Type Theory

Lecture 2: Polymorphic Type Theory

H. Geuvers

Radboud University
Nijmegen, NL

OPLSS 2026 Summer 2026

Outline

Why Polymorphic λ-calculus?

- Simple type theory $\lambda\!\rightarrow$ is not very expressive
- In simple type theory, we can not 'reuse' a function.
  E.g. $\lambda x{:}\alpha.x : \alpha\!\rightarrow\!\alpha$ and $\lambda x{:}\beta.x : \beta\!\rightarrow\!\beta$.

We want to define functions that can treat types parametrically:
add types $\forall\alpha.A$:
Examples

- $\forall\alpha.\alpha\!\rightarrow\!\alpha$
  If $M : \forall\alpha.\alpha\!\rightarrow\!\alpha$, then M can map any type to itself.
- $\forall\alpha.\forall\beta.\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha$
  If $M : \forall\alpha.\forall\beta.\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha$, then M can take two inputs (of arbitrary types) and return a value of the first input type.
- $M : \forall\alpha.\alpha\!\rightarrow\!\alpha$, with $M\,\alpha = \mathrm{S}$ for $\alpha = \mathrm{Nat}$ and $M\,\alpha = \mathrm{Id}$ for $\alpha \neq \mathrm{Nat}$ . . . ?? No: this M is an overloaded polymorphic function, not a parametric polymorphic function.

Derivation rules of $\lambda 2$ with full (system F-style) polymorphism

$$\mathsf{Typ} := \mathsf{TVar} \mid (\mathsf{Typ} \to \mathsf{Typ}) \mid \forall \alpha.\mathsf{Typ}$$

1. Church style:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \lambda \alpha.M : \forall \alpha.A}\ \alpha \notin \mathsf{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall \alpha.A}{\Gamma \vdash M\,B : A[\alpha := B]}$$

2. Curry style:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall \alpha.A}\ \alpha \notin \mathsf{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall \alpha.A}{\Gamma \vdash M : A[\alpha := B]}$$

Examples

- $\lambda 2$ à la Church: $\lambda\alpha.\lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\forall\beta.\alpha{\to}\beta{\to}\alpha$.
- $\lambda 2$ à la Curry: $\lambda x.\lambda y.x : \forall\alpha.\forall\beta.\alpha{\to}\beta{\to}\alpha$.
- $\lambda 2$ à la Church: $z : \forall\alpha.\alpha{\to}\alpha \vdash \lambda\alpha.z\,(\alpha{\to}\alpha)\,(z\,\alpha) : \forall\alpha.\alpha{\to}\alpha$.
- $\lambda 2$ à la Curry: $z : \forall\alpha.\alpha{\to}\alpha \vdash z\,z : \forall\alpha.\alpha{\to}\alpha$.
- $\lambda 2$ à la Church: $\lambda x{:}(\forall\alpha.\alpha).\lambda y{:}A.xB : (\forall\alpha.\alpha){\to}A{\to}B$.
- $\lambda 2$ à la Curry: $\lambda x.\lambda y.x : (\forall\alpha.\alpha){\to}A{\to}B$.

More examples of typing in $\lambda 2$

Abbreviate $\bot := \forall\alpha.\alpha$, $\top := \forall\alpha.\alpha{\to}\alpha$.

- Curry $\lambda 2$: $\lambda x.xx : \bot{\to}\bot$
- Church $\lambda 2$: $\lambda x{:}\bot.x(\bot{\to}\bot)x : \bot{\to}\bot$.
- Church $\lambda 2$: $\lambda x{:}\bot.\lambda\alpha.x(\alpha{\to}\alpha)(x\alpha) : \bot{\to}\bot$.

LEMMA. In $\lambda 2$ à la Curry, every λ-term in normal form is typable by giving all variables type $\bot$.

Exercises:

- Verify that in Church $\lambda 2$: $\lambda x{:}\top.x\top x : \top{\to}\top$.
- Verify that in Curry $\lambda 2$: $\lambda x.xx : \top{\to}\top$
- Find a type in Curry $\lambda 2$ for $\lambda x.x\,x\,x$
- Find a type in Curry $\lambda 2$ for $\lambda x.x\,x\,(\lambda y.y\,y)\,x\,x$

Derivation rules with types recorded in the context

Contexts Γ consist of declarations $x : A$ and $\alpha : *$
Condition:

in $\Gamma, x : A, \Delta$ we require that $\mathrm{FV}(A)$ are declared in Γ

1. Curry style:

$$\frac{\Gamma, \alpha : * \vdash M : A}{\Gamma \vdash M : \forall \alpha . A} \qquad \frac{\Gamma \vdash M : \forall \alpha . A}{\Gamma \vdash M : A[\alpha := B]} \text{ if } \mathrm{FV}(B) \text{ declared in } \Gamma$$

2. Church style:

$$\frac{\Gamma, \alpha : * \vdash M : A}{\Gamma \vdash \lambda \alpha . M : \forall \alpha . A} \qquad \frac{\Gamma \vdash M : \forall \alpha . A}{\Gamma \vdash MB : A[\alpha := B]} \text{ if } \mathrm{FV}(B) \text{ declared in } \Gamma$$

Fitch style $\lambda 2$ derivations

The Fitch style presentation of $\lambda 2$.

1	$\alpha : *$	
2	...	
3	...	
4	$M : A$	
5	$\lambda\alpha.M : \forall\alpha.A$	abs, 1, 4

1	...	
2	...	
3	$M : \forall\alpha.A$	
4	...	
5	...	
6	...	
7	$M\,B : A[\alpha := B]$	app, 3

(†) if FV(B) are declared

Erasure from $\lambda 2$ à la Church to $\lambda 2$ à la Curry

$$
\begin{array}{lcllcl}
|x| & := & x & & & \\
|\lambda x{:}A.M| & := & |\lambda x.M| & |\lambda \alpha.M| & := & |M| \\
|MN| & := & |M|\,|N| & |MA| & := & |M|
\end{array}
$$

THEOREM If $\Gamma \vdash M : A$ in $\lambda 2$ à la Church, then $\Gamma \vdash |M| : A$ in $\lambda 2$ à la Curry.

THEOREM If $\Gamma \vdash P : A$ in $\lambda 2$ à la Curry, then there is an M such that $|M| \equiv P$ and $\Gamma \vdash M : A$ in $\lambda 2$ à la Church.

PROOF. By induction on the derivation of $\Gamma \vdash M : A$.

THEOREM[Wells 1993] In $\lambda 2$ à la Curry, type checking is undecidable

Derivation rules for Weak (ML-style) polymorphism

Typ : add $\forall\alpha_1.\ldots.\forall\alpha_n.A$ for A a $\lambda{\to}$-type.

1. Curry style:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall\alpha.A}\ \alpha \notin \mathrm{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall\alpha.A}{\Gamma \vdash M : A[\alpha := B]}\ \text{for } B \text{ a } \lambda{\to}\text{-type}$$

2. Church style:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.A}\ \alpha \notin \mathrm{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall\alpha.A}{\Gamma \vdash MB : A[\alpha := B]}\ \text{for } B \text{ a } \lambda{\to}\text{-type}$$

- $\forall$ only occurs on the outside and is therefore usually left out: "all type variables are implicitly universally quantified"
- With weak polymorphism, type checking is still decidable: the principal types algorithm still works.

Derivation rules for Weak (ML-style) polymorphism

NB! Also the abstraction rule is restricted to $\lambda\!\!\rightarrow$-types:

1. Curry style:

$$\frac{\Gamma, x : B \vdash M : A}{\Gamma \vdash \lambda x.M : B\!\rightarrow\! A} \text{ for } A, B\ \lambda\!\!\rightarrow\text{-types}$$

2. Church style:

$$\frac{\Gamma, x : B \vdash M : A}{\Gamma \vdash \lambda x{:}B.M : B\!\rightarrow\! A} \text{ for } A, B\ \lambda\!\!\rightarrow\text{-types}$$

Examples

In Weak polymorphism:

- $\lambda 2$ à la Church: $z : \forall\alpha.\alpha{\to}\alpha \vdash \lambda\alpha.z\,(\alpha{\to}\alpha)\,(z\,\alpha) : \forall\alpha.\alpha{\to}\alpha$.
- $\lambda 2$ à la Curry: $z : \forall\alpha.\alpha{\to}\alpha \vdash z\,z : \forall\alpha.\alpha{\to}\alpha$.
- But NOT $\vdash \lambda z.z\,z : \ldots$

Let polymorphism in ML

To regain some of the "full polymorphism", ML has let polymorphism

$$\frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash \mathbf{let}\, x = M \,\mathbf{in}\, N : B} \text{ for } B \text{ a } \lambda{\to}\text{-type}, A \text{ a } \lambda 2\text{-type}$$

This allows the formation of a β-redex

$$(\lambda x{:}A.N)M$$

for A a polymorphic type.
But not $\lambda x{:}A.N : A{\to}B$

Recall: Important Properties

$\Gamma \vdash M : A?$	TCP
$\Gamma \vdash M : ?$	TSP
$\vdash ? : A$	TIP

Properties of polymorphic λ-calculus

- TIP is undecidable, TCP and TSP are equivalent.
TCP	à la Church	à la Curry
ML-style	decidable	decidable
System F-style	decidable	undecidable

  With full polymorphism (system F), untyped terms contain too little information to compute the type.

Meta properties of $\lambda 2$

- For $\lambda 2$ à la Church: Uniqueness of types
  If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A = B$.
- Subject Reduction
  If $\Gamma \vdash M : A$ and $M \to_{\beta\eta} N$, then $\Gamma \vdash N : A$.
- Strong Normalization
  If $\Gamma \vdash M : A$, then all $\beta\eta$-reductions from M terminate.

Formulas-as-types for $\lambda 2$

There is a formulas-as-types isomorphism between $\lambda 2$ and second order proposition logic, $\mathrm{PROP2}$
Derivation rules of $\mathrm{PROP2}$:

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall \alpha.A}\ \alpha \notin \mathsf{FV}(\Gamma) \qquad \frac{\Gamma \vdash \forall \alpha.A}{\Gamma \vdash A[\alpha := B]}$$

NB This is constructive second order proposition logic:

$$\forall \alpha.\forall \beta.((\alpha \to \beta) \to \alpha) \to \alpha \text{ Peirce's law}$$

is not derivable.

Definability of the other connectives

$$
\begin{array}{rcl}
\bot & := & \forall\alpha.\alpha \\
A\wedge B & := & \forall\alpha.(A\to B\to\alpha)\to\alpha \\
A\vee B & := & \forall\alpha.(A\to\alpha)\to(B\to\alpha)\to\alpha \\
\exists\alpha.A & := & \forall\beta.(\forall\alpha.A\to\beta)\to\beta
\end{array}
$$

and all the standard constructive derivation rules are derivable.
Example ($\wedge$-elimination):

$$
\dfrac{\dfrac{\forall\alpha.(A\to B\to\alpha)\to\alpha}{(A\to B\to A)\to A} \quad \dfrac{\dfrac{[A]^1}{B\to A}}{A\to B\to A}\,1}{A}
$$

Definability of connectives and derivation rules

$$\begin{array}{rcl} \bot & := & \forall\alpha.\alpha \\ A\wedge B & := & \forall\alpha.(A\to B\to\alpha)\to\alpha \\ A\vee B & := & \forall\alpha.(A\to\alpha)\to(B\to\alpha)\to\alpha \\ \exists\alpha.A & := & \forall\beta.(\forall\alpha.A\to\beta)\to\beta \end{array}$$

Example ($\wedge$-elimination) with λ-terms:

$$\dfrac{\dfrac{M : \forall\alpha.(A\to B\to\alpha)\to\alpha}{MA : (A\to B\to A)\to A} \quad \dfrac{\dfrac{[x : A]^1}{\lambda y{:}B.x : B\to A}}{\lambda x{:}A.\lambda y{:}B.x : A\to B\to A}\,1}{MA(\lambda x{:}A.\lambda y{:}B.x) : A}$$

So the following term is a 'witness' for the $\wedge$-elimination.

$$\lambda z{:}A\wedge B.z\,A\,(\lambda x{:}A.\lambda y{:}B.x) : (A\wedge B)\to A$$

Data types in $\lambda 2$

$$\mathsf{Nat} := \forall\alpha.\alpha{\to}(\alpha{\to}\alpha){\to}\alpha$$

This type uses the encoding of natural numbers as Church numerals

$$n \mapsto c_n := \lambda x.\lambda f.f(\ldots(f\,x)) \quad n\text{-times } f$$

- $0 := \lambda\alpha.\lambda x{:}\alpha.\lambda f{:}\alpha{\to}\alpha.x$
- $\mathrm{S} := \lambda n{:}\mathsf{Nat}.\lambda\alpha.\lambda x{:}\alpha.\lambda f{:}\alpha{\to}\alpha.f(n\,\alpha\,x\,f)$
- Iteration: if $c : A$ and $g : A{\to}A$, then $\mathrm{It}\,c\,g : \mathsf{Nat}{\to}A$ is defined as

  $$\lambda n{:}\mathsf{Nat}.n\,A\,c\,g$$

  We have $\mathrm{It}\,c\,g\,n = g(\ldots(g\,c))$ (n times g), i.e.

$$\mathrm{It}\,c\,g\,0 = c \quad \text{and} \quad \mathrm{It}\,c\,g\,(\mathrm{S}\,x) = g(\mathrm{It}\,c\,g\,x)$$

Examples

- Addition

$$\mathsf{Plus} := \lambda m{:}\mathsf{Nat}.\lambda n{:}\mathsf{Nat}.\mathrm{It}\, m\, \mathrm{S}\, n$$

  or $\mathsf{Plus} := \lambda m{:}\mathsf{Nat}.\lambda n{:}\mathsf{Nat}.n\, \mathsf{Nat}\, m\, \mathrm{S}$

- Multiplication

$$\mathsf{Mult} := \lambda m{:}\mathsf{Nat}.\lambda n{:}\mathsf{Nat}.\mathrm{It}\, 0\, (\lambda x{:}\mathsf{Nat}.\mathsf{Plus}\, m\, x)\, n$$

- Predecessor is difficult!
  This requires defining primitive recursion in terms of iteration.
  As a consequence:

$$\mathsf{Pred}(n+1) \twoheadrightarrow_\beta n$$

  in a number of steps of $O(n)$.

Iteration, the more general picture

Nat is a data-type with two constructors $0 : \mathrm{Nat}$ and $\mathrm{S} : \mathrm{Nat} \to \mathrm{Nat}$. This implies the Nat-iteration scheme for defining functions $f : \mathrm{Nat} \to D$ (for any type D).

LEMMA[Nat-iteration] If $d : D$ and $g : D \to D$, then there is a function $f : \mathrm{Nat} \to D$ satisfying

$$\begin{aligned} f\,0 &= d \\ f\,(\mathrm{S}\,x) &= g\,(f\,x) \end{aligned}$$

In $\lambda 2$, this f, also called $\mathrm{It}\,d\,g$, can be defined as $\lambda n{:}\mathrm{Nat}.n\,D\,d\,g$.

The other way around: if I want to have a function $h : \mathrm{Nat} \to D$ that I can specify by case distinction, satisfying these equations:

$$\begin{aligned} h\,0 &= d \\ h\,(\mathrm{S}\,x) &= g\,(h\,x) \end{aligned}$$

Then I have it, because I can take $h := \lambda n{:}\mathrm{Nat}.n\,D\,d\,g$.

Examples of Nat-iteration

NAT-ITERATION in $\lambda 2$: if $d : D$ and $g : D \to D$, then $f := \lambda n{:}\mathsf{Nat}.n\, D\, d\, g$ satisfies

$$\begin{aligned} f\,0 &= d \\ f\,(\mathrm{S}\,x) &= g\,(f\,x) \end{aligned}$$

We derive Plus and Mult using Nat-iteration.

$$\begin{aligned} \mathsf{Plus}\,m\,0 &= m \\ \mathsf{Plus}\,m\,(\mathrm{S}\,x) &= \mathrm{S}\,(\mathsf{Plus}\,m\,x) \end{aligned}$$

So we can take $\mathsf{Plus}\,m := \lambda n{:}\mathsf{Nat}.n\,\mathsf{Nat}\,m\,\mathrm{S}$ and we define $\mathsf{Plus} := \lambda m, n{:}\mathsf{Nat}.n\,\mathsf{Nat}\,m\,\mathrm{S}$

$$\begin{aligned} \mathsf{Mult}\,m\,0 &= 0 \\ \mathsf{Mult}\,m\,(\mathrm{S}\,x) &= \mathsf{Plus}\,m\,(\mathsf{Mult}\,m\,x) \end{aligned}$$

So we define $\mathsf{Mult} := \lambda m, n{:}\mathsf{Nat}.n\,\mathsf{Nat}\,0\,(\lambda y{:}\mathsf{Nat}.\mathsf{Plus}\,m\,y)$

Data types in $\lambda 2$ ctd.

$$\mathsf{List}_A := \forall\alpha.\alpha{\to}(A{\to}\alpha{\to}\alpha){\to}\alpha$$

The type of lists over A uses the following encoding:

$$[a_1, a_2, \ldots, a_n] \mapsto \lambda x.\lambda f.f\ a_1\,(f\ a_2\,(\ldots(f\ a_n\,x))) \quad n\text{-times } f$$

- $\mathsf{Nil} := \lambda\alpha.\lambda x{:}\alpha.\lambda f{:}A{\to}\alpha{\to}\alpha.x$
- $\mathsf{Cons} := \lambda a{:}A.\lambda\ell{:}\mathsf{List}_A.\lambda\alpha.\lambda x{:}\alpha.\lambda f{:}A{\to}\alpha{\to}\alpha.f\ a\,(\ell\,\alpha\,x\,f)$
- Iteration: if $c : A$ and $g : A{\to}A{\to}A$, then $\mathrm{It}\,c\,g : \mathsf{List}_A{\to}A$ is defined as

  $$\lambda\ell{:}\mathsf{List}_A.\ell\,A\,c\,g$$

  We have $\mathrm{It}\,c\,g\,[a_1, \ldots, a_n] = g\ a_1\,(\ldots(g\ a_n\,c))$ (n times g), that is:

$$\mathrm{It}\,c\,g\,\mathsf{Nil} = c \quad \text{and} \quad \mathrm{It}\,c\,g\,(\mathsf{Cons}\,a\,l) = g\ a\,(\mathrm{It}\,c\,g\,l)$$

The List-iteration scheme

List_A has constructors Nil : List_A and Cons : $A \to \text{List}_A \to \text{List}_A$.
This implies a List-iteration scheme for defining functions $f : \text{List}_A \to D$ (for any type D).

LEMMA[List-iteration] If $d : D$ and $g : A \to D \to D$, then there is a function $f : \text{List}_A \to D$ satisfying

$$\begin{aligned} f\,\text{Nil} &= d \\ f\,(\text{Cons}\,a\,x) &= g\,a\,(f\,x) \end{aligned}$$

In $\lambda 2$, this f can be defined as $\lambda\ell{:}\text{List}_A.\ell\,D\,d\,g$ (which is also written as $\text{It}\,d\,g$).

Example: the length of a list, length : $\text{List}_A \to \text{Nat}$. It satisfies

$$\begin{aligned} \text{length}\,\text{Nil} &= 0 \\ \text{length}\,(\text{Cons}\,a\,k) &= \text{S}\,(\text{length}\,k) \end{aligned}$$

So we can define length := $\text{It}\,0\,(\lambda a{:}A.\lambda n{:}\text{Nat}.\text{S}\,n)$, or in full detail length := $\lambda\ell{:}\text{List}_A.\ell\,\text{Nat}\,0\,(\lambda a{:}A.\lambda n{:}\text{Nat}.\text{S}\,n)$.

Example: Map

Map is one of the standard functional programs over List. Given $h : A \to B$, we have

$$\mathsf{Map}\, h : \mathsf{List}_A \to \mathsf{List}_B$$

which applies h to all elements in the input-list.
We can specify it via these equations:

$$\begin{aligned} \mathsf{Map}\, h\, \mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{Map}\, h\, (\mathsf{Cons}\, a\, k) &= \mathsf{Cons}\, (h\, a)(\mathsf{Map}\, h\, k) \end{aligned}$$

So we can define it using List-iteration:

$$\mathsf{Map} := \lambda h{:}A \to B.\mathrm{It}\, \mathsf{Nil}\, (\lambda x{:}A.\lambda k{:}\mathsf{List}_B.\mathsf{Cons}\, (h\, x)\, k).$$

Or in full detail

$$\mathsf{Map} := \lambda h{:}A \to B.\lambda \ell{:}\mathsf{List}_A.\ell\, \mathsf{List}_B\, \mathsf{Nil}(\lambda x{:}A.\lambda k{:}\mathsf{List}_B.\mathsf{Cons}\, (h\, x)\, k).$$

Many data-types can be defined in $\lambda 2$

- Product of two data-types: $A \times B := \forall \alpha.(A \to B \to \alpha) \to \alpha$
- Sum of two data-types: $A + B := \forall \alpha.(A \to \alpha) \to (B \to \alpha) \to \alpha$
- Unit type: $\mathrm{Unit} := \forall \alpha.\alpha \to \alpha$
- Binary trees with nodes in A and leaves in B:
  $\mathrm{Tree}_{A,B} := \forall \alpha.(B \to \alpha) \to (A \to \alpha \to \alpha \to \alpha) \to \alpha$
  $\mathrm{Tree}_{A,B}$ has two constructors, $\mathrm{leaf} : B \to \mathrm{Tree}_{A,B}$ and
  $\mathrm{join} : \mathrm{Tree}_{A,B} \to \mathrm{Tree}_{A,B} \to A \to \mathrm{Tree}_{A,B}$

Exercise:

- Define $\mathrm{inl} : A \to A + B$
- Define the first projection: $\pi_1 : A \times B \to A$
- Define $\mathrm{leaf} : B \to \mathrm{Tree}_{A,B}$ and
  $\mathrm{join} : \mathrm{Tree}_{A,B} \to \mathrm{Tree}_{A,B} \to A \to \mathrm{Tree}_{A,B}$
- Give the Tree-iteration scheme for $\mathrm{Tree}_{A,B}$ and define
  $h : \mathrm{Tree}_{A,B} \to \mathrm{Nat}$ that counts the number of leaves of a tree.

Weakly initial algebras are definable

We consider type schemes $F(\alpha)$, a type containing a free type variable α, that are "functorial type schemes": For $f : A \to B$ we have $F(f) : F(A) \to F(B)$.

THEOREM For $F(\alpha)$ a "functorial type scheme" in $\lambda 2$ we can define a weakly initial F-algebra as

$$\mu F := \forall \alpha.(F(\alpha) \to \alpha) \to \alpha.$$

PROOF. We have to define cons and $\mathrm{It}\, g$ (for every $g : F(B) \to B$) to make this diagram commute.

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{\mathrm{cons}} & \mu F \\ {\scriptstyle F(\mathrm{It}\, g)}\downarrow & & \downarrow{\scriptstyle \mathrm{It}\, g} \\ F(B) & \xrightarrow{g} & B \end{array}$$

- For $g : F(B) \to B$, define $\mathrm{It}\, g := \lambda x : \mu F.x\, B\, g$
- $\mathrm{cons} := \lambda y : F(\mu F).\lambda \alpha.\lambda f : F(\alpha) \to \alpha.f\, (F(\mathrm{It}\, f)\, y)$.
- Then indeed $\mathrm{It}\, g \circ \mathrm{cons} = g \circ F(\mathrm{It}\, g)$. □

Weakly terminal coalgebras are definable

THEOREM For $F(\alpha)$ a "functorial type scheme" in $\lambda 2$ we can define a weakly terminal F-coalgebra as

$$\nu F := \exists \alpha.(\alpha \times (\alpha \to F(\alpha))).$$

PROOF. We have to define dest and $\mathrm{Coit}\, g$ (for every $g : B \to F(B)$) to make this diagram commute.

$$\begin{array}{ccc} B & \xrightarrow{g} & F(B) \\ {\scriptstyle \mathrm{Coit}\, g}\downarrow & & \downarrow{\scriptstyle F(\mathrm{Coit}\, g)} \\ \nu F & \xrightarrow{\mathrm{dest}} & F(\nu F) \end{array}$$

- For $g : B \to F(B)$, define $\mathrm{Coit}\, g := \lambda x : B.\langle B, \langle x, g\rangle\rangle$.
- $\mathrm{dest} :=$
  $$\lambda y : \nu F.\exists\text{-el}\, y\, (\lambda\alpha.\lambda z : \alpha.\lambda h : \alpha \to F(\alpha).F(\mathrm{Coit}\, h)\, (h\, z)).$$
- Then indeed $\mathrm{dest} \circ \mathrm{Coit}\, g = F(\mathrm{Coit}\, g) \circ g$. □

Example: Streams over A

We define $\mathsf{Str}_A := \exists\alpha.\alpha \times (\alpha \to A) \times (\alpha \to \alpha)$

Exercise

1. Define $\mathrm{hd} : \mathsf{Str}_A \to A$, taking the head of a stream.
2. Define $\mathrm{tl} : \mathsf{Str}_A \to \mathsf{Str}_A$, taking the tail of a stream.
3. For B a type with $h : B \to A$ and $t : B \to B$, define $\mathrm{Coit}\, h\, t : B \to \mathsf{Str}_A$, and
4. Check that your definitions satisfy the proper equations for a weakly terminal coalgebra:

$$\begin{aligned} \mathrm{hd}\,(\mathrm{Coit}\, h\, t\, x) &=_\beta h\, x \\ \mathrm{tl}\,(\mathrm{Coit}\, h\, t\, x) &=_\beta \mathrm{Coit}\, h\, t\, (t\, x) \end{aligned}$$

Questions?