Linear Types inside Dependent Type Theory

TYPES 2025, University of Strathclyde, Glasgow
13 June 2025

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

Last year in Tallinn

In This Talk

A 100% free idea for a research project!

- I am too lazy to write a paper on that
- It is more efficient to point at the moon
- Invited talks are great for dissemination of such knowledge

Graded / quantitative types are a poor man's Dialectica

- More positively, Dialectica is a finer kind of graded types
- Compatible with rich types (i.e. MLTT)
- Dialectica as proof-relevant, higher-order complexity annotations

Pierre-Marie Pédrot, *Dialectica the Ultimate*, Trends in Linear Logic and Applications 08/07/2024

Embedding linear types in dependent type theory

- We'll take the target of the Dialectica translation as inspiration to carve out a linear type system. In a nutshell, we're deeply embedding linear logic in DTT.
- We can compute in the linear types, which gives rise to *dynamic multiplicities*:
  $\rightarrow$ capture if some variable is used depending on the value of another variable
- We'll implement this in Cubical Agda, which gives a practically useful type system. We'll incorporate *positive* types along the way.
- Finally, we'lll sketch how this idea gives rise to a *linear dependent type theory*.

Equipping codomains with finite multisets

To linearise a function, we equip its output with copies of the inputs.

Equipping codomains with finite multisets

To linearise a function, we equip its output with copies of the inputs.

write :: for cons and ◇ for nil

Let's introduce a type that can gather all inputs:

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

Equipping codomains with finite multisets

To linearise a function, we equip its output with copies of the inputs.

write :: for cons and ◇ for nil

Let's introduce a type that can gather all inputs:

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

Constructing supplies:

```
ι : A → Supply
ι a = (A , a) :: ◇

  ◇ : Supply
_⊗_ : Supply → Supply → Supply
```

Equipping codomains with finite multisets

To linearise a function, we equip its output with copies of the inputs.

write :: for cons and ◇ for nil

Let's introduce a type that can gather all inputs:

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

Constructing supplies:

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

```
 ◇  : Supply
_⊗_ : Supply → Supply → Supply
```

We introduce a linear judgment:

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

Equipping codomains with finite multisets

To linearise a function, we equip its output with copies of the inputs.

write :: for cons and ◇ for nil

Let's introduce a type that can gather all inputs:

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

Constructing supplies:

```
ι : A → Supply
ι a = (A , a) :: ◇

 ◇  : Supply
_⊗_ : Supply → Supply → Supply
```

We introduce a linear judgment:

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

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

Equipping codomains with finite multisets

To linearise a function, we equip its output with copies of the inputs.

write :: for cons and ◇ for nil

Let's introduce a type that can gather all inputs:

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

Constructing supplies:

```
ι : A → Supply
ι a = (A , a) :: ◇

◇ : Supply
_⊗_ : Supply → Supply → Supply
```

We introduce a linear judgment:

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

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

Equipping codomains with finite multisets

To linearise a function, we equip its output with copies of the inputs.

write :: for cons and ◇ for nil

Let's introduce a type that can gather all inputs:

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

Constructing supplies:

```
ι : A → Supply
ι a = (A , a) :: ◇

 ◇  : Supply
_⊗_ : Supply → Supply → Supply
```

We introduce a linear judgment:

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

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

`ι [] “≡“ ◇` and `ι (x :: xs) “≡“ ι (x , xs) “≡“ ι x ⊗ ι xs` etc.

Incorporating constructors

```
data _▷_ : Supply → Supply → Type where
    id : Δ ▷ Δ
    _∘_ : Δ₁ ▷ Δ₂ → Δ₀ ▷ Δ₁ → Δ₀ ▷ Δ₂
    _⊗ᶠ_ : Δ₀ ▷ Δ₁ → Δ₂ ▷ Δ₃ → (Δ₀ ⊗ Δ₂) ▷ (Δ₁ ⊗ Δ₃)
    opl, : ι (a , b)  ⋈ (ι a ⊗ ι b)  : lax,     (for a : A, b : B a)
    opl[]: ι []        ⋈ ◇             : lax[]
    opl:: : ι (x :: xs) ⋈ (ι x ⊗ ι xs) : lax::     (for x : A , xs : List A)
```

Incorporating constructors

```
data _▷_ : Supply → Supply → Type where
    id : Δ ▷ Δ
    _∘_ : Δ₁ ▷ Δ₂ → Δ₀ ▷ Δ₁ → Δ₀ ▷ Δ₂
    _⊗ᶠ_ : Δ₀ ▷ Δ₁ → Δ₂ ▷ Δ₃ → (Δ₀ ⊗ Δ₂) ▷ (Δ₁ ⊗ Δ₃)
    opl, : ι (a , b)  ⋈ (ι a ⊗ ι b)  : lax,     (for a : A, b : B a)
    opl[]: ι []       ⋈ ◇            : lax[]
    opl:: : ι (x :: xs) ⋈ (ι x ⊗ ι xs) : lax::  (for x : A , xs : List A)
```

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

Incorporating constructors

```
data _▷_ : Supply → Supply → Type where
    id : Δ ▷ Δ
    _∘_ : Δ₁ ▷ Δ₂ → Δ₀ ▷ Δ₁ → Δ₀ ▷ Δ₂
    _⊗ᶠ_ : Δ₀ ▷ Δ₁ → Δ₂ ▷ Δ₃ → (Δ₀ ⊗ Δ₂) ▷ (Δ₁ ⊗ Δ₃)
    opl, : ι (a , b)  ⋈ (ι a ⊗ ι b)  : lax,     (for a : A, b : B a)
    opl[]: ι []       ⋈ ◇            : lax[]
    opl:: : ι (x :: xs) ⋈ (ι x ⊗ ι xs) : lax::   (for x : A , xs : List A)
```

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

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

Incorporating constructors

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

```
data _▷_ : Supply → Supply → Type where
    id : Δ ▷ Δ
    _∘_ : Δ₁ ▷ Δ₂ → Δ₀ ▷ Δ₁ → Δ₀ ▷ Δ₂
    _⊗ᶠ_ : Δ₀ ▷ Δ₁ → Δ₂ ▷ Δ₃ → (Δ₀ ⊗ Δ₂) ▷ (Δ₁ ⊗ Δ₃)
    opl, : ι (a , b)  ⋈ (ι a ⊗ ι b)  : lax,      (for a : A, b : B a)
    opl[]: ι []       ⋈ ◇            : lax[]
    opl:: : ι (x :: xs) ⋈ (ι x ⊗ ι xs) : lax::    (for x : A , xs : List A)

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

Incorporating constructors

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

```
data _▷_ : Supply → Supply → Type where
    id : Δ ▷ Δ
    _∘_ : Δ₁ ▷ Δ₂ → Δ₀ ▷ Δ₁ → Δ₀ ▷ Δ₂
    _⊗ᶠ_ : Δ₀ ▷ Δ₁ → Δ₂ ▷ Δ₃ → (Δ₀ ⊗ Δ₂) ▷ (Δ₁ ⊗ Δ₃)
    opl, : ι (a , b)  ▷◁ (ι a ⊗ ι b)  : lax,     (for a : A, b : B a)
    opl[]: ι []       ▷◁ ◇            : lax[]
    opl:: : ι (x :: xs) ▷◁ (ι x ⊗ ι xs) : lax::   (for x : A , xs : List A)

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

Incorporating constructors

```
data _▷_ : Supply → Supply → Type where
    id : Δ ▷ Δ
    _∘_ : Δ₁ ▷ Δ₂ → Δ₀ ▷ Δ₁ → Δ₀ ▷ Δ₂
    _⊗ᶠ_ : Δ₀ ▷ Δ₁ → Δ₂ ▷ Δ₃ → (Δ₀ ⊗ Δ₂) ▷ (Δ₁ ⊗ Δ₃)
    opl, : ι (a , b)  ⋈ (ι a ⊗ ι b)  : lax,     (for a : A, b : B a)
    opl[]: ι []       ⋈ ◇             : lax[]
    opl:: : ι (x :: xs) ⋈ (ι x ⊗ ι xs) : lax::   (for x : A , xs : List A)
```

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

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

Linear elimination principles are derivable using dependent elimination:

```
foldr : ((x : A) → (b : B) → ι b ⊗ ι x ⊗ Δ₁ ⊩ B)
  → Δ₀ ⊩ B → (xs : List A)
  → Δ₀ ⊗ Δ₁ ^ (length xs) ⊗ ι xs ⊩ B
```

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

Incorporating constructors

```
data _▷_ : Supply → Supply → Type where
    id : Δ ▷ Δ
    _∘_ : Δ₁ ▷ Δ₂ → Δ₀ ▷ Δ₁ → Δ₀ ▷ Δ₂
    _⊗ᶠ_ : Δ₀ ▷ Δ₁ → Δ₂ ▷ Δ₃ → (Δ₀ ⊗ Δ₂) ▷ (Δ₁ ⊗ Δ₃)
    opl, : ι (a , b)  ⋈ (ι a ⊗ ι b)  : lax,     (for a : A, b : B a)
    opl[]: ι []       ⋈ ◇            : lax[]
    opl:: : ι (x :: xs) ⋈ (ι x ⊗ ι xs) : lax::   (for x : A , xs : List A)
```

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

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

Linear elimination principles are derivable using dependent elimination:

```
foldr : ((x : A) → (b : B) → ι b ⊗ ι x ⊗ Δ₁ ⊩ B)
  → Δ₀ ⊩ B → (xs : List A)
  → Δ₀ ⊗ Δ₁ ^ (length xs) ⊗ ι xs ⊩ B
```

where

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

We can construct many functional programs in this system.

Sketching the semantics

We have equipped DTT (given by category Cx with presheaves $Term$ etc.) with

- a presheaf valued in symmetric monoidal cats: $Supply : Cx^{op} \to \mathbf{SMCat}$
- A natural transformation embedding each term: $\iota : Term \Rightarrow Supply$

  Moreover, ι is strongly monoidal with respect to products of types, e.g.,
  $\iota\ (x :: xs) \simeq \iota\ x \otimes \iota\ xs$

Sketching the semantics

We have equipped DTT (given by category Cx with presheaves $Term$ etc.) with

- a presheaf valued in symmetric monoidal cats: $Supply : Cx^{op} \to \mathbf{SMCat}$
- A natural transformation embedding each term: $\iota : Term \Rightarrow Supply$

  Moreover, ι is strongly monoidal with respect to products of types, e.g.,
  $\iota\ (x :: xs) \simeq \iota\ x \otimes \iota\ xs$

This structure gives rise to a two-step calculus $\Gamma \vdash \underbrace{\Delta \Vdash A}$

defined as $\Sigma(a : A), \text{hom}_{Supply(\Gamma)}(\iota\ a, \Delta)$

Sketching the semantics

We have equipped DTT (given by category Cx with presheaves $Term$ etc.) with

- a presheaf valued in symmetric monoidal cats: $Supply : Cx^{op} \to \mathbf{SMCat}$
- A natural transformation embedding each term: $\iota : Term \Rightarrow Supply$

  Moreover, ι is strongly monoidal with respect to products of types, e.g., $\iota\ (x :: xs) \simeq \iota\ x \otimes \iota\ xs$

This structure gives rise to a two-step calculus $\Gamma \vdash \underbrace{\Delta \Vdash A}_{\text{defined as } \Sigma(a : A), \hom_{Supply(\Gamma)}(\iota\ a, \Delta)}$

To add function types, let's beef up $Supply(\Gamma)$ with more structure!

Linear dependent function types

To close our calculus under function types, we add two more things:

- exponentials $[\Delta_0, \Delta_1]$ each $Supply(\Gamma)$ is symmetric monoidal *closed*
- $\Lambda_{x:A}\Delta$ binding x in Δ functor $\Lambda_{x:A} : Supply(\Gamma, x : A) \to Supply(\Gamma)$ which is right adjoint to context extension $Supply(\mathbf{p}_{x:A})$

Linear dependent function types

To close our calculus under function types, we add two more things:

- exponentials $[\Delta_0, \Delta_1]$ — each $Supply(\Gamma)$ is symmetric monoidal *closed*
- $\Lambda_{x:A}\Delta$ binding x in Δ — functor $\Lambda_{x:A} : Supply(\Gamma, x : A) \to Supply(\Gamma)$ which is right adjoint to context extension $Supply(\mathbf{p}_{x:A})$

This allows us to define a type of dependent linear functions from A to B:

$$(x : A) \multimap B(x) := \underbrace{(x : A) \to B(x)}_{\text{usual dependent function}}\ ,\ \lambda\, f \to \Lambda_{x:A}\underbrace{[\iota\ x, \iota\ (f\ x)]}_{\text{production which establishes that } f \text{ needs one input to produce an output}}$$

Using linear dependent functions

$(x : A) \multimap B(x) := (x : A) \to B(x) \ , \ \lambda f \to \Lambda_{x:A}[\iota\ x, \iota\ (f\ x)]$

$\mathsf{cur} : (\Delta_0 \otimes \Delta_1 \rhd \Delta_2) \to (\Delta_0 \rhd [\Delta_1 \otimes \Delta_2])$
$\mathsf{bind}_{x:A} : (\Delta_0 \ \rhd \Delta_1(x)) \to (\Delta_0 \ \rhd \Lambda_{x:A}\Delta_1)$

Using linear dependent functions

$$(x : A) \multimap B(x) := (x : A) \to B(x)\ ,\ \lambda\, f \to \Lambda_{x:A}[\iota\ x, \iota\ (f\ x)]$$

$\mathsf{cur} : (\Delta_0 \otimes \Delta_1 \rhd \Delta_2) \to (\Delta_0 \rhd [\Delta_1 \otimes \Delta_2])$
$\mathsf{bind}_{x:A} : (\Delta_0\ \rhd \Delta_1(x)) \to (\Delta_0\ \rhd \Lambda_{x:A}\Delta_1)$

Suppose $x : A \vdash \iota\ x \Vdash b : B(x)$, so we have $(b, \delta) : \Sigma(b : B\ x), \iota\ x \rhd \iota\ b$

Using linear dependent functions

$$(x : A) \multimap B(x) := (x : A) \to B(x)\ ,\ \lambda\, f \to \Lambda_{x:A}[\iota\ x, \iota\ (f\ x)]$$

cur : $(\Delta_0 \otimes \Delta_1 \triangleright \Delta_2) \to (\Delta_0 \triangleright [\Delta_1 \otimes \Delta_2])$
bind$_{x:A}$: $(\Delta_0\ \triangleright \Delta_1(x)) \to (\Delta_0\ \triangleright \Lambda_{x:A}\Delta_1)$

Suppose $x : A \vdash \iota\ x \Vdash b : B(x)$, so we have $(b, \delta) : \Sigma(b : B\ x), \iota\ x \triangleright \iota\ b$

Setting $f := \lambda x\,.\, b$, we get bind$_{x:A}$(cur(δ)) : $\Lambda_{x:A}[\iota\ x, \iota\ (f\ x)]$.

Using linear dependent functions

$$(x : A) \multimap B(x) := (x : A) \to B(x)\ ,\ \lambda f \to \Lambda_{x:A}[\imath\ x, \imath\ (f\ x)]$$

$\mathsf{cur} : (\Delta_0 \otimes \Delta_1 \rhd \Delta_2) \to (\Delta_0 \rhd [\Delta_1 \otimes \Delta_2])$
$\mathsf{bind}_{x:A} : (\Delta_0\ \rhd \Delta_1(x)) \to (\Delta_0\ \rhd \Lambda_{x:A}\Delta_1)$

Suppose $x : A \vdash \imath\ x \Vdash b : B(x)$, so we have $(b, \delta) : \Sigma(b : B\ x), \imath\ x \rhd \imath\ b$

Setting $f := \lambda x\,.\,b$, we get $\mathsf{bind}_{x:A}(\mathsf{cur}(\delta)) : \Lambda_{x:A}[\imath\ x, \imath\ (f\ x)]$.

$$\frac{\Gamma, x : A \vdash \Delta \otimes \imath\ x \Vdash b : B(x)}{\Gamma \vdash \Delta \Vdash \lambda x\,.\,b : (x : A) \multimap B(x)}\ \multimap I\ (x \notin \Delta)$$

Using linear dependent functions

$$(x : A) \multimap B(x) := (x : A) \to B(x)\ ,\ \lambda f \to \Lambda_{x:A}[\iota\ x, \iota\ (f\ x)]$$

$$\mathsf{cur} : (\Delta_0 \otimes \Delta_1 \rhd \Delta_2) \to (\Delta_0 \rhd [\Delta_1 \otimes \Delta_2])$$
$$\mathsf{bind}_{x:A} : (\Delta_0\ \rhd \Delta_1(x)) \to (\Delta_0\ \rhd \Lambda_{x:A}\Delta_1)$$

Suppose $x : A \vdash \iota\ x \Vdash b : B(x)$, so we have $(b, \delta) : \Sigma(b : B\ x), \iota\ x \rhd \iota\ b$

Setting $f := \lambda x\,.\,b$, we get $\mathsf{bind}_{x:A}(\mathsf{cur}(\delta)) : \Lambda_{x:A}[\iota\ x, \iota\ (f\ x)]$.

$$\frac{\Gamma, x : A \vdash \Delta \otimes \iota\ x \Vdash b : B(x)}{\Gamma \vdash \Delta \Vdash \lambda x\,.\,b : (x : A) \multimap B(x)}\ \multimap I\ (x \notin \Delta)$$

Similarly, we can derive elimination rule. Also works for incorporating ^ m, e.g.,

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

Summary

- Dialectica gives rise to a practically useful linear type system in Cubical Agda.
- Essentially, we have deeply embedded linear logic in dependent type theory.
  $\rightarrow$ this allows us to compute in linear types using our host theory.
  Similar to index terms of Dal Lago & Gaboardi's dℓPCF (LICS 2011)
- We can extend this to a fully-fledged linear dependent type theory.
- For unrestricted variable use, equip $Supply(\Gamma)$ with exponential comonad.

Summary

- Dialectica gives rise to a practically useful linear type system in Cubical Agda.
- Essentially, we have deeply embedded linear logic in dependent type theory.
  $\to$ this allows us to compute in linear types using our host theory.
  Similar to index terms of Dal Lago & Gaboardi's dℓPCF (LICS 2011)
- We can extend this to a fully-fledged linear dependent type theory.
- For unrestricted variable use, equip $Supply(\Gamma)$ with exponential comonad.

Thanks to Pierre-Marie Pédrot, Valeria de Paiva, Nathan Corby, Daniel Gratzer, Evan Cavallo and several anonymous reviewers for very helpful discussions!

Summary

- Dialectica gives rise to a practically useful linear type system in Cubical Agda.
- Essentially, we have deeply embedded linear logic in dependent type theory.
  $\rightarrow$ this allows us to compute in linear types using our host theory.
  Similar to index terms of Dal Lago & Gaboardi's dℓPCF (LICS 2011)
- We can extend this to a fully-fledged linear dependent type theory.
- For unrestricted variable use, equip $Supply(\Gamma)$ with exponential comonad.

Thanks to Pierre-Marie Pédrot, Valeria de Paiva, Nathan Corby, Daniel Gratzer, Evan Cavallo and several anonymous reviewers for very helpful discussions!

Code: https://github.com/maxdore/dynltt and https://github.com/maxdore/dltt

Specifying variable use with linear types

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

Specifying variable use with linear types

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

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

Specifying variable use with linear types

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

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

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

```
copy : (x : A) ⊸² A × A
```

called *multiplicity* of x

Specifying variable use with linear types

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, Graded TT, 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)
```

Specifying variable use with linear types

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

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

Natural extension: quantitative types.
(Quantitative TT, Graded TT, 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 depends on whether xs is empty

Specifying variable use with linear types

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

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

Natural extension: quantitative types.
(Quantitative TT, Graded TT, 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 depends on whether xs is empty

Proposal: impose linear rules *inside* dependent type theory.
This allows us to have *dynamic/dependent* multiplicities.

$$\Gamma \vdash \underbrace{\Delta \Vdash A}$$

defined as certain dependent type

(Linear) judgment day

Cubical Agda supports finite multisets, which behave just like lists except that the order of elements does not matter.

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

(Linear) judgment day

Cubical Agda supports finite multisets, which behave just like lists except that the order of elements does not matter.

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

We can append finite multisets:

```
_⊗_ : FMSet A → FMSet A → FMSet A
```

(Linear) judgment day

Cubical Agda supports finite multisets, which behave just like lists except that the order of elements does not matter.

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

We can append finite multisets:

```
_⊗_ : FMSet A → FMSet A → FMSet A
```

We call a bag of terms a *supply*:

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

(Linear) judgment day

Cubical Agda supports finite multisets, which behave just like lists except that the order of elements does not matter.

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

We can append finite multisets:

```
_⊗_ : FMSet A → FMSet A → FMSet A
```

We call a bag of terms a *supply*:

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

We can define a unit supply:

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

(Linear) judgment day

Cubical Agda supports finite multisets, which behave just like lists except that the order of elements does not matter.

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

We can append finite multisets:

```
_⊗_ : FMSet A → FMSet A → FMSet A
```

We call a bag of terms a *supply*:

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

We can define a unit supply:

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

And introduce a *linear judgment*:

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

(Linear) judgment day

Cubical Agda supports finite multisets, which behave just like lists except that the order of elements does not matter.

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

We can append finite multisets:

```
_⊗_ : FMSet A → FMSet A → FMSet A
```

We call a bag of terms a *supply*:

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

We can define a unit supply:

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

And introduce a *linear judgment*:

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

we already have many useful equalities, e.g., `swap : Δ₀ ⊗ Δ₁ ≡ Δ₁ ⊗ Δ₀`

Same, but different. But still same

```
switch : (z : A × B) → ι z ⊩ B × A
switch (x , y) = (y , x) ,{Goal: ι (x , y) “≡“ ι (y , x) }
```

Same, but different. But still same

```
switch : (z : A × B) → ι z ⊩ B × A
switch (x , y) = (y , x) ,{Goal: ι (x , y) “≡“ ι (y , x) }
```

Adding and removing pair constructor doesn't change the free variables of a supply.
→ introduce notion of sameness for supplies, which we call *productions*.

```
data _▷_ : Supply → Supply → Type where
    id : Δ ▷ Δ
    _∘_ : Δ₁ ▷ Δ₂ → Δ₀ ▷ Δ₁ → Δ₀ ▷ Δ₂
    _⊗ᶠ_ : Δ₀ ▷ Δ₁ → Δ₂ ▷ Δ₃ → (Δ₀ ⊗ Δ₂) ▷ (Δ₁ ⊗ Δ₃)

    opl, : ι (a , b) ▷ (ι a ⊗ ι b) : lax,        (for a : A, b : B a)
```

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

Same, but different. But still same

```
switch : (z : A × B) → ι z ⊩ B × A
switch (x , y) = (y , x) , lax, ∘ swap (ι x) (ι y) ∘ opl,
```

Adding and removing pair constructor doesn’t change the free variables of a supply.
→ introduce notion of sameness for supplies, which we call *productions*.

```
data _▷_ : Supply → Supply → Type where
    id : Δ ▷ Δ
    _∘_ : Δ₁ ▷ Δ₂ → Δ₀ ▷ Δ₁ → Δ₀ ▷ Δ₂
    _⊗ᶠ_ : Δ₀ ▷ Δ₁ → Δ₂ ▷ Δ₃ → (Δ₀ ⊗ Δ₂) ▷ (Δ₁ ⊗ Δ₃)

    opl, : ι (a , b) ▷ (ι a ⊗ ι b) : lax,        (for a : A, b : B a)
```

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

A natural resource algebra

We get multiplicities for free using the standard natural numbers type:

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

A natural resource algebra

We get multiplicities for free using the standard natural numbers type:

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

It’s straightforward to work with these multiplicities:

```
copy : (x : A) → ι x ^ 2  ⊩ A × A
copy x = (x , x) , lax,
```

A natural resource algebra

We get multiplicities for free using the standard natural numbers type:

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

It's straightforward to work with these multiplicities:

```
copy : (x : A) → ι x ^ 2  ⊩ A × A
copy x = (x , x) , lax,

compose : ((x : A) → ι x ^ n ⊩ B) → ((y : B) → ι y ^ m ⊩ C)
        → (x : A) → ι x ^ (n · m) ⊩ C
compose f g x = g (f x .fst) .fst , …
```

A natural resource algebra

We get multiplicities for free using the standard natural numbers type:

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

It’s straightforward to work with these multiplicities:

```
copy : (x : A) → ι x ^ 2  ⊩ A × A
copy x = (x , x) , lax,

compose : ((x : A) → ι x ^ n ⊩ B) → ((y : B) → ι y ^ m ⊩ C)
        → (x : A) → ι x ^ (n · m) ⊩ C
compose f g x = g (f x .fst) .fst , …
```

some work is necessary here…

A natural resource algebra

We get multiplicities for free using the standard natural numbers type:

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

It’s straightforward to work with these multiplicities:

```
copy : (x : A) → ι x ^ 2  ⊩ A × A
copy x = (x , x) , lax,

compose : ((x : A) → ι x ^ n ⊩ B) → ((y : B) → ι y ^ m ⊩ C)
        → (x : A) → ι x ^ (n · m) ⊩ C
compose f g x = g (f x .fst) .fst , …
```

some work is necessary here…

```
copytwice : (x : A) → ι x ^ 4 ⊩ (A × A) × (A × A)
copytwice = compose copy copy
```

…but this directly computes!

Programming linearly with lists

We introduce productions for the lists constructors:

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

Programming linearly with lists

We introduce productions for the lists constructors:

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

That’s all we need to incorporate lists in our system!

Programming linearly with lists

We introduce productions for the lists constructors:

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

That’s all we need to incorporate lists in our system!

```
safeHead : (xs : List A) → (y : A)
  → ι y ^ (if null xs then 1 else 0) ⊗ ι xs ⊩ A × List A
safeHead []        y = (y , []) , {Goal: (ι y ^ 1 ⊗ ι []) ▷ ι (y , []) }
safeHead (x :: xs) y = (x , xs) , {Goal: (ι y ^ 0 ⊗ ι (x :: xs)) ▷ ι (x , xs) }
```

Programming linearly with lists

We introduce productions for the lists constructors:

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

That's all we need to incorporate lists in our system!

```
safeHead : (xs : List A) → (y : A)
  → ι y ^ (if null xs then 1 else 0) ⊗ ι xs ⊩ A × List A
safeHead []        y = (y , []) , lax,
safeHead (x :: xs) y = (x , xs) , {Goal: (ι y ^ 0 ⊗ ι (x :: xs)) ▷ ι (x , xs) }
```

Programming linearly with lists

We introduce productions for the lists constructors:

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

That’s all we need to incorporate lists in our system!

```
safeHead : (xs : List A) → (y : A)
  → ι y ^ (if null xs then 1 else 0) ⊗ ι xs ⊩ A × List A
safeHead []        y = (y , []) , lax,
safeHead (x :: xs) y = (x , xs) , lax, ∘ opl::
```

Programming linearly with lists

We introduce productions for the lists constructors:

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

That's all we need to incorporate lists in our system!

```
safeHead : (xs : List A) → (y : A)
  → ι y ^ (if null xs then 1 else 0) ⊗ ι xs ⊩ A × List A
safeHead []        y = (y , []) , lax,
safeHead (x :: xs) y = (x , xs) , lax, ∘ opl::

foldr : ((x : A) → (b : B) → ι b ⊗ ι x ⊗ Δ₁ ⊩ B) → Δ₀ ⊩ B
    → (xs : List A) → Δ₀ ⊗ Δ₁ ^ (length xs) ⊗ ι xs ⊩ B
foldr f (z , δ) [] =
foldr f z (x :: xs) =
```

Programming linearly with lists

We introduce productions for the lists constructors:

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

That's all we need to incorporate lists in our system!

```
safeHead : (xs : List A) → (y : A)
  → ι y ^ (if null xs then 1 else 0) ⊗ ι xs ⊩ A × List A
safeHead []        y = (y , []) , lax,
safeHead (x :: xs) y = (x , xs) , lax, ∘ opl::

foldr : ((x : A) → (b : B) → ι b ⊗ ι x ⊗ Δ₁ ⊩ B) → Δ₀ ⊩ B
    → (xs : List A) → Δ₀ ⊗ Δ₁ ^ (length xs) ⊗ ι xs ⊩ B
foldr f (z , δ) [] = {Goal: Δ₀ ⊗ Δ₁ ^ length [] ⊗ ι [] ⊩ B }
foldr f z (x :: xs) =
```

Programming linearly with lists

We introduce productions for the lists constructors:

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

That’s all we need to incorporate lists in our system!

```
safeHead : (xs : List A) → (y : A)
  → ι y ^ (if null xs then 1 else 0) ⊗ ι xs ⊩ A × List A
safeHead []        y = (y , []) , lax,
safeHead (x :: xs) y = (x , xs) , lax, ∘ opl::

foldr : ((x : A) → (b : B) → ι b ⊗ ι x ⊗ Δ₁ ⊩ B) → Δ₀ ⊩ B
    → (xs : List A) → Δ₀ ⊗ Δ₁ ^ (length xs) ⊗ ι xs ⊩ B
foldr f (z , δ) [] =  z , δ ⊗ᶠ opl[]
foldr f z (x :: xs) =
```

Programming linearly with lists

We introduce productions for the lists constructors:

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

That's all we need to incorporate lists in our system!

```
safeHead : (xs : List A) → (y : A)
  → ι y ^ (if null xs then 1 else 0) ⊗ ι xs ⊩ A × List A
safeHead []        y = (y , []) , lax,
safeHead (x :: xs) y = (x , xs) , lax, ∘ opl::

foldr : ((x : A) → (b : B) → ι b ⊗ ι x ⊗ Δ₁ ⊩ B) → Δ₀ ⊩ B
    → (xs : List A) → Δ₀ ⊗ Δ₁ ^ (length xs) ⊗ ι xs ⊩ B
foldr f (z , δ) [] =  z , δ ⊗ᶠ opl[]
foldr f z (x :: xs) =  f x @ foldr f z xs …
```

((x : A) → ι x ⊗ Δ₀ ⊩ B) → Δ₁ ⊩ A → Δ₀ ⊗ Δ₁ ⊩ B

Recap

- Supplies as *finite multisets of pointed types* are a useful notion of resource, dependent pairs allow us to define a linear judgment *inside* type theory.

```
Δ ⊩ A = Σ[ a ∈ A ] (Δ ▷ ι a)
```

- Productions capture which supplies have the *same multiset of free variables*. Incorporate datatypes by stipulating productions for each constructor.
  → quantitative elimination principles are *derived* using dependent elimination!

- Dependent types are naturally part of the system.

- This is already practical for programming, for example it's easy to construct sorting algorithms. Simple tactic could automatically find most productions.

Leaving cubical behind

We can carry out our construction in any dependent type theory with Π and Σ:

$$\begin{array}{l} Tm \;:\; Cx^{op} \to \mathbf{Set} \\ \big\downarrow \pi \\ Ty \;:\; Cx^{op} \to \mathbf{Set} \end{array}$$

Leaving cubical behind

We can carry out our construction in any dependent type theory with Π and Σ:

$$\begin{array}{c} Tm \\ \downarrow \pi \\ Ty \end{array}$$

Leaving cubical behind

We can carry out our construction in any dependent type theory with Π and Σ:

$$\begin{array}{ccc} Tm & \xrightarrow{\eta} & Sp : Cx^{op} \to \mathbf{SMCat} \\ \downarrow{\scriptstyle \pi} & & \\ Ty & & \end{array}$$

Leaving cubical behind

We can carry out our construction in any dependent type theory with Π and Σ:

$$\begin{array}{ccc} Tm & \xrightarrow{\eta} & Sp : Cx^{op} \to \mathbf{SMCat} \\ \downarrow{\scriptstyle \pi} & & \\ Ty & & \end{array}$$

- $Sp(\Gamma)$ live in type theory ($Sp(\Gamma) \in Ty(\Gamma)$ etc.)
- $\eta(a) \otimes \eta(b) \simeq \eta(a, b)$ for any $a : A$ and $b : B(a)$

Leaving cubical behind

We can carry out our construction in any dependent type theory with Π and Σ:

$$
\begin{array}{ccl}
Tm & \xrightarrow{\quad \eta \quad} & Sp : Cx^{op} \to \mathbf{SMCat} \\
\downarrow{\scriptstyle \pi} & & \\
Ty & &
\end{array}
$$

- $Sp(\Gamma)$ live in type theory ($Sp(\Gamma) \in Ty(\Gamma)$ etc.)
- $\eta(a) \otimes \eta(b) \simeq \eta(a, b)$ for any $a : A$ and $b : B(a)$

Using this, we can define a linear judgment, giving rise to a two-step derivation:

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

Leaving cubical behind

We can carry out our construction in any dependent type theory with Π and Σ:

$$\begin{array}{ccl} Tm & \xrightarrow{\eta} & Sp : Cx^{op} \to \mathbf{SMCat} \\ \downarrow{\scriptstyle \pi} & & \\ Ty & & \end{array}$$

- $Sp(\Gamma)$ live in type theory ($Sp(\Gamma) \in Ty(\Gamma)$ etc.)
- $\eta(a) \otimes \eta(b) \simeq \eta(a, b)$ for any $a : A$ and $b : B(a)$

Using this, we can define a linear judgment, giving rise to a two-step derivation:

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

Can we internalise this structure? In other words, how to add function types?

Proposal for internalising $\Gamma \vdash \Delta \Vdash A$

Add two more things:

- exponentials $[\Delta_0, \Delta_1]$
- $\Lambda_{x:A}\Delta$ binding x in Δ

$Sp : Cx^{op} \to \mathbf{SM\underline{C}Cat}$

functor $\Lambda_A : Sp(\Gamma\,.\,A) \to Sp(\Gamma)$ that's right adjoint to context extension $Sp(\mathbf{p}_A) : Sp(\Gamma) \to Sp(\Gamma\,.\,A)$

Proposal for internalising $\Gamma \vdash \Delta \Vdash A$

Add two more things:

- exponentials $[\Delta_0, \Delta_1]$ $\quad$ $Sp : Cx^{op} \to \mathbf{SM\underline{C}Cat}$
- $\Lambda_{x:A}\Delta$ binding x in Δ $\quad$ functor $\Lambda_A : Sp(\Gamma \,.\, A) \to Sp(\Gamma)$ that's right adjoint to context extension $Sp(\mathbf{p}_A) : Sp(\Gamma) \to Sp(\Gamma \,.\, A)$

This allows us to define a type of dependent linear functions from A to B:

$$(x : A) \multimap B(x) := (x : A) \to B(x) \ , \ \lambda f \to \Lambda_{x:A}[\eta(x), \eta(f\,x)]$$

We can derive intuitive introduction and elimination rules for $(x : A) \multimap B(x)$.

Proposal for internalising $\Gamma \vdash \Delta \Vdash A$

Add two more things:

- exponentials $[\Delta_0, \Delta_1]$ — $Sp : Cx^{op} \to \mathbf{SM\underline{C}Cat}$
- $\Lambda_{x:A}\Delta$ binding x in Δ — functor $\Lambda_A : Sp(\Gamma\,.\,A) \to Sp(\Gamma)$ that's right adjoint to context extension $Sp(\mathbf{p}_A) : Sp(\Gamma) \to Sp(\Gamma\,.\,A)$

This allows us to define a type of dependent linear functions from A to B:

$$(x : A) \multimap B(x) := (x : A) \to B(x)\ ,\ \lambda f \to \Lambda_{x:A}[\eta(x), \eta(f\ x)]$$

(generalise η to dependent supplies for higher-order functions)

We can derive intuitive introduction and elimination rules for $(x : A) \multimap B(x)$.

Summary

- Adding symmetric monoidal structure to dependent type theory is useful.
  - This also happens with non-idempotent intersection types (De Carvalho, Ronchi della Rocca, Gardner), but more powerful base theory makes our life easier.
- Quantitative features come for free, multiplicities are (open) terms of type $\mathbb{N}$.
  - We can type many more programs than systems with static resource algebra (QTT, Graded TT, Linear Haskell). Observation due to Pierre-Marie Pedrót (*Dialectica the Ultimate*, talk at TLLA 2024).
- WIP: expand idea to incorporate *dependent linear function types*.
  Gives rise to a *dependent linear type theory* with *dependent multiplicities*.

https://github.com/maxdore/dltt/

Dependent linear functions

$$(x : A) \multimap B(x) := (x : A) \to B(x) \ , \ \lambda f \to \Lambda_{x:A}[\eta(x), \eta(f\ x)]$$

$$\frac{\Gamma, x : A \vdash \Delta \otimes \eta(x)^m \Vdash b : B(x)}{\Gamma \vdash \Delta \Vdash \lambda x\,.\,b : (x : A) \multimap^m B(x)} \multimap I \ (x \notin \Delta)$$

$$\frac{\Gamma \vdash \Delta_0 \Vdash f : (x : A) \multimap^m B(x) \qquad \Gamma \vdash \Delta_1 \Vdash a : A}{\Gamma \vdash \Delta_0 \otimes \Delta_1^m \Vdash f\ a : B(a)} \multimap E$$

Dependent linear type theory

We can define a type theory with linear dependent types using the following:

$$\begin{array}{ccc} Tm & \xrightarrow{\eta} & Sp : Cx^{op} \to \mathbf{SMCCat} \\ \Big\downarrow{\scriptstyle \pi} & & \\ Ty & & \end{array}$$

$$Sp(\Gamma) \underset{\Lambda_A}{\overset{Sp(\mathbf{p}_A)}{\rightleftarrows}} Sp(\Gamma . A) \qquad \bot$$

+ for Σ types: iso between $\eta(a) \otimes \eta(b)$ and $\eta(a, b)$ for any $a : A$ and $b : B(a)$

Linear types without finite multisets

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

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

Currying example

$$\dfrac{\dfrac{\dfrac{\dfrac{x : A, y : B(x) \vdash \Delta \Vdash f : \Pi^1_{\mathsf{pair}(x,y):\Sigma_A(B)}(C(y)) \qquad \dfrac{}{x : A, y : B(x) \vdash \eta(\mathsf{pair}(x,y)) \Vdash \mathsf{pair}(x,y) : \Sigma_A(B)}\,\textsc{Id}}{x : A, y : B(x) \vdash \Delta \otimes \eta(\mathsf{pair}(x,y)) \Vdash f(\mathsf{pair}(x,y)) : C(y)}\,\Pi\textsc{App}}{x : A, y : B(x) \vdash \Delta \otimes \eta(x) \otimes \eta(y) \Vdash f(\mathsf{pair}(x,y)) : C(y)}\,\omega_{\mathsf{pair}}}{x : A \vdash \Delta \otimes \eta(x) \Vdash \lambda y. f(\mathsf{pair}(x,y)) : \Pi^1_{B(x)}(C)}\,\Pi\mathrm{I}}{\vdash \Delta \Vdash \lambda x.\lambda y. f(\mathsf{pair}(x,y)) : \Pi^1_{x:A}(\Pi^1_{B(x)}(C))}\,\Pi\mathrm{I}$$