The Church-Scott representation of inductive and coinductive data in (typed) λ calculus

Herman Geuvers

Radboud University Nijmegen
and
Eindhoven University of Technology
The Netherlands

Types 2014, Paris
March 15, 2014

Church numerals

The most well-known Church data type

$$\begin{array}{rcl} \overline{0} & := & \lambda x\, f.x \\ \overline{1} & := & \lambda x\, f.f\, x \\ \overline{2} & := & \lambda x\, f.f\,(f\, x) \end{array} \qquad \begin{array}{rcl} \overline{p} & := & \lambda x\, f.f^p\,(x) \\ \overline{\mathrm{Succ}} & := & \lambda n.\lambda x\, f.f\,(n\, x\, f) \end{array}$$

- The Church data types have iteration as basis. The numerals are iterators.
- Iteration scheme for nat. (Let D be any type.)

$$\frac{d : D \quad f : D \to D}{\mathrm{It}\, d\, f : \mathrm{nat} \to D} \quad \text{with} \quad \begin{array}{lcl} \mathrm{It}\, d\, f\, \overline{0} & \twoheadrightarrow & d \\ \mathrm{It}\, d\, f\, (\overline{\mathrm{Succ}}\, x) & \twoheadrightarrow & f\, (\mathrm{It}\, d\, f\, x) \end{array}$$

- Advantage: quite a bit of well-founded recursion for free.
- Disadvantage: no pattern matching built in; predecessor is hard to define.

Scott numerals

(First mentioned in Curry-Feys 1958)

$$\begin{array}{lll} \underline{0} & := & \lambda x\, f.x \\ \underline{1} & := & \lambda x\, f.f\,\underline{0} \\ \underline{2} & := & \lambda x\, f.f\,\underline{1} \end{array} \qquad \begin{array}{lll} \underline{n+1} & := & \lambda x\, f.f\,\underline{n} \\ \underline{\mathrm{Succ}} & := & \lambda p.\lambda x\, f.f\,p \end{array}$$

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

$$\frac{d : D \quad f : \mathrm{nat} \to D}{\mathrm{Case}\, d\, f : \mathrm{nat} \to D} \quad \text{with} \quad \begin{array}{lll} \mathrm{Case}\, d\, f\, \underline{0} & \twoheadrightarrow & d \\ \mathrm{Case}\, d\, f\, (\underline{\mathrm{Succ}}\, x) & \twoheadrightarrow & f\, x \end{array}$$

- Advantage: the predecessor can immediately be defined: $P := \lambda p.p\,\underline{0}\,(\lambda y.y)$.
- Disadvantage: No recursion (which one has to get from somewhere else, e.g. a fixed point-combinator).

Primitive recursion scheme

Can we define numerals such that we have the following definition scheme?

Primitive Recursion scheme for nat. (Let D be any type.)

$$\frac{d : D \quad f : \mathrm{nat} \to D \to D}{\mathrm{Rec}\, d\, f : \mathrm{nat} \to D} \qquad \begin{array}{lcl} \mathrm{Rec}\, d\, f\, 0 & \twoheadrightarrow & d \\ \mathrm{Rec}\, d\, f\, (\mathrm{Succ}\, x) & \twoheadrightarrow & f\, x\, (\mathrm{Rec}\, d\, f\, x) \end{array}$$

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

$$\cfrac{\cfrac{\cfrac{d : D \qquad\qquad f : \mathrm{nat} \to D \to D}{\langle 0, d\rangle : \mathrm{nat} \times D \qquad \lambda z.\langle z_1, f\, z_1\, z_2\rangle : \mathrm{nat} \times D \to \mathrm{nat} \times D}}{\mathrm{It}\,\langle 0, d\rangle\, \lambda z.\langle z_1, f\, z_1\, z_2\rangle : \mathrm{nat} \to \mathrm{nat} \times D}}{\lambda p.(\mathrm{It}\,\langle 0, d\rangle\, \lambda z.\langle z_1, f\, z_1\, z_2\rangle\, p)_2 : \mathrm{nat} \to D}$$

$\langle -, -\rangle$ 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:

$$\begin{array}{rcl} P(\mathrm{Succ}^{n+1}\,0) & \twoheadrightarrow & \mathrm{Succ}^{n}\,0 \\ \text{but not} \qquad P(\mathrm{Succ}\,x) & = & x \end{array}$$

- Computationally inefficient

$$P(\mathrm{Succ}^{n+1}\,0) \twoheadrightarrow \mathrm{Succ}^{n}\,0 \text{ in linear time}$$

Typing Church and Scott data types

- Church data types can be typed in polymorphic λ-calculus, $\lambda 2$. E.g. for Church numbers: $\mathrm{nat} := \forall X.X \to (X \to X) \to X$.
- 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 Φ.
  - Equality on types is the congruence generated from $\mu X.\Phi = \Phi[\mu X.\Phi/X]$.
  - Additional derivation rule:

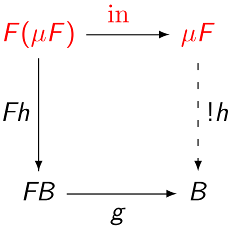
$$\frac{\Gamma \vdash M : A \qquad A = B}{\Gamma \vdash M : B}$$

E.g. for Scott numerals: $\mathrm{nat} := \mu Y.\forall X.X \to (Y \to X) \to X$, that is

$$\mathrm{nat} = \forall X.X \to (\mathrm{nat} \to X) \to X.$$

The categorical picture

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

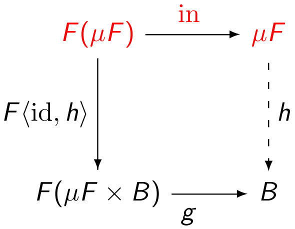


Due to the uniqueness:

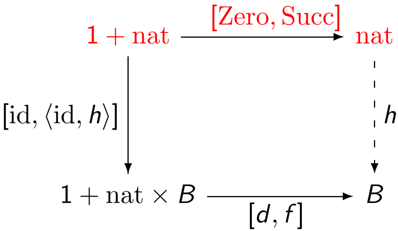
- in is an isomorphism, so it has an inverse $\mathrm{out} : \mu F \to F(\mu F)$. (In case $FX := 1 + X$, $\mu F = \mathrm{nat}$ and out is basically the predecessor.)
- we can derive the prim. rec. scheme via this diagram.
- But in syntax we only have weakly initial algebras: $\exists$, but not $\exists !$. So we get out and prim.rec. only in a weak slightly twisted form . . .

Recursive Algebras

We want something stronger than weakly initial ...
[G. 1992]: Recursive F-algebra: $(\mu F, \mathrm{in})$ s.t. $\forall (B, g)$, $\exists h$ such that



For nat this is: $\forall B, \forall d : 1 \to B, \forall f : \mathrm{nat} \times B \to B$, $\exists h$ such that



That is: $h = \mathrm{Rec}\, d\, f$.

Recursive algebras in type theory

- In [G. 1992] I added to $\lambda 2$
  1. inductive types with
  2. constructor in,
  3. eliminator Rec and
  4. reduction rules representing the commuting diagram. (Similarly for coinductive types.)
- But we can do better:
  - We can merge the Church and Scott approach and have recursive algebras already in untyped λ-calculus.
  - These can be typed in $\lambda 2\mu$.
  - We can do this dually for coinductive types.

Church-Scott numerals

Also called Parigot numerals (Parigot 1988, 1992).

		Church			Scott			Church-Scott
$\overline{0}$	$:=$	$\lambda x\, f.x$	$\underline{0}$	$:=$	$\lambda x\, f.x$	0	$:=$	$\lambda x\, f.x$
$\overline{1}$	$:=$	$\lambda x\, f.f\, x$	$\underline{1}$	$:=$	$\lambda x\, f.f\, \underline{0}$	1	$:=$	$\lambda x\, f.f\, 0\, x$
$\overline{2}$	$:=$	$\lambda x\, f.f\, (f\, x)$	$\underline{2}$	$:=$	$\lambda x\, f.f\, \underline{1}$	2	$:=$	$\lambda x\, f.f\, 1\, (f\, 0\, x)$

For Church-Scott:

$$\begin{aligned} n+1 &:= \lambda x\, f.f\; n\, (n\, x\, f) \\ \mathrm{Succ} &:= \lambda p.\lambda x\, f.f\; p\, (p\, x\, f) \end{aligned}$$

- These can be typed in $\lambda 2\mu$ as

$$\mathrm{nat} = \forall X.X \to (\mathrm{nat} \to X \to X) \to X.$$

- This is a recursive algebra
- NB This works very generally for all algebraic data types.

Church-Scott numerals

$$\begin{aligned} \mathrm{nat} &= \forall X.X \to (\mathrm{nat} \to X \to X) \to X \\ \mathrm{Succ} &:= \lambda p.\lambda x\, f.f\, p\,(p\, x\, f) \end{aligned}$$

Positive: we have Rec directly

$$\frac{d : D \quad f : \mathrm{nat} \to D \to D}{\mathrm{Rec}\, d\, f := \lambda n : \mathrm{nat}.n\, d\, f \,:\, \mathrm{nat} \to D}$$

$$\begin{aligned} \mathrm{Rec}\, d\, f\, 0 &\twoheadrightarrow d \\ \mathrm{Rec}\, d\, f\, (\mathrm{Succ}\, x) &=_\beta f\, x\, (\mathrm{Rec}\, d\, f\, x) \end{aligned}$$

Church-Scott numerals

$$\begin{array}{rcl} \mathrm{nat} & = & \forall X.X \to (\mathrm{nat} \to X \to X) \to X \\ \mathrm{Succ} & := & \lambda p.\lambda x\, f.f\, p\,(p\, x\, f) \end{array}$$

Negative:

- Representation of n is exponential in the size of n.
- No canonicity: There are closed terms of type 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 $\exists n \in \mathbf{N}(t =_\beta \overline{n})$.
  Similarly for Scott numerals.

Dually: coinductive types

Our pet example is Str_A, streams over A. Its (standard) definition in $\lambda 2$ as a "Church datatype" is

$$\begin{array}{rcl} \mathrm{Str}_A & := & \exists X.X \times (X \to A \times X) \\ \mathrm{hd} & := & \lambda s.(s_2\, s_1)_1 \\ \mathrm{tl} & := & \lambda s.\langle (s_2\, s_1)_2, s_2 \rangle \end{array}$$

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

$$\begin{array}{rcl} \mathrm{ones} & := & \langle 1, \lambda x.\langle 1, x \rangle \rangle : \mathrm{Str}_{\mathrm{nat}} \\ \mathrm{nats} & := & \langle 0, \lambda x.\langle x, \mathrm{Succ}\, x \rangle \rangle : \mathrm{Str}_{\mathrm{nat}} \end{array}$$

NB Representations of streams in λ-calculus are finite terms in normal form!

Constructor for streams?

Church datatype Str_A

$$\begin{array}{rcl} \mathrm{Str}_A & := & \exists X.X \times (X \to A \times X) \\ \mathrm{hd} & := & \lambda s.(s_2\, s_1)_1 \\ \mathrm{tl} & := & \lambda s.\langle (s_2\, s_1)_2, s_2 \rangle \end{array}$$

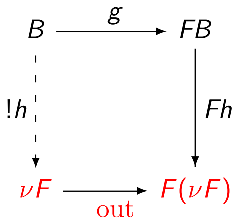
Problem: we cannot define

$$\mathrm{cons} : A \to \mathrm{Str}_A \to \mathrm{Str}_A.$$

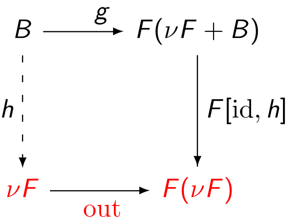
Problem arises because Str_A is only a weakly terminal co-algebra.
(No uniqueness in the diagram.)
We need a co-recursive co-algebra in the syntax.

Co-recursive co-algebra

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



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



Streams as Church-Scott data type

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

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

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

$$\begin{aligned}
\mathrm{Str}_A \quad &= \quad \exists X.X \times (X \to A \times (Str_A + X)) \\
\mathrm{hd} \quad &:= \quad \lambda s.(s_2\, s_1)_1 \\
\mathrm{tl} \quad &:= \quad \lambda s.\mathrm{case}\,(s_2\, s_1)_2 \text{ of } (\mathrm{inl}\, y \Rightarrow y)\,(\mathrm{inr}\, x \Rightarrow \langle x, s_2\rangle) \\
\mathrm{cons} \quad &:= \quad \lambda a\, s.\langle a, \lambda x.\langle a, \mathrm{inl}\, s\rangle\rangle \qquad\qquad [\text{take } X := A]
\end{aligned}$$

And we can check that

$$\begin{aligned}
\mathrm{hd}(\mathrm{cons}\, a\, s) \quad &:= \quad a \\
\mathrm{tl}(\mathrm{cons}\, a\, s) \quad &:= \quad s
\end{aligned}$$

Other definitions of cons are possible, e.g.

$$\mathrm{cons} := \lambda a\, s.\langle\langle a, s\rangle, \lambda v.\langle v_1, \mathrm{inl}\, v_2\rangle\rangle \qquad\qquad [\text{take } X := A \times \mathrm{Str}_A]$$

Programming with proofs

Following Krivine, Parigot, Leivant we can use proof terms in second order logic (SOL) as programs. This also works for recursively defined types in SOL. The natural numbers example:

$$\mathrm{nat}(x) := \forall X.X(\mathcal{Z}) \to (\forall y.\mathrm{nat}(y) \to X(y) \to X(\mathcal{S}\, y)) \to X(x)$$

where $\mathcal{Z}$ and $\mathcal{S}$ are a constant and a unary function in some ambient domain U.
Now define the untyped λ-terms 0 and Succ as the proof-terms

$$\begin{array}{rcl} 0 & : & \mathrm{nat}(\mathcal{Z}) \\ \mathrm{Succ} & : & \forall x.\mathrm{nat}(x) \to \mathrm{nat}(\mathcal{S}\, x) \end{array}$$

Then

$$\begin{array}{rcl} 0 & =_\beta & \lambda z\, f.z \\ \mathrm{Succ} & =_\beta & \lambda p.\lambda z\, f.f\; p\,(p\, z\, f) \end{array}$$

Recursive programming with proofs

$$\mathrm{nat}(x) := \forall X.X(\mathcal{Z}) \to (\forall y.\mathrm{nat}(y) \to X(y) \to X(\mathcal{S}\,y) \to X(x)$$

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

$$\begin{array}{rcl} \mathcal{A}(\mathcal{Z}, y) & = & y \\ \mathcal{A}(\mathcal{S}(x), y) & = & \mathcal{S}(\mathcal{A}(x, y)) \end{array}$$

And then prove

$$\forall x, y.\mathrm{nat}(x) \to \mathrm{nat}(y) \to \mathrm{nat}(\mathcal{A}(x, y))$$

This proof (the proof-term) is an implementation of addition in untyped λ-calculus.

Corecursive programming with proofs

Given a data type A, and unary functions $\mathcal{H}$ and $\mathcal{T}$, we define streams over A by

$$\mathrm{Str}_A(x) := \exists X.X(x) \times (\forall y.X(y) \to A(\mathcal{H}\,y) \times X(\mathcal{T}\,y))$$

We find that for our familiar functions hd and tl:

$$\begin{aligned} \mathrm{hd} := \lambda s.(s_2\,s_1)_1 \quad &: \quad \forall x.\mathrm{Str}_A(x) \to A(\mathcal{H}\,x) \\ \mathrm{tl} := \lambda s.\langle (s_2\,s_1)_2, s_2\rangle \quad &: \quad \forall x.\mathrm{Str}_A(x) \to \mathrm{Str}_A(\mathcal{T}\,x) \end{aligned}$$

To define cons, we need to make this into a recursive type:

$$\mathrm{Str}_A(x) := \exists X.X(x) \times (\forall y.X(y) \to A(\mathcal{H}\,y) \times (\mathrm{Str}_A(\mathcal{T}\,y) + X(\mathcal{T}\,y)))$$

and we see that

$$\mathrm{cons} := \lambda a\,s.\langle\langle a, s\rangle, \lambda v.\langle v_1, \mathrm{inl}\,v_2\rangle\rangle$$

is a well-typed constructor function
[take $X(x) := A(\mathcal{H}\,x) \times \mathrm{Str}_A(\mathcal{T}\,x)$].

Conclusion

- Church-Scott data types provide a good union of the two,
  - giving (co)-recursion in untyped λ-calculus
  - being typable in $\lambda 2\mu$
  - but the size of representation is a problem.
- We can prevent closed terms that don't represent data, by moving to types in SOL

Some questions:

- Does the "programming with proofs" approach in SOL for inductive types fully generalize to coinductive types?
- Does that include corecursive types?