A type system for Continuation Calculus

Herman Geuvers[1,2], Wouter Geraedts[1], Bram Geron[3], Judith van Stegeren[1]

[1] Radboud University Nijmegen, the Netherlands
[2] Technical University Eindhoven, the Netherlands
[3] School of Computer Science, University of Birmingham, UK

Abstract

Continuation Calculus (CC), introduced by Geron and Geuvers [2], is a simple foundational model for functional computation. It is closely related to lambda calculus and term rewriting, but it has no variable binding and no pattern matching. It is Turing complete and evaluation is deterministic. Notions like “call-by-value” and “call-by-name” computation are available by choosing appropriate function definitions: e.g. there is a call-by-value and a call-by-name addition function.

In the present paper we extend CC with types, to be able to define data types in a canonical way, and functions over these data types, defined by iteration. Data type definitions follow the so-called “Scott encoding” of data, as opposed to the more familiar “Church encoding”.

The iteration scheme comes in two flavors: a call-by-value and a call-by-name iteration scheme. The call-by-value variant is a double negation variant of call-by-name iteration. The double negation translation allows to move between call-by-name and call-by-value.

Continuation calculus (or CC) [2] is a crossover between term rewriting systems and λ-calculus. Rather than focusing on expressions, continuation calculus treats continuations as its fundamental object. This is realized by restricting evaluation to strictly top-level, discarding the need for evaluation inside contexts. This also fixes an evaluation order, so the representation of a program in CC depends on whether call-by-value or call-by-name is desired. Furthermore, CC “separates code from data” by placing the former in a static *program*, which is sourced for reductions on a term. Variables are absent from terms, and no substitution happens inside terms.

Despite the obvious differences between CC and λ-calculus with continuations (or λ_C), there seems to be a strong correspondence. For instance, it has been suggested [3] that programs in either can be simulated in the other up to parametrized (non)termination, in an untyped setting. If the correspondence turns out to be sufficiently strong, continuation calculus could become an alternative characterization of λ_C, and theorems in one system could apply without much effort to the other.

The purpose of this paper is to strengthen the correspondence between CC and λ-calculus, by introducing a type system for CC and by showing how data types and functions over data can be defined in CC. The type system rejects some undesired terms and the types emphasize the difference between call-by-name and call-by-value. Also, the types pave the way for proving properties of the programs. The types themselves do not enforce termination, because the system is ‘open’: one can add whatever program one wants. However, if the programs on data types are defined using only iteration and non-circular program rules, all programs are terminating, which we show using a translation to a simply typed λ-calculus with data-types and iterators in call-by-name and call-by-value style. We show this λ-calculus to be strongly normalizing.

Informal definition of CC Terms in CC are of the shape $n.t_1.t_2.\dots.t_k$, where n is a name and t_i is again a term. The ‘dot’ denotes binary application, which is left-associative. In CC, terms can be evaluated by applying *program rules* which are of the shape

$$n.x_1.x_2.\dots.x_p \longrightarrow u, (*)$$

where u is a term over variables $x_1 \dots x_p$. However, this rule can only be applied on the ‘top level’:

- reduction is not a congruence;
- rule (*) can only be applied to the term $n.t_1.t_2.\dots.t_k$ in case $k = p$,
- then this term evaluates to $u[t_1/x_1, \dots, t_p/x_p]$.

CC has no pattern matching or variable binding, but it is Turing complete and a faithful translation to and from the untyped λ-calculus can be defined, see [3].

In continuation calculus, the natural numbers are represented by the names **Zero** and **Succ** and the following two program-rules:

$$\begin{array}{lcl} \mathbf{Zero}.c_1.c_2 & \longrightarrow & c_1 \\ \mathbf{Succ}.x.c_1.c_2 & \longrightarrow & c_2.x \end{array}$$

So **Zero** represents 0, **Succ**.**Zero** represents 1, **Succ**.(**Succ**.**Zero**) represents 2 etcetera. This representation of data follows the so-called Scott encoding, which is known from the untyped lambda calculus by defining **Zero** $:= \lambda x\, y.x$, **Succ** $:= \lambda n.\lambda x\, y.y\, n$ (e.g. see [1, 4]). The Scott numerals have "case-distinction" built in (distinguishing between 0 and $n+1$), which can be used to mimic pattern matching. The more familiar Church numerals have iteration built in. For Scott numerals, iteration has to be added, or it can be obtained from the fixed-point combinator in the case of untyped lambda calculus. For CC the situation is similar: we have to add iteration ourselves.

As an example, we define addition in two ways: in call-by-value (CBV) and in call-by-name (CBN) style ([2]).
Example 1.

$$\begin{array}{lcl} \mathbf{AddCBV}.n.m.c & \longrightarrow & n.(c.m).(\mathbf{AddCBV'}.m.c) \\ \mathbf{AddCBV'}.m.c.n' & \longrightarrow & \mathbf{AddCBV}.n'.(\mathbf{Succ}.m).c \end{array}$$

$$\begin{array}{lcl} \mathbf{AddCBN}.n.m.c_1.c_2 & \longrightarrow & n.(m.c_1.c_2).(\mathbf{AddCBN'}.m.c_2) \\ \mathbf{AddCBN'}.m.c_2.n' & \longrightarrow & c_2.(\mathbf{AddCBN}.n'.m) \end{array}$$

For **AddCBV** *we find that* **AddCBV**.(**Succ**^{n}.**Zero**).(**Succ**^{m}.**Zero**).c *evaluates to* c.(**Succ**$^{n+m}$.**Zero**)*: the result of the addition function is computed completely and passed as argument to the continuation* c*. For* **AddCBN***, only a first step in the computation is carried out and then the result is passed to the appropriate continuation* c_1 *or* c_2*.*

Continuation calculus as it occurs in [2] is untyped. In the present work we present a typing system for continuation calculus, using simple types and positive recursive types. The typing system gives the user some guarantee about the meaning and well-formedness of well-typed terms. We also develop a general procedure for defining algebraic data-types as types in CC and for transforming functions defined over these data types into valid typed terms in CC.

References

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

[2] B. Geron and H. Geuvers. Continuation calculus. In *Proceedings of COS 2013*, volume 127 of *EPTCS*, pages 66–85, 2013.

[3] Bram Geron. Continuation calculus, master's thesis. `http://alexandria.tue.nl/extra1/afstversl/wsk-i/geron2013.pdf`, 2013.

[4] J.M. Jansen. Programming in the λ-calculus: From Church to Scott and back. In *The Beauty of Functional Code*, volume 8106 of *Lecture Notes in Computer Science*, pages 168–180, 2013.