Paiva and many others, in particular the ICFP reviewers!