Inductive and Coinductive Data Types in Typed Lambda Calculus Revisited

Herman Geuvers

Radboud University Nijmegen
and
Eindhoven University of Technology
The Netherlands

TLCA 2015, Warsaw, July 2
Contents

- Looping a function
- The categorical picture: initial algebras
- Initial algebras in syntax
- Church and Scott data types
- Dualizing: final co-algebras
- Extracting programs from proofs
- Related Work
How a programmer may look at a recursive function

P: Given $f : A \rightarrow A$, I want to loop $f$ until it stops
T: But if you keep calling $f$, it will never stop
P: Ehh...
T: You mean that you have a function $f : A \rightarrow A + B$, and if you get a value in $A$, you continue, and if you get a value in $B$, you stop?
P: That's right! And the function I want to define in the end is from $A$ to $B$ anyway!
T: So you want

\[
\begin{align*}
    f : A &\rightarrow A + B \\
    \text{loop } f : A &\rightarrow B
\end{align*}
\]

satisfying

\[
\text{loop } f \ a = \text{case } f \ a \text{ of } (\text{inl } a' \Rightarrow \text{loop } f \ a') (\text{inr } b \Rightarrow b)
\]
Can we dualize this looping?

\[
f : A \to A + B
\]

\[
\text{loop } f : A \to B
\]

\[
\text{loop } f \ a = \text{case } f \ a \text{ of } (\text{inl } a' \Rightarrow \text{loop } f \ a')(\text{inr } b \Rightarrow b)
\]

Dually:

\[
f : A \times B \to A
\]

\[
\text{coloop } f : B \to A
\]

satisfying

\[
\text{coloop } f \ b = f \langle \text{coloop } f \ b, b \rangle.
\]

\text{coloop } f \ b \text{ is a fixed point of } \lambda a. f \langle a, b \rangle.

So loops and fixed-points are dual. (Filinski 1994)
Inductive and coinductive data types

We want

- terminating functions
- pattern matching on data
- profit from duality
The categorical picture

Syntax for inductive data types is derived from categorical semantics:
Initial $F$-algebra: $(\mu F, \text{in})$ s.t. $\forall (B, g), \exists! h$ such that the diagram commutes:

\[
\begin{array}{ccc}
F(\mu F) & \xrightarrow{\text{in}} & \mu F \\
Fh & & Fh \\
FB & \xrightarrow{g} & B
\end{array}
\]

- Due to the uniqueness: in is an isomorphism, so it has an inverse out : $\mu F \rightarrow F(\mu F)$.
In case $FX := 1 + X$, $\mu F = \text{Nat}$ and out is basically the predecessor.
Inductive types are initial algebras

We derive the iteration scheme: a function definition principle + a reduction rule. The $h$ in the diagram is called $\text{It} \ g$

$g : F B \to B$

$\text{It} \ g : \mu F \to B$

with $\text{It} \ g \ (\text{in} \ x) \to g \ (F(\text{It} \ g) \ x)$

In case $FX := 1 + X$, $\mu F = \text{Nat}$ and in decomposes in $0 : \text{Nat}$, $\text{Succ} : \text{Nat} \to \text{Nat}$;

$d : D \to D \to D$ with $\text{It} \ d \ f \ 0 \to d$

$\text{It} \ d \ f \ (\text{Succ} \ x) \to f \ (\text{It} \ d \ f \ x)$
Given \( d : B, \ g : \text{Nat} \times B \rightarrow B \), I want \( h : \text{Nat} \rightarrow B \) satisfying

\[
\begin{align*}
    h0 & \rightarrow d \\
    h(\text{Succ } x) & \rightarrow g \times (h \times x)
\end{align*}
\]
Defining primitive recursion

Given $d : B$, $g : \text{Nat} \times B \to B$

We derive the primitive recursion scheme:

- From uniqueness it follows that $h_1 = \text{Id}$ (identity)
- From that we derive for $h_2$:

$$
\begin{align*}
  d : B & \quad g : \text{Nat} \times B \to B \\
  h_2 : \text{Nat} \to B
\end{align*}
$$

with

$$
\begin{align*}
  h_2 0 & \Rightarrow d \\
  h_2(\text{Succ} \times x) & \Rightarrow g \times (h_2 \times)
\end{align*}
$$
The induction proof principle also follows from this

Given \( p_0 : P \, 0 \), \( p_S : \forall x : \text{Nat}.\, P \, x \to P \,(\text{Succ} \, x) \)

\[
\begin{array}{c}
1 + \text{Nat} \\
\downarrow \text{Id} + \langle h_1, h_2 \rangle
\end{array}
\quad [0, \text{Succ}]
\quad \begin{array}{c}
\text{Nat} \\
\downarrow \langle h_1, h_2 \rangle
\end{array}
\]

\[
egin{array}{c}
1 + (\sum x : \text{Nat}.\, P \, x)
\end{array}
\quad [\langle 0, p_0 \rangle, \text{Succ} \times p_S]
\quad \begin{array}{c}
\sum x : \text{Nat}.\, P \, x
\end{array}
\]

We derive the induction scheme:

- From uniqueness it follows that \( h_1 = \text{Id} \) (identity)
- From that we derive for \( h_2 \):

\[
\begin{array}{c}
p_0 : P \, 0 \quad p_S : \forall x : \text{Nat}.\, P \, x \to P \,(\text{Succ} \, x) \end{array}
\quad \begin{array}{c}
h_2 : \forall x : \text{Nat}.\, P \, x
\end{array}
\]
In syntax, inductive types are only weakly initial algebras

\[
\begin{array}{ccc}
F(\mu F) & \xrightarrow{\text{in}} & \mu F \\
\downarrow Fh & & \downarrow h \\
FB & \xrightarrow{g} & B
\end{array}
\]

- In syntax we only have weakly initial algebras: \(\exists\), but not \(\exists!\).
- So we get out and primitive recursion only in a weak slightly twisted form.
- We can derive the primitive recursion scheme via this diagram.
Consider the following Primitive Recursion scheme for \( \text{Nat} \). (Let \( D \) be any type.)

\[
\begin{align*}
\text{Rec } d \ f \ 0 & \rightarrow \ d \\
\text{Rec } d \ f \ (\text{Succ } x) & \rightarrow \ f \ x \ (\text{Rec } d \ f \ x)
\end{align*}
\]

One can define \( \text{Rec} \) in terms of \( \text{It} \). (This is what Kleene found out at the dentist.)

\[
\begin{align*}
\langle 0, \ d \rangle : \text{Nat} \times D & \quad \lambda z.\langle \text{Succ } z_1, f \ z_1 \ z_2 \rangle : \text{Nat} \times D \rightarrow \text{Nat} \times D \\
\text{It } \langle 0, \ d \rangle \ \lambda z.\langle \text{Succ } z_1, f \ z_1 \ z_2 \rangle & : \text{Nat} \rightarrow \text{Nat} \times D \\
\lambda p.\ (\text{It } \langle 0, \ d \rangle \ \lambda z.\langle \text{Succ } z_1, f \ z_1 \ z_2 \rangle \ p)_2 & : \text{Nat} \rightarrow D
\end{align*}
\]

\( \langle -, - \rangle \) denotes the pair; \((-)_1\) and \((-)_2\) denote projections.
Primitive recursion in terms of iteration

Problems:

- Only works for values. For the now definable predecessor $P$ we have:

  $P(Succ^{n+1} 0) \rightarrow Succ^n 0$

  but not $P(Succ x) = x$

- Computationally inefficient

  $P(Succ^{n+1} 0) \rightarrow Succ^n 0$ in linear time
Iterative, primitive recursive algebras, algebras with case

- An iterative $T$-algebra (also weakly initial $T$-algebra) is a triple $(A, \text{in}, \text{It})$
- An $T$-algebra with case is a triple $(A, \text{in}, \text{Case})$
- A primitive recursive $T$-algebra is a triple $(A, \text{in}, \text{Rec})$
Defining data types in lambda calculus

- Iterative algebras can be encoded as Church data types
- Algebras with case can be encoded as Scott data types
- Primitive recursive algebras can be encoded as Church-Scott or Parigot data types
Church numerals

The most well-known Church data type

\[
\begin{align*}
\bar{0} & := \lambda x f . x \\
\bar{1} & := \lambda x f . f x \\
\bar{2} & := \lambda x f . f (f x)
\end{align*}
\]

\[
\begin{align*}
\bar{p} & := \lambda x f . f^p (x) \\
\text{Succ} & := \lambda n . \lambda x f . f (n x f)
\end{align*}
\]

- The Church data types have iteration as basis. The numerals are iterators.
- Iteration scheme for \( \mathbb{Nat} \). (Let \( D \) be any type.)

\[
\begin{align*}
d : D & \quad f : D \to D \\
\text{It } d f & : \mathbb{Nat} \to D
\end{align*}
\]

with

\[
\begin{align*}
\text{It } d f \bar{0} & \to d \\
\text{It } d f (\text{Succ } x) & \to f (\text{It } d f x)
\end{align*}
\]

- Advantage: quite a bit of well-founded recursion for free.
- Disadvantage: no pattern matching built in; predecessor is hard to define. (Parigot: predecessor cannot be defined in constant time on Church numerals.)
Scott numerals
(First mentioned in Curry-Feys 1958)

\[
\begin{align*}
0 & : = \lambda x \ f . x \\
1 & : = \lambda x \ f . f \ 0 \\
\quad 2 & : = \lambda x \ f . f \ 1 \\
\quad n + 1 & : = \lambda x \ f . f \ n \\
\quad \text{Succ} & : = \lambda p . \lambda x \ f . f \ p
\end{align*}
\]

- The Scott numerals have case distinction as a basis: the numerals are case distinguishors.
- **Case scheme** for Nat. (Let \( D \) be any type.)

\[
\begin{align*}
\text{Case} \ 0 \ : & \ : d : D \quad f : \text{Nat} \rightarrow D \\
\text{Case Successor} \ : & \ : \text{Case} \ d \ f \ (\text{Succ} \ x) \ : \ : f \ x
\end{align*}
\]

- **Advantage:** the predecessor can immediately be defined:
\[
P \ : = \lambda p . p \ 0 \ (\lambda y . y).
\]
- **Disadvantage:** No recursion (which one has to get from somewhere else, e.g. a fixed point-combinator).
Church-Scott numerals

Also called Parigot numerals (Parigot 1988, 1992).

\[
\begin{align*}
\bar{0} & := \lambda x f.x \\
\bar{1} & := \lambda x f.f x \\
\bar{2} & := \lambda x f.f (f x)
\end{align*}
\]

\[
\begin{align*}
0 & := \lambda x f.x \\
1 & := \lambda x f.f 0 \\
2 & := \lambda x f.f 1
\end{align*}
\]

For Church-Scott:

\[
\begin{align*}
n + 1 & := \lambda x f.f n(n x f) \\
\text{Succ} & := \lambda p.\lambda x f.f p(p x f)
\end{align*}
\]

Primitive recursion scheme for Nat. (Let D be any type.)

\[
\begin{align*}
d : D & \quad f : \text{Nat} \to D \to D \\
\Rightarrow & \quad \text{Rec } d f : \text{Nat} \to D
\end{align*}
\]

with

\[
\begin{align*}
\text{Rec } d f 0 & \to d \\
\text{Rec } d f (\text{Succ } x) & \to f x (\text{Rec } d f x)
\end{align*}
\]
Church-Scott numerals

Also called Parigot numerals (Parigot 1988, 1992).

<table>
<thead>
<tr>
<th>Church</th>
<th>Scott</th>
<th>Church-Scott</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bar{0}$ := $\lambda x f.x$</td>
<td>$0$ := $\lambda x f.x$</td>
<td>$0$ := $\lambda x f.x$</td>
</tr>
<tr>
<td>$\bar{1}$ := $\lambda x f.f x$</td>
<td>$1$ := $\lambda x f.f 0$</td>
<td>$1$ := $\lambda x f.f 0 x$</td>
</tr>
<tr>
<td>$\bar{2}$ := $\lambda x f.f (f x)$</td>
<td>$2$ := $\lambda x f.f 1$</td>
<td>$2$ := $\lambda x f.f 1(f 0 x)$</td>
</tr>
</tbody>
</table>

- **Advantage:** the predecessor can immediately be defined:
  \[ P := \lambda p.p\bar{0}(\lambda y.y). \]

- **Advantage:** quite a lot of recursion built in.

- **Disadvantage:** Data-representation of $n \in \mathbb{N}$ is exponential in $n$. (But: see recent work by Stump & Fu.)

- **Disadvantage:** No canonicity. There are closed terms of type $\text{Nat}$ that do not represent a number, e.g. $\lambda x f.f 2 x$.

NB For Church numerals we have canonicity:
If $\vdash t : \forall X.X \to (X \to X) \to X$, then $\exists n \in \mathbb{N}(t = \beta n)$. Similarly for Scott numerals.
Typing Church and Scott data types

- Church data types can be typed in polymorphic $\lambda$-calculus, $\lambda 2$. E.g. for Church numbers: $\text{Nat} := \forall X. X \rightarrow (X \rightarrow X) \rightarrow X$.

- To type Scott data types we need $\lambda 2\mu$: $\lambda 2 +$ positive recursive types:
  - $\mu X. \Phi$ is well-formed if $X$ occurs positively in $\Phi$.
  - Equality is generated from $\mu X. \Phi = \Phi[\mu X. \Phi / X]$.
  - Additional derivation rule:
    \[
    \frac{\Gamma \vdash M : A \quad A = B}{\Gamma \vdash M : B}
    \]

For Scott numerals: $\text{Nat} := \mu Y. \forall X. X \rightarrow (Y \rightarrow X) \rightarrow X$, i.e.
\[
\text{Nat} = \forall X. X \rightarrow (\text{Nat} \rightarrow X) \rightarrow X.
\]

- Similarly for Church-Scott numerals:
  $\text{Nat} := \mu Y. \forall X. X \rightarrow (Y \rightarrow X \rightarrow X) \rightarrow X$, 
  \[
  \text{Nat} = \forall X. X \rightarrow (\text{Nat} \rightarrow X \rightarrow X) \rightarrow X.
  \]
Dually: coinductive types

Our pet example is $\text{Str}_A$, streams over $A$. Its (standard) definition in $\lambda2$ as a “Church data type” is

$$\text{Str}_A := \exists X. X \times (X \to A \times X)$$

$$\text{hd} := \lambda s. (s_2 s_1)_1$$

$$\text{tl} := \lambda s. \langle (s_2 s_1)_2, s_2 \rangle$$

NB1: I do typing à la Curry, so $\exists$-elim/$\exists$-intro are done ‘silently’.

NB2: $\langle - , - \rangle$ denotes pairing and $( - )_i$ denotes projection.

Two examples

$$\text{ones} := \langle 1, \lambda x. \langle 1, x \rangle \rangle : \text{Str}_\text{Nat}$$

$$\text{nats} := \langle 0, \lambda x. \langle x, \text{Succ} x \rangle \rangle : \text{Str}_\text{Nat}$$

NB Representations of streams in $\lambda$-calculus are finite terms in normal form.
Constructor for streams?

Church data type \( \text{Str}_A \)

\[
\text{Str}_A := \exists X. X \times (X \rightarrow A \times X)
\]

\[
hd := \lambda s. (s_2 \ s_1)_1
\]

\[
tl := \lambda s. \langle (s_2 \ s_1)_2, s_2 \rangle
\]

**Problem:** we cannot define

\[
\text{Cons} : A \rightarrow \text{Str}_A \rightarrow \text{Str}_A.
\]

Problem arises because \( \text{Str}_A \) is only a **weakly final co-algebra**. (No uniqueness in the diagram.)

We need a **co-algebra with co-case** in the syntax or a **primitive co-recursive co-algebra**
Coinductive types are final co-algebra’s

Final $F$-coalgebra: $(\nu F, \text{out})$ s.t. $\forall (B, g)$, $\exists! h$ such that the diagram commutes:

$$
\begin{array}{ccc}
B & \xrightarrow{g} & FB \\
\downarrow \hspace{2cm} \downarrow \hspace{2cm} \downarrow \hspace{2cm} \downarrow \\
\nu F & \xrightarrow{\text{out}} & F(\nu F) \\
\end{array}
$$

For streams over $A$, $F X = A \times X$.

$$
\begin{array}{ccc}
B & \xrightarrow{\langle g_1, g_2 \rangle} & A \times B \\
\downarrow \hspace{2cm} \downarrow \hspace{2cm} \downarrow \\
\text{Str}_A & \xrightarrow{\langle \text{hd}, \text{tl} \rangle} & A \times \text{Str}_A \\
\end{array}
$$

\[
\begin{align*}
\text{hd}(h b) & = g_1 b \\
\text{tl}(h b) & = h(g_2 b)
\end{align*}
\]
Co-iterative, prim. co-recursive, co-algebras with co-case

- A co-iterative $T$-co-algebra (also weakly final $T$-co-algebra) is a triple $(A, \text{out}, \text{CoIt})$
- A $T$-co-algebra with co-case is a triple $(A, \text{out}, \text{CoCase})$
- A primitive co-recursive $T$-co-algebra is a triple $(A, \text{out}, \text{CoRec})$

\[ g : B \rightarrow T(B) \]
\[ \text{CoIt} g : A \rightarrow T(A) \]
\[ g : T(\text{CoIt} g) \rightarrow T(A + B) \]
\[ g : \text{CoRec} g : A \rightarrow T(A) \]
For Streams over $A$ this amounts to the following

Streams over $A$ with $\text{CoCase}$ and Streams over $A$ with $\text{CoRec}$

\[ \begin{align*}
\text{CoCase } g : \quad & B \xrightarrow{g} A \times \text{Str}_A \\
\text{Str}_A \quad & \xrightarrow{\langle \text{hd}, \text{tl} \rangle} A \times \text{Str}_A
\end{align*} \]

\[ \begin{align*}
\text{CoRec } g : \quad & B \xrightarrow{g} A \times (\text{Str}_A + B) \\
\text{Str}_A \quad & \xrightarrow{\langle \text{hd}, \text{tl} \rangle} A \times \text{Str}_A
\end{align*} \]

- $\text{CoCase}$ with $B : = A \times \text{Str}_A$ and $g : = \text{Id}$ gives the constructor for streams:

  \[ \text{CoCase } \text{Id} : A \times \text{Str}_A \rightarrow \text{Str}_A \]

- $\text{CoRec}$ with $B : = A \times \text{Str}_A$ and $g : = \text{Id} \times \text{inl}$ gives the constructor for streams:

  \[ \text{CoRec } (\text{Id} \times \text{inl}) : A \times \text{Str}_A \rightarrow \text{Str}_A \]
Streams à la Scott and à la Church-Scott

Streams as a Church data type (in $\lambda 2$):

$$\text{Str}_A := \exists X. X \times (X \to A \times X)$$

Streams as a Scott data type (in $\lambda 2\mu$):

$$\text{Str}_A = \exists X. X \times (X \to A \times \text{Str}_A)$$

$$\text{hd} := \lambda s. (s_2 s_1)_1$$

$$\text{tl} := \lambda s. (s_2 s_1)_2$$

$$\text{Cons} := \lambda a s. \langle a, \lambda x. \langle a, s \rangle \rangle \quad \text{[take } X := A\text{]}$$

Streams as a Church-Scott data type (in $\lambda 2\mu$):

$$\text{Str}_A = \exists X. X \times (X \to A \times (\text{Str}_A + X))$$

$$\text{hd} := \lambda s. (s_2 s_1)_1$$

$$\text{tl} := \lambda s. \text{case } (s_2 s_1)_2 \text{ of } (\text{inl } y \Rightarrow y) \ (\text{inr } x \Rightarrow \langle x, s_2 \rangle)$$

$$\text{Cons} := \lambda a s. \langle a, \lambda x. \langle a, \text{inl } s \rangle \rangle \quad \text{[take } X := A\text{]}$$
We immediately check that

\[
\begin{align*}
\text{hd}(\text{Cons } a \ s) & \rightarrow a \\
\text{tl}(\text{Cons } a \ s) & \rightarrow s
\end{align*}
\]

Remark: Other definitions of Cons are possible, e.g.

\[
\text{Cons} := \lambda a \ s. \langle \langle a, s \rangle, \lambda v. \langle v_1, \text{inl} \ v_2 \rangle \rangle \\
\text{[take } X := A \times \text{Str}_A]\]
The general pattern (inductive types)

Let \( \Phi(X) \) be a positive type scheme, i.e. \( X \) occurs only positively in the type expression \( \Phi(X) \).

- We view \( \Phi(X) \) as a functor on types. Positivity guarantees that \( \Phi \) acts functorially on terms: we can define \( \Phi(f) \) satisfying

\[
\begin{align*}
f : A &\rightarrow B \\
\Phi(f) &: \Phi(A) \rightarrow \Phi(B)
\end{align*}
\]

- We can define an iterative \( \Phi \)-algebra, a \( \Phi \)-algebra with case and a primitive recursive \( \Phi \)-algebra in the type theory as follows:

  - Church data type (iterative), in \( \lambda2 \)
  \[
  A := \forall X.(\Phi(X) \rightarrow X) \rightarrow X
  \]

  - Scott data type (case), in \( \lambda2\mu \)
  \[
  A = \forall X.(\Phi(A) \rightarrow X) \rightarrow X
  \]

  - Church-Scott data type (primitive recursive), in \( \lambda2\mu \)
  \[
  A = \forall X.(\Phi(A \times X) \rightarrow X) \rightarrow X
  \]
The general pattern (coinductive types)

Let again $\Phi(X)$ be a positive type scheme.

We can define an co-iterative $\Phi$-co-algebra, a $\Phi$-co-algebra with co-case and a primitive co-recursive $\Phi$-co-algebra in the type theory as follows:

- **Church data type (co-iterative), in $\lambda 2$**

  $$A := \exists X. X \times (X \to \Phi(X))$$

- **Scott data type (co-case), in $\lambda 2\mu$**

  $$A := \exists X. X \times (X \to \Phi(A))$$

- **Church-Scott data type (primitive co-recursive), in $\lambda 2\mu$**

  $$A := \exists X. X \times (X \to \Phi(A + X))$$
Definition of streams in Coq

In the Coq system, CoInductive types are defined using constructors and not using destructors. Question: Can we reconcile this?

CoInductive Stream (T: Type): Type :=
  Cons: T -> Stream T -> Stream T.

The destructors are defined by pattern matching.
How to define

    ones = 1 :: ones

with ones : Stream nat

CoFixpoint ones : Stream nat :=
  Cons 1 ones.

The recursive call to ones is guarded by the constructor Cons.
NB. The term ones does not reduce to Cons 1 ones.
Zipping and streams as sequences

The following definition is accepted by Coq

CoFixpoint zip (s t : Stream A) :=
  Cons (hd s) (zip t (tl s)).

There is an isomorphism between Stream A and nat -> A.

CoFixpoint F (f:nat->A) : Stream A :=
  Cons (f 0)(F (fun n:nat => f (S n))).

This defines

\[ F(f) := f(0) :: F(\lambda n.f(n + 1)) \]

which is correct, because \( F \) is guarded by the constructor.
For a co-inductive type definition, Coq gives the following

- Cons : $F \nu F \rightarrow \nu F$
- out $\circ$ Cons = Id
  (For streams: hd(Cons a s) = a and tl(Cons a s) = s).
- $\forall x : \nu F, \exists y : F(\nu F), x = \text{Cons } y$
- A guarded definition principle

Can we recover these from the final algebra diagram?
Coq’s coinductive types from final coalgebras

\[
\begin{array}{c}
B \xrightarrow{g} FB \\
\downarrow \quad \downarrow \quad \downarrow \\
!h \quad F h \\
\nu F \xrightarrow{\text{out}} F(\nu F)
\end{array}
\]

- We define \( \text{Cons} := \text{CoIt}(F \text{out}) \) (the \( h \) we get if we take \( g := F \text{out} \).
- Then \( \text{out} \circ \text{Cons} = \text{Id} \) (By Lambek’s Lemma)
- From this one can prove

\[
\text{Cons} \circ \text{out} = \text{Id}
\]

- Then for \( \forall x : A, \exists y : FA, x = \text{Cons} y \), by taking \( y := \text{out} \, x \).
Deriving Coq’s guarded definitions from final coalgebras for $\text{Str}_A$

The left of the diagram can be further decomposed.
Coq’s guarded definitions from final coalgebras

Coq actually uses this property to define the function $h$.

```coq
CoFixpoint h (x:A) := Cons(g1 x) (h (g2 x))
```
Following Krivine, Parigot, Leivant we can use proof terms in second order logic (AF2) as programs. This also works for recursively defined data types. 

Assume some ambient domain \( U \), with a constant \( Z \) and a unary function \( S \).

The natural numbers defined as a predicate on \( U \):

\[
\text{Nat}(x) := \forall X. X(Z) \rightarrow (\forall y. \text{Nat}(y) \rightarrow X(y) \rightarrow X(S\ y)) \rightarrow X(x)
\]

When we erase all first order parts, we get the Church-Scott natural numbers:

\[
\text{Nat} := \forall X. X \rightarrow (\text{Nat} \rightarrow X \rightarrow X) \rightarrow X
\]
Programming with proofs

The method now defines the untyped $\lambda$-terms 0 and Succ as the proof-terms

$$0 : \text{Nat}(Z)$$
$$\text{Succ} : \forall x. \text{Nat}(x) \rightarrow \text{Nat}(Sx)$$

Then

$$0 =_{\beta} \lambda z . f . z$$
$$\text{Succ} =_{\beta} \lambda p . \lambda z . f . f . p . (p . z . f)$$

All the proofs of $\text{Nat}(t)$ are representations of numbers; there is no ‘junk’
Recursive programming with proofs

Nat(x) := ∀X. X(Z) → (∀y. Nat(y) → X(y) → X(S y) → X(x)

Programming can now be done by adding a function symbol with an equational specification, e.g.

\[ A(Z, y) = y \]
\[ A(S(x), y) = S(A(x, y)) \]

Then give a proof term

\[ \text{Add} : \forall x, y. \text{Nat}(x) \rightarrow \text{Nat}(y) \rightarrow \text{Nat}(A(x, y)) \]

The proof-term Add is an implementation of addition in untyped \( \lambda \)-calculus.
Corecursive programming with proofs

Given a data type \( A \), and unary functions \( \text{hd} \) and \( \text{tl} \), we define streams over \( A \) by

\[
\text{Str}_A(x) := \exists X. X(x) \times (\forall y. X(y) \rightarrow A(\text{hd} y) \times X(\text{tl} y))
\]

We find that for our familiar functions \( \text{hd} \) and \( \text{tl} \):

\[
\text{hd} := \lambda s. (s_2 s_1)_1 : \forall x. \text{Str}_A(x) \rightarrow A(\text{hd} x)
\]

\[
\text{tl} := \lambda s. \langle (s_2 s_1)_2, s_2 \rangle : \forall x. \text{Str}_A(x) \rightarrow \text{Str}_A(\text{tl} x)
\]

Adding equations for \( \text{ones} \):

\[
\text{hd}(\text{ones}) = 1
\]

\[
\text{tl}(\text{ones}) = \text{ones}
\]

we can give a proof term

\[
\text{ones} : \text{Str}_{\text{Nat}}(\text{ones})
\]

for example by taking

\[
\text{ones} := \langle \text{Id}, \lambda x. \langle 1, \text{Id} \rangle \rangle : \text{Str}_{\text{Nat}}
\]
Correctness of corecursive programming with proofs

The proof term ones is guaranteed to be correct:

\[
\begin{align*}
\text{hd}(\text{ones}) & \rightarrow 1 \\
\text{tl}(\text{ones}) & \rightarrow \text{ones}
\end{align*}
\]
Corecursive programming with proofs

To define \text{Cons}, we need to make \text{Str}_A into a recursive type:

\[
\text{Str}_A(x) := \exists X. X(x) \times (\forall y. X(y) \rightarrow A(\text{hd} \ y) \times (\text{Str}_A(\text{tl} \ y) + X(\text{tl} \ y)))
\]

Adding equations for \text{Cons}:

\[
\begin{align*}
\text{hd}(\text{Cons} \ x \ y) & = x \\
\text{tl}(\text{Cons} \ x \ y) & = y
\end{align*}
\]

We see that with

\[
\text{Cons} := \lambda a \ s. \langle a, \langle a, \lambda v. \langle v_1, \text{inl} \ v_2 \rangle \rangle \rangle
\]

\[\text{take } X(x) := A(\text{hd} \ x) \times \text{Str}_A(\text{tl} \ x)\].

we have

\[
\text{Cons} : \forall x, y, A(x) \rightarrow \text{Str}_A(y) \rightarrow \text{Str}_A(\text{Cons} \times y)
\]
The typing system

To avoid syntactic overload and to get untyped $\lambda$ terms, we use Curry style typing (as in AF2)

$p : A \in \Gamma \quad \Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A \quad \Gamma, p : A \vdash M : B$

$\Gamma \vdash p : A \quad \Gamma \vdash M N : A \quad \Gamma \vdash \lambda p. M : A \rightarrow B$

$\Gamma \vdash M : A \quad X \notin FV(\Gamma) \quad \Gamma \vdash M : \forall X. A$

$\Gamma \vdash M : \forall X. A \quad \Gamma \vdash M : A[B(\bar{x})/X]$

$\Gamma \vdash M : A \quad x \notin FV(\Gamma) \quad \Gamma \vdash M : \forall x. A$

$\Gamma \vdash M : \forall x. A \quad \Gamma \vdash M : A[t/x]$

This works all very well for the inductive data types case
Problem

For the coinductive case, we have to deal with \( \exists \). Curry-style exists rules are:

\[
\frac{\Gamma \vdash M : A[B(\vec{x})/X]}{\Gamma \vdash M : \exists X.A}
\]

\[
\frac{\Gamma \vdash M : \exists X.A \quad \Gamma, p : A \vdash N : B}{\Gamma \vdash N[M/p] : B \quad \text{if } X \notin \text{FV}(\Gamma, B)}
\]

Problem: This system does not satisfy the not Subject Reduction property! (See counterexample in Sørensen-Urzyczyn)

The rules should be:

\[
\frac{\Gamma \vdash M : A[B(\vec{x})/X]}{\Gamma \vdash \lambda h.h M : \exists X.A}
\]

\[
\frac{\Gamma \vdash M : \exists X.A \quad \Gamma, p : A \vdash N : B}{\Gamma \vdash M(\lambda p.N) : B \quad \text{if } X \notin \text{FV}(\Gamma, B)}
\]
Conclusion/Questions

▶ Church-Scott data types provide a good union of the two,
  ▶ giving (co)-recursion in untyped $\lambda$-calculus
  ▶ being typable in $\lambda2\mu$
  ▶ but: the size of representation is a problem. (Recent work by Stump and Fu)

▶ We can prevent closed terms that don’t represent data, by moving to types in AF2

Some questions:

▶ Can the “programming with proofs” approach in AF2 for inductive types fully generalize to coinductive types? Using Curry-style typing?
▶ Does that include corecursive types?
▶ Can we reconcile with the “naive” looping intuition?
Related Work on (co)inductive types in non-dependent type theories, lots

- Mendler style inductive/coinductive types: Mendler, Matthes, Uustalu, Vene
- Extending to course-of-value recursion: Matthes, Uustalu, Vene
- Impossibility results: Parigot, Malaria, Splawski & Urzyczyn
- General recursion via coinductive types: Capretta
- Recursive Coalgebras and Corecursive Algebras: Osius; Capretta & Uustalu & Vene
Related Work on coinductive types in dependent type theories

- Coquand, Gimenez
- Copatterns by Abel, Pientka, Setzer
- Type theory based solely on inductive and coinductive types: Basold, H.G.
Related Work on programs from proofs, lots

- Mendler style inductive/coinductive types: Miranda-Perea & González-Huesca
- Christophe Raffalli: infinitary terms
- Tatsuta: first order logic with (co)-inductive definitions
- Leivant
Related Work on Scott numerals/data

- “Types for Scott Numerals” Abadi, Cardelli, Plotkin 1993
- Brunel & Terui: capture polynomial time functions using Scott data types and linear types.
- Similar use in Baillot & De Benedetti & Ronchi della Rocca
- Scott data types with call-by-value and call-by-name iteration (H.G.)
Questions?