Dependent Lambda Encoding with Self Types

Peng Fu, Aaron Stump
Computer Science, The University of Iowa

It is well known that natural numbers can be encoded as lambda terms using Church encoding [2] or Scott encoding (reported in [4]). So operations such as plus, multiplication can be performed by beta-reduction on lambda terms. Other inductive data structures such as trees, lists, etc. ([1], chapter 11 in [6]) can also be represented in a similar fashion.

Church-encoded data can be typed in system **F** [5]. But this approach is rarely adopted in dependent type systems. As summarized by Werner [8], it is inefficient to define certain operation on Church-encoded data, e.g. the predecessor function; the induction principle is not derivable and $0 \neq 1$ cannot be proved. Thus we are led to the consideration of extending the Calculus of Construction(CoC) [3] with inductive datatypes [7].

In CoC à la Curry, we define $\mathsf{Nat} := \forall X.(X \to X) \to X \to X$. One can obtain a notion of *indexed iterator* by defining $\mathsf{It} := \lambda x.\lambda f.\lambda a.xfa$ and $\mathsf{It} : \forall X.\Pi x : \mathsf{Nat}.(X \to X) \to X \to X$. Thus we have $\mathsf{It}\ \bar{n} =_\beta \lambda f.\lambda a.\bar{n}\ f\ a =_\beta \lambda f.\lambda a.\underbrace{f(f(f...(f}_{n} a)...))$.

An indexed iterator is nice, but one may want to know if we can obtain a finer version, namely, the induction principle Id such that:

$\mathsf{Id} : \forall P : \mathsf{Nat} \to *.\Pi x : \mathsf{Nat}.(\Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y))) \to P\ \bar{0} \to P\ x$

Let us try to construct such an Id. First observe the following beta equalities:

$\mathsf{Id}\ \bar{0} =_\beta \lambda f.\lambda a.a$

$\mathsf{Id}\ \bar{n} =_\beta \lambda f.\lambda a.\underbrace{f\ \overline{n-1}(...f\ \bar{1}\ (f}_{n>0}\ \bar{0}\ a))$.

with $f : \Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y)), a : P\ \bar{0}$.

So the above equalities suggest $\mathsf{Id} := \lambda x.\lambda f.\lambda a.x\ f\ a$, with a different notion of lambda numerals, i.e.

$\bar{0} := \lambda s.\lambda z.z$

$\bar{n} := \lambda s.\lambda z.s\ \overline{n-1}\ (\overline{n-1}\ s\ z)$.

Now let us try to type these lambda numerals. It is reasonable to assign $s : \Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y))$ and $z : P\ \bar{0}$. Thus we have the following typing relation:

$\bar{0} : \Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ \bar{0}$

$\bar{1} : \Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ \bar{1}$

$\bar{n} : \Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ \bar{n}$

So we are led to define

$\mathsf{Nat} := \Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ \bar{n}$ for any $\bar{n}$.

Two problems arise with this scheme of encoding. The first problem involves mutual recursion. The definiens of Nat contains Nat and $\mathsf{S}, \bar{0}$, while the type of S is $\mathsf{Nat} \to \mathsf{Nat}$ and the type of $\bar{0}$ is Nat. This problem can be addressed by adopting mutually recursive definitions. The second problem is about quantification. We want to define a type Nat for any $\bar{n}$, but right now what we really have is one Nat for each numerals $\bar{n}$. We aims to solve this problem by introducing a new type construct $\iota x.T$ called *self type*. The idea is that the $\iota x.T$ allows T to refer, via bound variable x, to the term which the self type is typing. Thus we define $\mathsf{Nat} := \iota x.\Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ x$. The self type can only be instantiated/generalized by its own subject, so we add the following two rules and the judgement:

$$\frac{\Gamma \vdash t : [t/x]T}{\Gamma \vdash t : \iota x.T}\ \textit{SelfGen} \quad \frac{\Gamma \vdash t : \iota x.T}{\Gamma \vdash t : [t/x]T}\ \textit{SelfInst} \quad \frac{\bar{n} : \Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ \bar{n}}{\bar{n} : \iota x.\Pi y : \mathsf{Nat}.(P\ y \to P(\mathsf{S}\ y)) \to P\ \bar{0} \to P\ x}$$

In this talk, we will introduce a type system called $\mathsf{Selfstar}$ with mutually recursive definitions, self types, and $* : *$. We will see how standard Church- and Scott-encoded datatype can be presented in $\mathsf{Selfstar}$.

References

[1] Henk Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Symbolic Logic*, 3(2):181–215, 1997.

[2] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies)*. Princeton University Press, Princeton, NJ, USA, 1985.

[3] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988.

[4] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic, Volume II.* North-Holland, 1972.

[5] Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, 1972.

[6] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types.* Cambridge University Press, New York, NY, USA, 1989.

[7] Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In *Typed lambda calculi and applications*, pages 328–345. Springer, 1993.

[8] B. Werner. A Normalization Proof for an Impredicative Type System with Large Elimination over Integers. In B. Nordström, K. Petersson, and G. Plotkin, editors, *International Workshop on Types for Proofs and Programs (TYPES)*, pages 341–357, 1992.