The Church-Scott representation of inductive and coinductive data in typed lambda calculus

Herman Geuvers[1,2]

[1] ICIS, Radboud University Nijmegen, the Netherlands
[2] Technical University Eindhoven, the Netherlands

Data in the lambda calculus is usually represented using the "Church encoding", which gives closed terms for the constructors and which naturally allows to define functions by "iteration". An additional nice feature is that in system F (polymorphically typed lambda calculus) one can define closed data types for this data, the iteration scheme is well-typed and beta-reduction is always terminating. A problem is that primitive recursion is not directly available: it can be coded in terms of iteration at the cost of inefficiency (e.g. a predecessor with linear run-time). The much less well-known Scott encoding [1] has case distinction as a primitive. (For the numerals, these are also known as 'Parigot numerals'[3,4].) The terms are not typable in system F and there is no iteration scheme, but there is a constant time destructor (e.g. predecessor).

We will present a unification of the Church and Scott presentation of data types, which has primitive recursion as basic. We show how these can be typed in the polymorphic lambda calculus extended with recursive types and we show that all terms are strongly normalizing. We also show that this works for the dual case, co-inductive data types, and we show how programs can be extracted from proofs in second order predicate logic.

Church and Scott data types As a step back, we look at data in the untyped λ-calculus. Church numerals:

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

The Church numerals have *iteration* as basis: the numerals are iterators. An advantage is that one gets quite a bit of "well-founded recursion" for free. A disadvantage is that there is no pattern matching built in, so the predecessor is hard to define.

Scott numerals:

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

The Scott numerals have *case* as a basis: the numerals are *case distinctors*: $\underline{n}AB = A$ if $n = 0$ and $\underline{n}AB = B\underline{m}$ if $n = m + 1$. An advantage is that the predecessor can immediately be defined, but one has to get "recursion" from somewhere else (e.g. by using a fixed point-combinator).

A more general definition of Church and Scott data in the untyped λ-calculus is the following. Given a data type with constructors $\mathbf{c}_1, \ldots, \mathbf{c}_k$, each with a fixed arity, say the arity of constructor $\mathbf{c}_i$ is $\mathsf{ar}(i)$, we have a *Church encoding*:

$$\overline{\mathbf{c}_i} := \lambda x_1 \ldots x_{\mathsf{ar}(i)}.\lambda c_1 \ldots c_k.c_i\, (x_1\, \vec{c}) \ldots (x_{\mathsf{ar}(i)}\, \vec{c}) \qquad \begin{array}{lcl} \overline{0} & := & \lambda x\, f.x \\ \overline{S} & := & \lambda n.\lambda x\, f.f\, (n\, x\, f) \end{array}$$

and a *Scott encoding*:

$$\underline{\mathbf{c}_i} := \lambda x_1 \ldots x_{\mathsf{ar}(i)}.\lambda c_1 \ldots c_k.c_i\, x_1 \ldots x_{\mathsf{ar}(i)} \qquad \begin{array}{lcl} \underline{0} & := & \lambda x\, f.x \\ \underline{S} & := & \lambda n.\lambda x\, f.f\, n \end{array}$$

The Scott encoding is simpler, but it's not very well-known and seldom used. Why? Probably the main reason is that Church data can be typed in the polymorphic λ-calculus $\lambda 2$, and already quite a lot of functions can be typed in simple type theory, $\lambda\!\!\rightarrow$.

To type Church numerals of type $\mathtt{nat}$, we need $\mathtt{nat} = A \to (A \to A) \to A$. (Church 1940). In polymorphic λ-calculus, we can even do better by taking

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

There is a (well-known) general pattern behind this

$$\begin{array}{rcl} \mathtt{bool} & := & \forall X.X \to X \to X \\ \mathtt{list}_A & := & \forall X.X \to (A \to X \to X) \to X \\ \mathtt{bintree}_{A,b} & := & \forall X.(A \to X) \to (B \to X \to X \to X) \to X \end{array}$$

This provides a nice *function definition scheme* in $\lambda 2$. As an example we give the *iteration scheme* for $\mathtt{nat}$ and $\mathtt{list}_A$. (Let D be a type.)

$$\frac{d : D \quad f : D \to D}{\mathsf{It}\, d\, f : \mathtt{nat} \to D}$$

with
$\mathsf{It}\, d\, f\, \overline{0} \twoheadrightarrow d$
$\mathsf{It}\, d\, f\, (\overline{S}\, x) \twoheadrightarrow f\, (\mathsf{It}\, d\, f\, x)$

$$\frac{d : D \quad f : A \to D \to D}{\mathsf{It}\, d\, f : \mathtt{list}_A \to D}$$

with
$\mathsf{It}\, d\, f\, \overline{\mathtt{nil}} \twoheadrightarrow d$
$\mathsf{It}\, d\, f\, (\overline{\mathtt{cons}}\, a\, x) \twoheadrightarrow f\, a\, (\mathsf{It}\, d\, f\, x)$

Important features: (1) This scheme is available in $\lambda 2$: we can define It for $\mathtt{nat}$, $\mathtt{list}_A$, ...; (2) Using this we can define as λ-terms very many functions: $+$, $\times$, exp, Ackermann, ..., map-list, fold, ...; (3) Because these terms are typed in $\lambda 2$, these are all terminating.

Do we have types for Scott data? To type Scott numerals we need $\mathtt{nat} = A \to (\mathtt{nat} \to A) \to A$.

In $\lambda 2$, we cannot do this, unless we extend it with (positive) recursive types, obtaining $\lambda 2\mu$:

$$\mathtt{nat} := \mu Y.\forall X.X \to (Y \to X) \to X.$$

Type formation rule: $\mu Y.\Phi[Y]$ is a well-formed type if Y occurs positive in the type expression $\Phi[Y]$. A drawback of this approach is that one does not get any well-founded recursion "for free".

Combined Church-Scott encoding To get the best of both worlds, we can define the CS (Church-Scott) numerals by

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

These numerals can be types as well in $\lambda 2\mu$:

$$\mathtt{nat} := \mu Y.\forall X.X \to (Y \to X \to X) \to X.$$

The advantage is that now one obtains the primitive *recursion scheme* for free.

$$\frac{d : D \quad f : \mathtt{nat} \to D \to D}{\mathsf{Rec}\, d\, f : \mathtt{nat} \to D}$$

with $\mathsf{Rec}\, d\, f := \lambda n : \mathtt{nat}.n\, d\, f$, satisfying $\mathsf{Rec}\, d\, f\, \underline{\overline{0}} = d$ and $\mathsf{Rec}\, d\, f\, (\underline{\overline{S}}\, x) = f\, x\, (\mathsf{Rec}\, d\, f\, x)$.

For other known data types, we can do the same, if we observe that

- in $\lambda 2$, $\mathtt{nat} := \mathrm{lfp}\,\Phi$, with $\Phi(X) = 1 + X$ and $\mathrm{lfp}\,\Phi$ is the well-known definable (weak) least fixed point in $\lambda 2$,
- in our new definition $\mathtt{nat} := \mu Y.\mathrm{lfp}\,\Phi^{\times Y}$, where $\Phi^{\times Y}(X) = 1 + (Y \times X)$.

The fact that all this works is due to that fact that we can define *recursive algebras* (in the terminology of [2]) inside $\lambda 2\mu$ in a very generic way.

The dual case: streams It is well-know that streams over a base type A can also be defined in $\lambda 2$:

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

If we use the same type of implicit Curry-style typing for $\exists$ that we have also used for $\forall$ above, and we use $\langle -, - \rangle$ for pairing and $(-)_1$ and $(-)_2$ for the projections. we see that the definitions of head and tail are:

$$\begin{array}{lcl} \mathtt{hd}\, s & := & (s_2\, s_1)_1 \\ \mathtt{tl}\, s & := & \langle (s_2\, s_1)_2, s_2 \rangle \end{array}$$

Now it is hard to define the cons operator, that takes an $a : A$ and an $s : \mathtt{Str}_A$ to create $\mathtt{cons}\, a\, s : \mathtt{Str}_A$. However, in $\lambda 2\mu$ we can define a *co-recursive co-algebra* for the functor $X \mapsto A \times X$ as follows:

$$\mathtt{Str}_A := \mu Y.\exists X.X \times (X \to A \times (X + Y))$$

Now we can define

$$\begin{array}{rcl} \mathtt{hd}\, s & := & (s_2\, s_1)_1 \\ \mathtt{tl}\, s & := & \mathrm{case}\,(s_2\, s_1)_2\ \mathrm{of}\ (\mathtt{inl} x \Rightarrow \langle x, s_2 \rangle)\ (\mathtt{inr} y \Rightarrow y) \\ \mathtt{cons}\, a\, s & := & \langle a, \lambda x.\langle a, \mathtt{inr}\, s \rangle \rangle \end{array}$$

And we can check that

$$\begin{array}{lcl} \mathtt{hd}(\mathtt{cons}\, a\, s) & := & a \\ \mathtt{tl}(\mathtt{cons}\, a\, s) & := & s \end{array}$$

It can be shown that the approach sketched above works for more general inductive and co-inductive data-types. It gives "nice" finite representations of infinite data in the untyped lambda calculus. (E.g. the stream of natural numbers is a term in normal form.) It can also be shown that the programs from proof extraction mechanism as developed by Krivine, Leivant and Parigot [3] works nicely for these data type definitions.

Reference

[1] M. Abadi, L. Cardelli and G. Plotkin, Types for the Scott Numerals, 1993, `http://lucacardelli.name/Papers/Notes/scott2.pdf`.

[2] H. Geuvers, Inductive and Coinductive Types with Iteration and Recursion, in the informal proceedings of the 1992 workshop on Types for Proofs and Programs, Bastad 1992, Sweden, eds. B. Nordström, K. Petersson and G. Plotkin, pp 183–207. `http://www.cs.ru.nl/~herman/PUBS/BRABasInf_RecTyp.ps.gz`

[3] P. Fu, A. Stump, Self Types for Dependently Typed Lambda Encodings, Submitted to RTA-TLCA 2014.

[4] M. Parigot, Recursive Programming with Proofs, Theor. Comput. Sci., 94, 2, 1992, pp. 335-336.