A type system for Continuation Calculus

Herman Geuvers
joint work with Bram Geron (Birmingham), Wouter Geraedts (Nijmegen), Judith van Stegeren (Nijmegen)

Radboud University Nijmegen
and
Eindhoven University of Technology
The Netherlands

Types 2014, Paris
March 12, 2014

Contents

Continuation calculus; motivation

- Simple model for functional computation. (Inspired by λ-calculus and term rewriting)
- But: no variable binding, no pattern matching
- Treat continuations as the fundamental object (rather than expressions or data)
- Deterministic computation
- Turing complete
- Call-by-name and Call-by-value via function definitions

Continuation calculus; rules

- Infinite set of names, N, usually indicated by a capital.
- Terms: either a name, or two terms combined by a dot.

$$T ::= N \mid (T.T)$$

- We write $a.b.c$ as shorthand for $(a.b).c$.
- A program P consists of a set of rules, of the form

$$N.x_1.\ldots.x_n \to t$$

  - N is a name and $x_1, \ldots, x_n$ are distinct variables,
  - t is a term over the variables $x_1, \ldots, x_n$
- Proviso : for each name N there is at most one rule. We say that the rule defines the name N.
- Evalutation (reduction) of terms is defined by
  if $N.x_1.\ldots.x_n \to t \in P$, then

$$N.t_1.\ldots.t_n \to_P t[t_1/x_1, \ldots, t_n/x_n]$$

Continuation calculus, remarks

Assume we have a program rule

$$N.x_1.\dots.x_n \to t.$$

- A term $N.t_1.\dots.t_k$ with $k \neq n$ does not reduce at all.
- There is only head reduction: $M.(N.t_1.\dots.t_n).\dots$ does not reduce.
- Reduction is trivially confluent, because at most one step is possible.
- Reduction is not necessarily terminating, e.g.

$$\mathrm{Omega}.x \to x.x$$

- There is no pattern matching (as one has e.g. in TRS)
- There is no variable binding (as one has in λ-calculus)
- CC is Turing complete

Continuation calculus, examples

- booleans

$$\begin{aligned} \mathrm{True}.x.y &\to x \\ \mathrm{False}.x.y &\to y \end{aligned}$$

- natural numbers

$$\begin{aligned} \mathrm{Zero}.z.s &\to z \\ \mathrm{Succ}.x.z.s &\to s.x \end{aligned}$$

  Interpretation of data as CC-terms:

$$\langle m \rangle := \mathrm{Succ}^m.\mathrm{Zero}$$

  which is $\mathrm{Succ}.(\mathrm{Succ}.\ldots.(\mathrm{Succ}.\mathrm{Zero})\ldots)$ m-times.

- lists

$$\begin{aligned} \mathrm{Nil}.n.l &\to n \\ \mathrm{Cons}.x.y.n.l &\to l.x.y \end{aligned}$$

Continuation calculus, Scott data types

Compute functions by computing a value and passing it on to the next function (continuation)

Natural numbers

$$\begin{aligned} \mathrm{Zero}.z.s &\rightarrow z \\ \mathrm{Succ}.x.z.s &\rightarrow s.x \end{aligned}$$

Idea: A number takes two continuations z and s and either

- continues with z (if the number is Zero)
- continues with $s.t$ (if the number is $\mathrm{Succ}.t$)

The definition of data in CC follows the Scott data types approach in untyped λ-calculus (as opposed to the Church approach):

$$\begin{aligned} \mathrm{Zero} &:= \lambda\, z\, s.z \\ \mathrm{Succ} &:= \lambda\, x\, z\, s.s\, x \end{aligned}$$

The Scott approach has case distinction as basic and not iteration.

Continuation calculus, example

Natural numbers

$$\begin{aligned} \mathrm{Zero}.z.s &\rightarrow z \\ \mathrm{Succ}.x.z.s &\rightarrow s.x \end{aligned}$$

For addition we want

$$\mathrm{Add}.\langle m \rangle.\langle p \rangle.r \twoheadrightarrow r.\langle m+p \rangle$$

The algorithm for addition that we implement is basically the following term rewriting system.

$$\begin{aligned} \mathrm{plus}(0, m) &\rightarrow m \\ \mathrm{plus}(S(p), m) &\rightarrow \mathrm{plus}(p, S(m)) \end{aligned}$$

Continuation calculus, computing addition

Natural numbers

$$\begin{aligned} \mathrm{Zero}.z.s &\rightarrow z \\ \mathrm{Succ}.x.z.s &\rightarrow s.x \end{aligned}$$

We want $\mathrm{Add}.\langle m \rangle.\langle p \rangle.r \twoheadrightarrow r.\langle m+p \rangle$
So we take as program rule for Add:

$$\mathrm{Add}.x.y.r \quad \rightarrow \quad x.(r.y).t \text{ with } t \text{ yet to be found}$$

Because then

$$\begin{aligned} \mathrm{Add}.\mathrm{Zero}.y.r &\rightarrow \mathrm{Zero}.(r.y).t \rightarrow r.y \\ \mathrm{Add}.(\mathrm{Succ}.x).y.r &\rightarrow \mathrm{Succ}.x.(r.y).t \rightarrow t.x \stackrel{??}{\rightarrow} \mathrm{Add}.x.(\mathrm{Succ}.y).r \end{aligned}$$

So take $t = \mathrm{B}.y.r$ with

$$\mathrm{B}.y.r.x \rightarrow \mathrm{Add}.x.(\mathrm{Succ}.y).r$$

Continuation calculus, call-by-value addition

Natural numbers

$$\begin{aligned}\mathrm{Zero}.z.s &\to z\\ \mathrm{Succ}.x.z.s &\to s.x\end{aligned}$$

$$\begin{aligned}\mathrm{Add}.x.y.r &\to x.(r.y).(\mathrm{B}.y.r)\\ \mathrm{B}.y.r.x &\to \mathrm{Add}.x.(\mathrm{Succ}.y).r\end{aligned}$$

Lemma. For all $m, p \in \mathbb{N}$: $\mathrm{Add}.\langle m\rangle.\langle p\rangle.r \twoheadrightarrow r.\langle m+p\rangle$
The function is call-by-value in its first argument:
$\mathrm{Add}.x.y.r$ first computes x, say $x = \langle p\rangle$, then it returns $r.(\mathrm{Succ}^p.y)$

Continuation calculus, call-by-name addition

Natural numbers

$$\begin{aligned}\mathrm{Zero}.z.s &\to z\\ \mathrm{Succ}.x.z.s &\to s.x\end{aligned}$$

Call-by-name addition function:

$$\begin{aligned}\mathrm{AddCBN}.x.y.z.s &\to x.(y.z.s).(\mathrm{C}.y.s)\\ \mathrm{C}.y.s.x' &\to s.(\mathrm{AddCBN}.x'.y)\end{aligned}$$

Then we have

$$\begin{aligned}\mathrm{AddCBN}.(\mathrm{Succ}.\langle m\rangle).\langle p\rangle.z.s &\twoheadrightarrow \mathrm{Succ}.\langle m\rangle.(\langle p\rangle.z.s).(\mathrm{C}.\langle p\rangle.s)\\ &\twoheadrightarrow \mathrm{C}.\langle p\rangle.s.\langle m\rangle\\ &\twoheadrightarrow s.(\mathrm{AddCBN}.\langle m\rangle.\langle p\rangle)\end{aligned}$$

which is a normal form.

We need a notion of observational equivalence $\approx$ to prove

$$\mathrm{AddCBN}.\langle m\rangle.\langle p\rangle \approx \langle m+p\rangle.$$

Continuation calculus, Observational equivalence

Definition: Terms M and N are observationally equivalent under program P, when for all extension programs $P' \supseteq P$ and terms X:

$$X.M \downarrow_{P'} \Longleftrightarrow X.N \downarrow_{P'}$$

where $T \downarrow_{P'}$ denotes that T terminates with the program rules P'.
Notation: $M \approx_P N$,

A step back: Typed λ-calculus for Scott data?

Scott numerals:

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

To type this 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.

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

with rule

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

[Abadi, Cardelli, Plotkin 1993]
NB. One does not get iteration (recursion) "for free".

Types for Scott data in CC

In CC everything happens on the 'top level'. Call this level $\bot$.

Types:

$$T := \bot | X | T \to T | \mu X.T$$

under the condition that X is positive in T.

Type equality rule

$$\mu X.T = T[\mu X.T/X]$$

Examples:

$$\begin{array}{rcl} \texttt{bool} & := & \bot \to \bot \to \bot \\ \texttt{nat} & := & \bot \to (\texttt{nat} \to \bot) \to \bot \\ \texttt{list}_A & := & \bot \to (A \to \texttt{list}_A \to \bot) \to \bot \end{array}$$

$$\begin{array}{rcl} \mathrm{Zero}.z.s & \to & z \\ \mathrm{Succ}.x.z.s & \to & s.x \\ \mathrm{Nil}.n.l & \to & n \\ \mathrm{Cons}.x.y.n.l & \to & l.x.y \end{array}$$

Typing addition in CC

$$\texttt{nat} := \bot \to (\texttt{nat} \to \bot) \to \bot$$

$$\begin{array}{rcl} \mathrm{Zero}.z.s & \to & z \\ \mathrm{Succ}.x.z.s & \to & s.x \\ \mathrm{AddCBV}.x.y.r & \to & x.(r.y).(\mathrm{B}.y.r) \\ \mathrm{B}.y.r.x & \to & \mathrm{AddCBV}.x.(\mathrm{Succ}.y).r \end{array}$$

Then

$$\mathrm{AddCBV} : \texttt{nat} \to \texttt{nat} \to \neg\neg\texttt{nat}$$

where $\neg A$ is defined as $A \to \bot$.
We can also type the call-by-name addition

$$\mathrm{AddCBN} : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$$

Type system for CC judgments

- program signature Σ: finite list of distinct names assigned to types
  $\Sigma = n_1 : A_1, \ldots, n_p : A_p$
- typing context Γ: finite list of distinct variables assigned to types
  $\Gamma = x_1 : A_1, \ldots, x_n : A_n$
- Σ gives the types of the names (specific for a program P). The context is just a "temporary" set of variables, to define program rules.
- Two kinds of judgment:
  1. $\Sigma \vdash P$ to express that, given a program signature Σ, P is a well-typed program. (So P will consist of program rules.)
  2. $\Gamma \vdash_\Sigma M : A$ to express that the term M with free variables in Γ has type A, given signature Σ and context Γ.

Derivation rules

- The derivation rules for typing judgments:

$$\frac{x : A \in \Gamma}{\Gamma \vdash_\Sigma x : A} \qquad \frac{n : A \in \Sigma}{\Gamma \vdash_\Sigma n : A}$$

$$\frac{\Gamma \vdash_\Sigma M : A \to B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M.N : B} \qquad \frac{\Gamma \vdash_A M : A \quad A = B}{\Gamma \vdash_\Sigma M : B}$$

- The derivation rules for program judgments:

$$\frac{}{\Sigma \vdash \emptyset}$$

$$\frac{\Sigma \vdash P \quad \vec{x} : \vec{A} \vdash_\Sigma q : \bot \quad n : A_1 \to \ldots \to A_k \to \bot \ \in \ \Sigma}{\Sigma \vdash P \cup \{n.x_1.\ldots.x_k \longrightarrow q\}}$$

if n not defined in P

(NB. $\vec{x} : \vec{A}$ denotes $x_1 : A_1, \ldots, x_k : A_k$)

Call-by-name and call-by-value iteration

To be able to define functions in CC in a generic and safe way, we add for every data-type two program rules for iteration:

- $\mathrm{ItCBN}_{D \to B}$ for call-by-name iteration from data-type D to B, Idea: compute first constructor of the output of type B and pass to the appropriate continuation for type B.
- $\mathrm{ItCBV}_{D \to B}$ for call-by-value iteration from data-type D to B. Idea: compute input value completely and then pass on to the continuation of type $B \to \bot$.

Call-by-name and call-by-value iteration for nat

We show the case for $D = B = \texttt{nat}$.

Call-by-name:

$$\frac{f_1 : \texttt{nat} \quad f_2 : \texttt{nat} \to \texttt{nat} \quad x : \texttt{nat} \quad c_1 : \bot \quad c_2 : \texttt{nat} \to \bot}{\mathrm{ItCBN}_{\texttt{nat}\to\texttt{nat}}.f_1.f_2.x.c_1.c_2 : \bot}$$

with associated reduction rules (program rules).

$$\mathrm{ItCBN}_{\texttt{nat}\to\texttt{nat}}.f_1.f_2 : \texttt{nat} \to \texttt{nat}$$

Call-by-value:

$$\frac{f_1 : \neg\neg\texttt{nat} \quad f_2 : \texttt{nat} \to \neg\neg\texttt{nat} \quad x : \texttt{nat} \quad c : \neg\texttt{nat}}{\mathrm{ItCBV}_{\texttt{nat}\to\texttt{nat}}.f_1.f_2.x.c : \bot}$$

with associated reduction rules (program rules).

$$\mathrm{ItCBV}_{\texttt{nat}\to\texttt{nat}}.f_1.f_2 : \texttt{nat} \to \neg\neg\texttt{nat}$$

Call-by-name and call-by-value iteration: program rules

Rules for $\mathrm{ItCBN}_{\mathtt{nat}\to\mathtt{nat}}$:

$$\begin{aligned}
\mathrm{ItCBN}.f_1.f_2.x.c_1.c_2 &\longrightarrow x.(f_1.c_1.c_2).(\mathrm{ItCBN}^{\mathsf{Succ}}.f_1.f_2.c_1.c_2)\\
\mathrm{ItCBN}^{\mathsf{Succ}}.f_1.f_2.c_1.c_2.x_1 &\longrightarrow f_2.(\mathrm{ItCBN}.f_1.f_2.x_1).c_1.c_2
\end{aligned}$$

Rules for $\mathrm{ItCBV}_{\mathtt{nat}\to\mathtt{nat}}$:

$$\begin{aligned}
\mathrm{ItCBV}.f_1.f_2.x.c &\longrightarrow x.(f_1.c).(\mathrm{ItCBV}^{\mathsf{Succ},1}.f_1.f_2.c)\\
\mathrm{ItCBV}^{\mathsf{Succ},1}.f_1.f_2.c.x_1 &\longrightarrow \mathrm{ItCBV}.f_1.f_2.x_1.(\mathrm{ItCBV}^{\mathsf{Succ},2}.f_1.f_2.c)\\
\mathrm{ItCBV}^{\mathsf{Succ},2}.f_1.f_2.c.r_1 &\longrightarrow f_2.r_1.c
\end{aligned}$$

Combining call-by-value and call-by-name

A storage operator in CC

$$\begin{aligned} \mathrm{StoreNat}.n.r &\longrightarrow n.(r.\mathrm{Zero}).(\mathrm{A}.r) \\ \mathrm{A}.r.m &\longrightarrow \mathrm{StoreNat}.m.(\mathrm{B}.r) \\ \mathrm{B}.r.m' &\longrightarrow r.(\mathrm{Succ}.m') \end{aligned}$$

Then $\mathrm{StoreNat} : \mathtt{nat} \to \neg\neg\mathtt{nat}$. If $t \approx \langle p \rangle$, then for any r:

$$\mathrm{StoreNat}.t.r \twoheadrightarrow r.(\mathrm{Succ}^p.\mathrm{Zero})$$

$\mathrm{StoreNat}$ always first evaluates t, before using it. (Mimicking CBV by CBN.)
In the reverse direction: $\mathrm{UnstoreNat} : \neg\neg\mathtt{nat} \to \mathtt{nat}$ defined by: given $f : \neg\neg\mathtt{nat}$, $z : \bot$, $s : \mathtt{nat} \to \bot$,

$$\begin{aligned} \mathrm{UnstoreNat}.f.z.s &\longrightarrow f.(\mathrm{UseNat}.z.s) \\ \mathrm{UseNat}.z.s.n &\longrightarrow n.z.s \end{aligned}$$

Lemma For all $n \in \mathbf{N}$,
$\mathrm{UnstoreNat}.(\mathrm{StoreNat}.\langle n \rangle) \approx \langle n \rangle$.

Results

- Confluence is trivial, because (untyped) CC is deterministic
- Subject reduction holds
- Strong Normalization holds for all CC programs written using
  - constructors of data types and
  - iterators (cbn and cbv) and
  - non-circular well-typed rules
- The SN proof is by
  - translation to a typed λ-calculus with simple and pos.rec. types and CBN and CBV iterator combinators;
  - proving that this typed λ-calculus is SN