

A typed λ -calculus with call-by-name and call-by-value iteration

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 the polymorphic λ calculus one can define closed data types for this data, the iteration scheme is well-typed and β -reduction is always terminating. The much less well-known *Scott encoding* [1] has *case distinction* as a primitive. (For the natural numbers, these are also known as *Parigot numerals* [3,5].) The terms are not typable in polymorphic λ calculus and there is no iteration scheme, but there is a constant time *destructor* (e.g. predecessor).

We will shed a different light on the Scott data types, by viewing the different cases as *continuations*. So, for natural numbers, there is a continuation for the 0-case and a continuation for the $n + 1$ -case (which takes n as input). As Scott data types do not have iteration (or recursion) built in, we add as primitives a *call-by-name iterator* and a *call-by-value iterator*. The reduction rules for the iterators expose the expected evaluation behaviour if one does left-most-reduction in the calculus.

To represent the Scott data types we need simple types plus positive recursive types, where we add the two iterators as additional constants with special reduction rules. We show that the calculus is strongly normalising (SN) by adapting the well-known saturated sets proof in a non-trivial way.

The interest of the calculus lies in the fact that one can *choose* for a call-by-name or call-by-value function by choosing an iteration scheme. The work is partly motivated by our studies into Continuation Calculus, [2,4], which aims at being a simplified model of functional computation. The proof of SN of λ It also implies SN of the typed Continuation Calculus of [4].

We now give some details of the data types and the system, where we limit ourselves to the natural numbers.

Scott numerals. These are defined as follows.

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

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. in the untyped λ -calculus by using a fixed point-combinator).

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

$$\underline{\mathbf{c}_i} := \lambda x_1 \dots x_{\text{ar}(i)}. \lambda c_1 \dots c_k. c_i x_1 \dots x_{\text{ar}(i)}$$

To type Scott numerals we need $\text{Nat} = A \rightarrow (\text{Nat} \rightarrow A) \rightarrow A$. In $\lambda 2$, we cannot do this, unless we extend it with (positive) recursive types, obtaining $\lambda 2\mu$:

$$\text{Nat} := \mu Y. \forall X. X \rightarrow (Y \rightarrow X) \rightarrow X.$$

Using this approach, which is explained in [1], one still does not get any well-founded recursion “for free”, so we have to add iterators.

We view the Scott data types slightly differently, following [2,4], where $\perp$ is taken to be the type of *executable terms* and

$$\text{Nat} := \mu Y. \perp \rightarrow (Y \rightarrow \perp) \rightarrow \perp.$$

A number takes two continuations, $c : \perp$ and $s : \text{Nat} \rightarrow \perp$ and returns c – in case the number is 0 – or sn – in case the number is $n + 1$. So, in λIt , we have simple types, a base type $\perp$ and positive recursive types. (As usual from logic we abbreviate $A \rightarrow \perp$ to $\neg A$.) Furthermore, we have iterators.

The Iterators. Let $B = B^1 \rightarrow \dots \rightarrow B^m \rightarrow \perp$ with each B^k of the shape $\dots \rightarrow \perp$. We abbreviate $B^1 \rightarrow \dots \rightarrow B^m \rightarrow \perp$ to $\overline{B} \rightarrow \perp$. The iterators for Nat are:

$$\begin{array}{c} \text{(It-CBN)} \frac{f_1 : B \quad f_2 : B \rightarrow B \quad c_1 : B^1 \dots c_m : B^m \quad n : \text{Nat}}{\text{ltcbn } f_1 f_2 n c_1 \dots c_m : \perp} \\ \\ \text{(It-CBV)} \frac{f_1 : \neg \neg B \quad f_2 : B \rightarrow \neg \neg B \quad c : \neg B \quad n : \text{Nat}}{\text{ltcbv } f_1 f_2 c n : \perp} \end{array}$$

With reduction rules

$$\begin{array}{l} \text{ltcbn } f_1 f_2 n \bar{c} \longrightarrow_n n (f_1 \bar{c}) (\lambda x : \text{Nat}. f_2 (\lambda \bar{z} : \overline{B}. \text{ltcbn } f_1 f_2 x \bar{z}) \bar{c}) \\ \text{ltcbv } f_1 f_2 c n \longrightarrow_v n (f_1 c) (\lambda x : \text{Nat}. \text{ltcbv } f_1 f_2 (\lambda y : B. f_2 y c) x) \end{array}$$

References

- [1] M. Abadi, L. Cardelli and G. Plotkin, Types for the Scott Numerals, 1993, <http://lucacardelli.name/Papers/Notes/scott2.pdf>.
- [2] B. Geron and H. Geuvers, Continuation calculus, in *Proceedings of the First Workshop on Control Operators and their Semantics* (COS), Eindhoven, The Netherlands, June 24-25, 2013 Eds. Ugo de'Liguoro and Alexis Saurin, EPTCS 127.
- [3] P. Fu, A. Stump, Self Types for Dependently Typed Lambda Encodings, Accepted for RTA-TLCA 2014.
- [4] H. Geuvers, W. Geraedts, B. Geron and J. van Stegeren, A type system for Continuation Calculus, submitted to CL&C 2014.
- [5] M. Parigot, Recursive Programming with Proofs, *Theor. Comput. Sci.*, 94, 2, 1992, pp. 335-336.