Dependent Multiplicities in Dependent Linear Type Theory

19th South of England Regional Programming Languages Seminar, University of Cambridge

Maximilian Doré

1 July 2026

University of Oxford

maximilian.dore@cs.ox.ac.uk

Linear logic for program verification

- Suppose you want to prove your program `intricateSort : List A → List A` is correct.

  Complicated part: show that your sorting program is a permutation.
- If you work in a linear language, this property comes for free!

  `intricateSort : List A ⊸ List A` is always a permutation (left to show: output sorted)
- Quantitative type theory allows for giving *multiplicities* to inputs:

  `snd : A ^ 0 ⊸ A ^ 1 ⊸ A` has a unique implementation.

Caveat of this approach: few programs use inputs in a way that can be prescribed statically.

This talk: Proper extension of QTT in which multiplicities are *dependent*;
which has natural semantics combining models of DTT and linear logic;
and can linearly type a large class of programs involving branching and recursion.

Embedding linear logic in dependent type theory

Our linear dependent type theory comes with two entailment relations:

- a standard dependent type theory $\Gamma \vdash a : A$ where Γ `Ctxt` and A `Type`.
- a linear calculus $\Delta \Vdash a : A$ where Δ `Sply` (*supply*) and A `LType`. Also have `Sply` `Type`.

$$\frac{}{\Gamma \vdash \diamond \texttt{ Sply}} \qquad \frac{\Gamma \vdash \Delta_0 \texttt{ Sply} \quad \Gamma \vdash \Delta_1 \texttt{ Sply}}{\Gamma \vdash (\Delta_0 \ ; \ \Delta_1) \texttt{ Sply}} \qquad \frac{\Delta_0 \ ; \ \Delta_1 \Vdash a : A}{\Delta_1 \ ; \ \Delta_0 \Vdash a : A}$$

Any A `LType` has an underlying $A^\bullet$ `Type`.

$$\frac{\Gamma \vdash A \texttt{ LType}}{\Gamma \ , \ x : A^\bullet \ \vdash x : A \texttt{ Sply}} \qquad \frac{\Gamma \vdash A \texttt{ LType}}{x : A \ \Vdash x : A} \qquad \frac{\Gamma' \vdash \gamma : \Gamma \quad \Delta \Vdash a : A}{\Delta[\gamma] \Vdash a[\gamma] : A[\gamma]}$$

Example: the supply $x : A \ ; y : B(x) \ ; x : A$ lives in context $x : A^\bullet, y : B^\bullet(x)$.

Linear types with dependent multiplicities.

We can take m copies of Δ $\mathtt{Sply}$ for any $m : \mathbb{N}$ with $\quad \Delta \hat{} m = \mathtt{rec}\mathbb{N}\ \diamond\ (\Delta\ ;(-))\ m$

This allows us to have a function type $(x : A) \hat{} m \to B(x)$ where m is an (open) term:

$$\frac{\Gamma \vdash m : \mathbb{N} \qquad \Delta\ ;(x : A) \hat{} m \Vdash b : B(x)}{\Delta \Vdash \lambda x.b : (x : A) \hat{} m \to B(x)} \qquad \frac{\Delta_0 \Vdash f : (x : A) \hat{} m \to B(x) \qquad \Delta_1 \Vdash a : A}{\Delta_0\ ;\Delta_1\ \hat{}\ m \Vdash f\ a : B(a)}$$

Similarly, introduce a pair type $(x : A) \hat{} m \otimes B(x)$. With $\mathtt{Bool}$ we can branch:

$$\frac{}{\diamond \Vdash \mathtt{true}\ \mathtt{false} : \mathtt{Bool}} \qquad \frac{\diamond \Vdash b : \mathtt{Bool} \qquad \Delta_0 \Vdash c_0 : C\ \mathtt{true} \qquad \Delta_1 \Vdash c_1 : C\ \mathtt{false}}{\Delta_0\ \hat{}\ |\ b\ |\ \ ;\ \Delta_1\ \hat{}\ |\ \neg b\ |\ \ \Vdash\ \mathtt{if}\ b\ \mathtt{then}\ c_0\ \mathtt{else}\ c_1 : C\ b}$$

Example: Defining $A \oplus B$ as $(b : \mathtt{Bool}) \otimes (\mathtt{if}\ b\ \mathtt{then}\ A\ \mathtt{else}\ B)$ and $\mathtt{Maybe}\ A$ as $\top \oplus A$:

$$\mathtt{fromMaybe} : (x : \mathtt{Maybe}\ A) \multimap (y : A) \hat{}\ |\ \mathtt{isnothing}\ x\ | \multimap A$$

Denotational semantics of dependent linear type theory

To interpret our syntax we need a *linear model of DTT*, which is

- *a model of DTT*, ie presheafs $\mathcal{T}m, \mathcal{T}y : \mathcal{C}x^{\mathsf{op}} \to \boldsymbol{Set}$ and a representable $\pi : \mathcal{T}m \to \mathcal{T}y$,
- *indexed in models of linear logic*, ie a functor $\mathcal{S}p : \mathcal{C}x^{\mathsf{op}} \to \boldsymbol{SMCCat}$ which is
  - *embedded*, ie the objects and hom-set of each $\mathcal{S}p(\Gamma)$ form a type in context Γ,
  - *closed under reindexing*, ie have a right adjoint to $\mathcal{S}p(p_A) : \mathcal{S}p(\Gamma) \to \mathcal{S}p(\Gamma.A)$,
  - *populated by ground terms*, ie have $\iota : \mathcal{T}m_0 \to \mathcal{S}p$.

We interpret $\Delta \Vdash a : A$ as $\quad (a, \delta) : (a : \mathcal{T}m(\Gamma, A)) \times hom(\Delta, \Theta(a))$,
where A is interpreted as $\quad (A, \Theta) : (A : \mathcal{T}y) \times \mathcal{S}p(\Gamma.A)$.

Proposition: Any linear model of DTT is a model of QTT (with resource algebra $\mathbb{N}$).

Turning the semantics into a prototypical Agda implementation

Dependent type theory is expressive enough to internalise the denotational semantics.

```
data Sply : Type where
  ι : {A : GType} → A → Sply
  ◇ : Sply
  _;_ [_,_] : Sply → Sply → Sply
  Λ : (A : Type) → (A → Sply) → Sply
```

```
data _⇒_ : Sply → Sply → Type where
  id : Λ Δ → Δ ⇒ Δ
  _∘_ : Δ₁ ⇒ Δ₂ → Δ₀ ⇒ Δ₁ → Δ₀ ⇒ Δ₂
  swap : Δ₀ ; Δ₁ ⇒ Δ₁ ; Δ₀
  ...
```

```
_⊩_ : Sply → LType → Type
Δ ⊩ A = (a : A•) × (Δ ⇒ θ a)
  where (A• , θ) = ⟦ A ⟧
```

```
_^_ : Sply → ℕ → Sply
Δ ^_ = recℕ (λ _ → _; Δ) ◇
```

```
⟦_⟧ : LType → (A : Type) × (A → Sply)
⟦ GType A ⟧ = A , (λ x → ι x)
⟦ Bool ⟧ = Bool• , (λ _ → ◇)
⟦ (x : A)^ m ⊗ (B x) ⟧ = ((x : A•) × B• x) , (λ (x , y) → (θ x) ^ m ; (Ξ y))
⟦ (x : A)^ m → (B x) ⟧ = ((x : A•) → B• x) , (λ f → Λ(x : A•) [(θ x)^m,Ξ(f x)]
```

Programming in the Agda prototype

Internalising the semantics gives rise to a somewhat practical implementation of the system, reminiscient of the Dialectica construction.

Recall: `Δ ⊩ A = (a : A•) × (Δ ⇒ θ a)`

We can then implement:

```
fromMaybe : (x : (Maybe A)•) (y : A•)
          → (x : Maybe A)  ⨟  (y : A) ^ | isnothing x | ⊩ A
fromMaybe (just x) y = x , unitl ∘ unitr
fromMaybe (nothing) y = y ,  unitl ∘ unitl ⨟ᶠ unitr
```

> We *program* in Agda, and *prove* validity of the linear derivation.

Multiplicities depending on inductive types

We can add inductive types to our system without requiring anything else from our models (other than the host DTT supporting inductive types):

```
⟦ List A ⟧ = List• A• , foldr (λ x → (x : A) ;_) ◇
```

With this interpretation it's easy to justify the following recursor:

$$\frac{\Delta_0 \Vdash B \qquad x : A \;;\; y : B \;;\; \Delta_1 \Vdash B}{xs : \texttt{List}\ A \;;\; \Delta_0 \;;\; \Delta_1 \,\hat{}\, (\texttt{length}\ xs) \Vdash B}$$

We can give multiplicities to inputs which are used depending on inductive inputs, eg

```
map : (xs : List A) ⊸ (f : A ⊸ B) ^ (length xs) ⊸ List B
```

Taking fractions of inductive types

Many programs only use parts of a value, with *fractional* multiplicities we can type these:

```
ListMult : List A• → Type
ListMult = foldr (λ _ → ℕ ×_) ⊤
```

Let's change the interpretation of lists to take fractions into account:

```
⟦ List A ^ m ⟧ = List A• , λ xs → ListSply A xs (m xs) where
  ListSply : (xs : List A•) → ListMult xs → Sply
  ListSply [] m = ◇
  ListSply (x ∷ xs) (m , ms) = (x : A) ^ m ; ListSply xs ms
```

We can now type a large class of programs using inductive types, eg

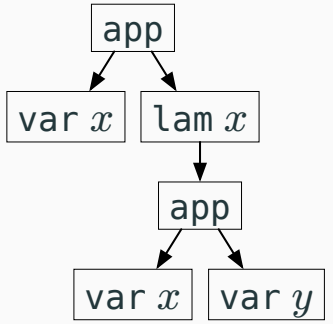
```
head : (xs : List A ^ (λ {_∷_ → (1 , const 0)})) ⊸ Maybe A
```

Case study: substitution in the simply typed lambda calculus

For a given linear type `Var`, consider an encoding of the lambda calculus:

```
data Term : LType where
  var : Var → Term
  app : Term → Term → Term
  lam : Var → Term → Term
```

Eg, the lambda term $x(\lambda x.xy)$ has the AST:



Example: substitute some term into `s` for every free occurrence of variable `x`:

```
exceptFree x (var y) = | y ≠ x |
exceptFree x (app u v) = (1 , exceptFree x)
exceptFree x (lam y) = if y ≡ x then (const 1) else (1 , exceptFree x)

subst : (x : Var) ⊸ (s : Term ^ exceptFree x) ⊸ Term ^ freeOccs x s ⊸ Term
```

Conclusions

New approach to combining linear logic with dependent type theory: *embed* linear structure in an otherwise unchanged DTT. Gives rise to a powerful and ergonomic typing discipline.

Straightforward categorical semantics, technical contribution is interpreting the linear types.

Future work: broaden scope beyond value linearity:

- Use linear logic as a foundation for complexity analysis/theory.
- Apply approach with other substructural logics, eg separation logic.

> What happens if we allow the assumptions Δ in a judgment $\Delta \Vdash a : A$ to be dynamic?

Many thanks to Pierre-Marie Pédrot, Valeria de Paiva and many others!