A type system for Continuation Calculus

Herman Geuvers

Radboud University Nijmegen,   Technical University Eindhoven, the Netherlands

Wouter Geraedts   Bram Geron

Radboud University Nijmegen, the Netherlands   School of Computer Science, University of Birmingham, UK

Judith van Stegeren

Radboud University Nijmegen, the Netherlands

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.

1   Introduction

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. This we show in a separate paper.

1.1 Informal definition of CC

Terms in CC are of the shape $n.t_1.t_2.\ldots.t_k$, 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

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

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

- reduction is not a congruence;

- rule (*) can only be applied to the term $n.t_1.t_2.\ldots.t_k$ in case $k = p$,

- then this term evaluates to $u[t_1/x_1, \ldots, t_p/x_p]$.

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:

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

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** $:= \lambda xy.x$, **Succ** $:= \lambda n.\lambda xy.yn$ (e.g. see [1,5]). 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 1.1

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

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

For **AddCBV** *we find that* **AddCBV**.($\mathbf{Succ}^n$.**Zero**).($\mathbf{Succ}^m$.**Zero**).c *evaluates to* $c.(\mathbf{Succ}^{n+m}.\mathbf{Zero})$: *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. 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. In a separate paper we prove termination of all well-formed iterative CC programs [4].

2 Formal definition of CC

For the detailed formal definition, we refer to [2]. Here we give a short recap of CC. The *Terms* are generated by the following grammar:

$$\mathcal{U} ::= \mathcal{N} \mid (\mathcal{U}.\mathcal{U}),$$

where $\mathcal{N}$ is any infinite set of *names*. So, the terms do not contain variables. (One could add them, but it's not necessary.) Names act as labels for functions and constructors in CC. *Names* in CC start with an uppercase letter and are printed in **bold**. The dot is left-associative, so we write $(((n.t_1).t_2).\ldots.t_k)$ as $n.t_1.\ldots.t_k$.

The *head* of a term is its 'leftmost' name: $head(n.t_1.t_2.\ldots.t_k) = n$. The *length* of a term is the number of dots towards the head: $length(n.t_1.t_2.\ldots.t_k) = k$.

To define programs we assume any infinite set $\mathcal{V}$ of *variables*. A *program* is a set of *rules*, each of the following shape

$$n.x_1.x_2.\ldots.x_k \longrightarrow u$$

where the x_i are distinct variables and u is a *term over the variables* $x_1, \ldots, x_k$, so u is a term that may use, apart from names, also the variables $x_1, \ldots, x_k$. We say that the rule *defines* the name n. Within a program, a name may occur at most once as the head of a rule. If P is a program, the *domain of P*, $dom(P)$ is the set of names that is defined in P.

Let a program P be given. A term can be *evaluated in P* by applying one of the rules of P to the whole term as follows. Suppose P contains the rule $n.x_1.x_2.\ldots.x_k \longrightarrow u$. Then

$$n.t_1.t_2.\ldots.t_k \longrightarrow_P u\,[t_1/x_1, \ldots, t_k/x_k]$$

where the latter denotes the substitution of $t_1, \ldots, t_k$ for $x_1, \ldots, x_k$. We usually omit the subscript P and just write $\longrightarrow$, as P will be clear from the context.

It should be noted that one does not evaluate 'under the application'. To make this explicit we introduce some more terminology. A name n has *arity k* in P if P contains a rule of the form

$$n.x_1.x_2.\ldots.x_k \longrightarrow u$$

Similarly, a term t has an *arity* in P if $arity(head(t)) \leq length(t)$ and we define

$$arity(t) := arity(head(t)) - length(t)$$

A term t is *defined in P* if $head(t) \in dom(P)$. Otherwise t is *undefined in P*. A defined term is either *complete*, if $arity(t) = 0$, or *incomplete* if $arity(t) > 0$, or *invalid* if it has no arity.

We write $A \longrightarrow B$ for "B is a reduct of A" and $A \longrightarrow\!\!\!\!\rightarrow B$ for "A reduces in zero or more steps to B". Because every name is defined at most once in the set of program rules, every program is a deterministic rewriting system.

A term M is said to be *terminating* (or strongly normalizing) in P if there exists a reduct N such that N can no longer be rewritten using the rules from P. We write

$$M \downarrow_P \quad \text{if } M \text{ can not be rewritten using the rules of } P$$
$$M \downarrow\!\!\downarrow_P \quad \text{if } \exists N (M \longrightarrow\!\!\!\!\rightarrow N \wedge N \downarrow_P)$$

Note that M $\longrightarrow\!\!\!\!\rightarrow\downarrow_P$ implies $M \longrightarrow\!\!\!\!\rightarrow N \downarrow_P$, as $M \longrightarrow\!\!\!\!\rightarrow N$ can mean that M rewrites to N in zero steps.

The simplest notion of equality between terms in CC is the transitive, symmetric, reflexive closure of $\longrightarrow$, which we denote by $=_P$. So $M_1 =_P M_2$ in case there is an N such that $M_1 \longrightarrow\!\!\!\!\rightarrow_P N$ and $M_2 \longrightarrow\!\!\!\!\rightarrow_P N$. This is an interesting equivalence relation, however, it is much too fine, as we show in the following example. (See also [2].)

Example 2.1 *For the call-by-name addition of Example 1.1, we find that*

$$\mathbf{AddCBN}.(\mathbf{Succ}.\mathbf{Zero}).\mathbf{Zero}.c_1.c_2 \longrightarrow\!\!\!\!\rightarrow \mathbf{AddCBN}'.\mathbf{Zero}.c_2.\mathbf{Zero} \longrightarrow\!\!\!\!\rightarrow c_2.(\mathbf{AddCBN}.\mathbf{Zero}.\mathbf{Zero})$$

If we also compute $\mathbf{Succ}.\mathbf{Zero}.c_1.c_2$*, we obtain* $c_2.\mathbf{Zero}$*, which is not the same term, so we don't have* $\mathbf{AddCBN}.(\mathbf{Succ}.\mathbf{Zero}).\mathbf{Zero}.c_1.c_2 =_P \mathbf{Succ}.\mathbf{Zero}.c_1.c_2$.
If we also allow computing 'under the function c_2*', the terms are still not equal:* $\mathbf{AddCBN}.\mathbf{Zero}.\mathbf{Zero}$ *does not reduce to* $\mathbf{Zero}$*. However, when supplied with two continuations,* d_1 *and* d_2*, they are equal:* $\mathbf{AddCBN}.\mathbf{Zero}.\mathbf{Zero}.d_1.d_2 \longrightarrow\!\!\!\!\rightarrow d_1$ *and* $\mathbf{Zero}.d_1.d_2 \longrightarrow\!\!\!\!\rightarrow d_1$.

In the example we see two terms M and N which are 'equal for all practical purposes', but we don't have $M =_P N$. We say that two terms M and N are *observationally equivalent* under program P, notation $M \approx_P N$, if for all extension programs $P' \supseteq P$ and all terms X

$$X.M \downarrow\!\!\downarrow_{P'} \iff X.N \downarrow\!\!\downarrow_{P'}$$

We recall some properties about $\approx_P$ from [2]. Proofs can be found in [2].

Lemma 2.2 *The relation* $\approx_P$ *is a congruence, that is, if* $M_1 \approx_P M_2$ *and* $N_1 \approx_P N_2$ *, then* $M_1.N_1 \approx_P M_2.N_2$.

Lemma 2.3 *Let* M, N *be terms of arity* k*. If* $M.c_1. \ldots .c_k =_P N.c_1. \ldots .c_k$ *for fresh names* $c_1, \ldots, c_k$*, then* $M \approx_P N$.

Corollary 2.4 *If* $M =_P N$ *and* $arity(M) = arity(N) = 0$. *Then* $M \approx_P N$.

It is not in general the case that $M =_P N$ implies $M \approx_P N$. The reason is that reduction of a term need not respect the arity, giving rise to undesired situations, as can be seen in the following example (also taken from [2]). Our typing system will prevent these situations to occur.

Example 2.5 *Consider the following program rules*

$$\mathbf{Id}.x \quad \longrightarrow \quad x$$
$$\mathbf{Omega}.x \quad \longrightarrow \quad x.x$$

Then $\mathbf{Id}.\mathbf{Omega} \longrightarrow \mathbf{Omega}$*, which is an incomplete term. If we appending another term to* $\mathbf{Id}.\mathbf{Omega}$*, it becomes invalid:* $\mathbf{Id}.\mathbf{Omega}.M$ *has no arity. On the other hand,* $\mathbf{Omega}.\mathbf{Omega} \longrightarrow \mathbf{Omega}.\mathbf{Omega}$*, so this term is non-terminating. Hence,* $\mathbf{Id}.\mathbf{Omega} \longrightarrow \mathbf{Omega}$*, but* $\mathbf{Id}.\mathbf{Omega} \not\approx_P \mathbf{Omega}$.

The type system will prevent situations as in Example 2.5, by making the program rule for **Omega** not 'well-typed' (and thereby not allowed). Also note that **Id**.**Omega** $\not\approx_P$ **Omega** is only possible because arity(**Id**.**Omega**) $\neq$ arity(**Omega**). The type system will make sure that, if $M \longrightarrow N$, then N also has arity 0.

3 Types for Continuation Calculus

Definition 3.1 *We define* types *of CC as follows.*

$$Type := \bot \mid Var \mid (Type \rightarrow Type) \mid \mu Var.Type.$$

where, in $\mu X.\Phi$, we require Φ to be of the shape

$$\sigma^1 \rightarrow \ldots \rightarrow \sigma^n \rightarrow \bot \text{ (with } n \geq 0),$$

with each σ^i of the shape

$$\tau_1^i \rightarrow \ldots \rightarrow \tau_{a_i}^i \rightarrow \bot \text{ (with } a_i \geq 0),$$

where each τ_j^i is either X or does not contain X.

As usual, we leave out the parentheses around functions types, so $A \rightarrow B \rightarrow C$ always means $A \rightarrow (B \rightarrow C)$.

As a consequence of the above definition, if we have a type $\mu X.\Phi(X)$, then X occurs *positively* in Φ. We could have been more liberal, by allowing *all* types $\mu X.\Phi(X)$ where X occurs positively in $\Phi(X)$, but we don't need that to interpret first order data-types. We could also have been more restrictive by only allowing arrow-types that end with $\bot$, so replacing the clause Type $\rightarrow$ Type by Type, ..., Type $\rightarrow \bot$, where the left-hand side denotes a finite sequence of types and $\sigma_1, \ldots, \sigma_n \rightarrow \bot$ is to be understood as $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \bot$. This however complicates the syntax in an inessential way.

The intention of the recursive type $\mu X.\Phi(X)$ is that it denotes the type σ for which $\sigma = \Phi(\sigma)$. To give the μ-types their semantics, we introduce type equalities.

Definition 3.2 *We define* equality between types, $\sigma = \tau$, *as the least equivalence relation that can be derived using the following rules.*

$$\frac{}{\mu X.\tau = \tau \left[\mu X.\tau / X \right]} \ (\mu\text{-conv}) \qquad\qquad \frac{\sigma = \tau \quad \upsilon = \rho}{\sigma \rightarrow \upsilon = \tau \rightarrow \rho} \ (f\text{-conv})$$

For a program rule $n.x_1. \ldots .x_n \longrightarrow u$ with $x_1 : \tau_1, \ldots, x_k : \tau_k$, we will define n to have the following type: $n : \tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow \bot$. So $\bot$ will be used as the type of complete CC-terms. This is very much in line with the approach taken by Miquel [7].

CC-types will be printed in **bold**. For example, the type representing natural numbers $\mathbb{N}$ will be printed as **Nat**. Abstract types (i.e. for any $\sigma \in$ Type) are printed as σ, τ, A, B, etc.

Example 3.3 *The types in CC of some well-known algebraic data-types*

$$
\begin{aligned}
\textbf{Bool} \quad &:= \quad \bot \rightarrow \bot \rightarrow \bot \\
\textbf{Nat} \quad &:= \quad \mu T.\bot \rightarrow (T \rightarrow \bot) \rightarrow \bot \\
\textbf{List}_A \quad &:= \quad \mu T.\bot \rightarrow (A \rightarrow T \rightarrow \bot) \rightarrow \bot \\
\textbf{BinTree}_A \quad &:= \quad \mu T.\bot \rightarrow (A \rightarrow T \rightarrow T \rightarrow \bot) \rightarrow \bot
\end{aligned}
$$

Convention 3.4 *We make use of the convention in logic to define $\neg A$ as $A \to \bot$ to introduce $\neg\sigma$ as an abbreviation for the type $\sigma \to \bot$. Similarly, $\neg\neg\sigma$ denotes $(\sigma \to \bot) \to \bot$.*

Definition 3.5 *A* program signature Σ *is a finite set* $\mathcal{N} \times Type$

$$\Sigma = n_1 : \sigma_1, \ldots, n_p : \sigma_p \quad \text{(with all } n_i \text{ distinct)}$$

A typing context Γ *is a finite set* $\mathcal{V} \times Type$:

$$\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n \quad \text{(with all } x_i \text{ distinct)}$$

The signature gives the types of the names; it is constructed specifically for a program P. The context is just a "temporary" set of variables; contexts will be used to define program rules.

We are interested in two kinds of judgment:

1. $\Sigma \vdash P$, a *program judgment*, to express that, given a program signature Σ, P is a well-typed program. So P will consist of program rules.

2. $\Gamma \vdash_\Sigma M : A$, a *typing judgment*, to express that M – a term with free variables in Γ – has type A, given program signature Σ and typing context Γ.

Definition 3.6 *The derivation rules to derive* typing judgments *are the following*

$$(Var) \quad \frac{x : \sigma \in \Gamma}{\Gamma \vdash_\Sigma x : \sigma} \qquad\qquad (Name) \quad \frac{n : \sigma \in \Sigma}{\Gamma \vdash_\Sigma n : \sigma}$$

$$(Appl) \quad \frac{\Gamma \vdash_\Sigma M : \sigma \to \tau \quad \Gamma \vdash_\Sigma N : \sigma}{\Gamma \vdash_\Sigma M.N : \tau} \qquad (\text{=-}conv) \quad \frac{\Gamma \vdash_\Sigma M : \sigma \quad \sigma = \tau}{\Gamma \vdash_\Sigma M : \tau}$$

Definition 3.7 *The derivation rules to derive* program judgments *are the following*

$$(Nil) \quad \frac{}{\Sigma \vdash \emptyset}$$

$$(Cons) \quad \frac{\Sigma \vdash P \quad x_1 : A_1, \ldots, x_k : A_k \vdash_\Sigma q : \bot \quad n : A_1 \to \ldots \to A_k \to \bot \in \Sigma}{\Sigma \vdash P \cup \{n.x_1.\ldots.x_k \longrightarrow q\}} \text{ if } n \text{ not defined in } P$$

We say that program P *is well-typed in* Σ *in case* $\Sigma \vdash P$. *Usually,* Σ *will be clear and we just say that P is well-typed. Similarly, we say that the program rule $n.x_1.\ldots.x_k \longrightarrow q$ is well-typed P in case $P \cup \{n.x_1.\ldots.x_k \longrightarrow q\}$ is well-typed.*

The second and third premise in the (Cons) rule say that the types of $n.x_1.\ldots.x_k$ and q should be both $\bot$. This guarantees that we can only rewrite terms of type $\bot$.

Example 3.8   *1. Recall the term **Zero** with rule **Zero**.$c_1.c_2 \longrightarrow c_1$ and . We easily verify that this rule is well-typed if we let **Zero** : $\bot \to (\mathbf{Nat} \to \bot) \to \bot$, i.e. **Zero** : **Nat**.*

2. *Similarly, recall the rule for **Succ**: **Succ**.$x.c_1.c_2 \longrightarrow c_2.x$. It is well-typed if we let **Succ** : **Nat** $\to \bot \to (\mathbf{Nat} \to \bot) \to \bot$, i.e. **Succ** : **Nat** $\to$ **Nat**.*

3. *Recalling the definition of* **AddCBV** *in Example 1.1, we see that* **AddCBV** *takes arguments of type* **Nat**, **Nat** *and* **Nat** $\to \perp$ *to produce a term of type* $\perp$. *So the rule is well-typed if we take* **AddCBV** : **Nat** $\to$ **Nat** $\to \neg\neg$**Nat**. *(NB.* **AddCBV**$'$: **Nat** $\to (\neg$**Nat**$) \to$ **Nat** $\to \perp$.*)*

4. *For the definition of* **AddCBN** *in Example 2.1, we see that* **AddCBN** : **Nat** $\to$ **Nat** $\to$ **Nat**.

5. *To type the rule for* **Omega** *in Example 2.5, we need* **Omega** : $\sigma \to \perp$ *with* $\sigma = \sigma \to \perp$. *But there is no type* σ *for which* $\sigma = \sigma \to \perp$, *so the rule for* **Omega** *is not well-typed.*

We have the following properties.

Lemma 3.9   *1. [Substitution] If $n : \tau \in \Sigma$, $\vdash_\Sigma t : \sigma$ and $\vdash_\Sigma q : \tau$, then $\vdash_\Sigma t[q/n] : \sigma$.*

2. *[Subject reduction] If $\vdash_\Sigma t : \sigma$, and $t \longrightarrow p$, then $\sigma = \perp$ and $\vdash_\Sigma p : \perp$.*

Proof   1. By induction on the derivation of $\vdash_\Sigma t : \sigma$.

2. Using the first property. If $t \longrightarrow p$ by the rule $n.x_1. \ldots .x_k \longrightarrow q$, then $t = n.t_1. \ldots .t_k$ and $p = q[t_1/x_1, \ldots, t_k/x_k]$.

   We have $x_1 : A_1, \ldots, x_k : A_k \vdash_\Sigma q : \perp$ and $x_1 : A_1, \ldots, x_k : A_k \vdash_\Sigma n.x_1. \ldots .x_k : \perp$, so by substitution (where we now substitute terms for variables, but the argument is the same) we have $t : \perp$ and $p : \perp$. $\square$

We have the following Corollary of the above and of Corollary 2.4.

Corollary 3.10 *If M and N are well-typed terms of type $\perp$ and $M =_P N$, then $M \approx_P N$.*

3.1   Data types in CC

We have seen the definitions of the types of booleans, natural numbers and lists in Example 3.3. Here we give a general way of defining constructors and first order algebraic data types in CC. (That is, for now we don't allow higher order types in the constructor types.)

Definition 3.11 *A first order data type will be written as*

$$\mathbf{data-type} \quad D \quad \textit{with constructors}$$
$$\mathbf{C}_1^D \quad : \quad D_1^1 \to \ldots \to D_{a_1}^1 \to D$$
$$\ldots$$
$$\mathbf{C}_n^D \quad : \quad D_1^n \to \ldots \to D_{a_n}^n \to D$$

where each of the D_j^i is either D or a type expression that does not contain D. If D is clear from the context, we will omit it as a superscript and write $\mathbf{C}_i$ in stead of $\mathbf{C}_i^D$.

This defines an algebraic data-type D with n constructors with names $\mathbf{C}_1, \ldots, \mathbf{C}_n$. Each constructor $\mathbf{C}_i$ has arity a_i, which can also be 0, and then the constructor is a constant.

Convention 3.12 *To simplify notation later, we abbreviate the list of argument types of a constructor, writing T^1 for $D_1^1 \ldots D_{a_1}^1$ etc. As a matter of fact, this is a kind of uncurrying, replacing $\mathbf{C}_1 : D_1^1 \to \ldots \to D_{a_1}^1 \to D$ by $\mathbf{C}_1 : D_1^1 \times \ldots \times D_{a_1}^1 \to D$, and then we abbreviate $D_1^1 \times \ldots \times D_{a_1}^1$ to T^1.*

For every constructor we will introduce a name in CC and a rule that defines it. This program rule acts as a destructor of D. If a term of type D has constructor $\mathbf{Cons}_i^D$ as its head, all the arguments of that constructor $t_1^i, \ldots, t_{a_i}^i$ will be returned to the corresponding continuation c_i.

Example 3.13 *Consider the algebraic data type of lists over a type A, $\mathbf{List}_A$: In Example 3.3, we have defined this data-type in CC $\mu T. \bot \to (A \to T \to \bot) \to \bot$. The constructors for lists are added to CC by introducing the following program rules to our program, where $\mathbf{Nil} : \bot$ and $\mathbf{Cons} : A \to \mathbf{List}_A \to \bot$.*

$$
\begin{aligned}
\mathbf{Nil}.c_1.c_2 &\longrightarrow c_1 \\
\mathbf{Cons}.x_1.x_2.c_1.c_2 &\longrightarrow c_2.x_1.x_2
\end{aligned}
$$

We now give the general definition of first order data-type in CC.

Definition 3.14 *Given a first order data type D as in Definition 3.11 with n constructors, where, for $1 \le i \le n$, $\mathbf{C}_i : D_1^i \to \ldots D_{a_i}^i \to D$, we define the following type D in CC.*

$$
D := \mu X.(T^1[X/D] \to \bot) \to \ldots \to (T^n[X/D] \to \bot) \to \bot.
$$

For $i \in [1 \ldots n]$, we add the following constructor $\mathbf{Cons}_i^D$ to the signature Σ.

$$
\mathbf{Cons}_i^D : T^i \to D.
$$

Finally, we add for each i ($1 \le i \le n$) the following program rule that acts as a destructor for D.

$$
\mathbf{Cons}_i^D.x_1^i. \ldots .x_{a_i}^i.c_1. \ldots .c_n \longrightarrow c_i.x_1^i. \ldots .x_{a_i}^i
$$

So, in CC we always have $\mathrm{arity}(\mathbf{Cons}_i^D) = a_i + n$. Example 3.13 conforms with this definition. The constructors are well-typed in CC because we have the equation

$$
D = (T^1 \to \bot) \to \ldots \to (T^n \to \bot) \to \bot.
$$

Notation 3.15 *Let D be a data type and $\mathbf{D}$ its representation as a type in Continuation Calculus. If $d : D$ (so d is a data type element of D), we denote by $\langle d \rangle : \mathbf{D}$ the encoding of d as a term in CC. (So $\langle d \rangle$ is defined in the canonical way using the constructors of Definition 3.14.)*

Convention 3.16 *Unless otherwise specified, D_j^i is the type of the jth argument of the ith constructor of data type D. In the case of $\mathbf{List}_A$: $D_1^2 = A$, $D_2^2 = \mathbf{List}_A$.*

We often give the typing of terms via a derivation rule.

$$
\frac{x_1 : D_1^i \quad \ldots \quad x_{a_i} : D_{a_i}^i \quad c_1 : T^1 \to \bot \quad \ldots \quad c_n : T^n \to \bot}{\mathbf{Cons}_i^D.x_1. \ldots .x_{a_i}.c_1. \ldots .c_n : \bot}
$$

3.2   Iteration schemes

In this section we give iteration schemes for continuation calculus that provides general mechanisms for defining functions by recursion. An iteration scheme defines recursive functions in a general way,

ensuring well-definedness and termination for these functions. In CC we have a call-by-name and a call-by-value variant of the iteration scheme.

The simplest and most well-know form of iteration is over $\mathbb{N}$: Given $b : B$, $f : B \to B$, the function $\mathbf{It}(b, f) : \mathbb{N} \to B$ *defined by iteration from b and f*, is given by

$$\mathbf{It}(b, f)(n) = \begin{cases} b & \text{if } n = 0 \\ f(\mathbf{It}(b, f)(m)) & \text{if } n = m + 1 \end{cases}$$

An iterator $\mathbf{It}_D$ for a general data-type D (following the general scheme for first order data-types in Definition 3.11) to some type B has the following type:

$$\frac{f_1 : T^1[B/D] \to B \quad \ldots \quad f_n : T^n[B/D] \to B}{\mathbf{It}_D f_1 \ldots f_n : D \to B}$$

with

$$\mathbf{It}_D f_1 \ldots f_n(\mathbf{C}_i v_1 \ldots v_{a_i}) = f_i V_1 \ldots V_{a_i},$$

where $V_j = \mathbf{It}_D f_1 \ldots f_n v_j$ if $v_j : D$ and $V_j = v_j$ otherwise.

This is not yet the correct type for an iteration scheme in CC. We do not yet have any continuations as parameters. We will provide separate CBN and CBV iteration schemes below.

3.3 Call-by-name iterators

For a call-by-name iterator for a data-type D, we also have to consider the return data-type B. CBV calculates the entire return value, but for CBN it is enough to return the proper continuations with the proper parameters after calculating only one step in the recursion. So the CBN-iterator also passes around the continuations to the resulting values. If result type B has m constructors, then the iterator $\mathbf{ItCBN}_{D \to B}$ also needs m continuations as arguments. This differs from a call-by-value iterator, where we only have one continuation.

Let in the following, D be a data-type with n constructors and B a data-type with m constructors.

Definition 3.17 *We define the* call-by-name iterator for type D to type B *as follows. We first give the types of the new names. We abbreviate* $f_1 \ldots f_n$ *to* $\vec{f}$, $c_1 \ldots c_m$ *to* $\vec{c}$ *and* $x_1 \ldots x_{a_i}$ *to* $\vec{x}$.

$$\frac{f_1 : T^1[B/D] \to B \ldots f_n : T^n[B/D] \to B \quad c_1 : B^1 \ldots c_m : B^m \quad x : D}{\mathbf{ItCBN}_{D \to B}.\vec{f}.x.\vec{c} : \bot}$$

$$\frac{f_1 : T^1[B/D] \to B \ldots f_n : T^n[B/D] \to B \quad c_1 : B^1 \ldots c_m : B^m \quad x_1 : D_1^i \ldots x_{a_i}^i : D_{a_i}^i}{\mathbf{ItCBN}^i_{D \to B}.\vec{f}.\vec{c}.x_1 \ldots x_{a_i} : \bot}$$

The program rules are

$$\mathbf{ItCBN}_{D \to B}.\vec{f}.x.\vec{c} \longrightarrow x.(\mathbf{ItCBN}^1_{D \to B}.\vec{f}.\vec{c}). \ldots .(\mathbf{ItCBN}^n_{D \to B}.\vec{f}.\vec{c}),$$

and for $i \in [1 \ldots n]$:

$$\mathbf{ItCBN}^i_{D \to B}.\vec{f}.\vec{c}.\vec{x} \longrightarrow f_i.b(x_1). \ldots .b(x_{a_i}).\vec{c}$$

$$with\ b(x) = \begin{cases} \mathbf{ItCBN}_{D \to B}.\vec{f}.x & if\ x : D \\ x & otherwise \end{cases}$$

In Section 4, we give in Example 4.1 the call-by-name iterator for $\mathbf{List}_A$ to $\mathbf{Nat}$. The following can easily be checked. (See Definition 3.7 for the formal definition of well-typed rules.)

Lemma 3.18 *The rules given in Definition 3.17 are well-typed.*

3.4 Call-by-value iterators

Call-by-value iterators differ from their call-by-name cousins in the sense that the result of the computation is 'normalized' or fully evaluated at the end of the computation.

Definition 3.19 *We define the* call-by-value iterator for a type D to B as follows. (We abbreviate $f_1 \ldots f_n$ to $\vec{f}$.)

$$\frac{f_1 : T^1[B/D] \to \neg\neg B \ldots f_n : T^n[B/D] \to \neg\neg B \quad c : \neg B \quad d : D}{\mathbf{ItCBV}_{D \to B}\,\vec{f}\,c\,d : \bot}$$

and for $i \in [1,n]$ and $j \in [1,a_i]$, under the same typing hypotheses for $\vec{f}$ and c:

$$\frac{x_j : D^i_j \ldots x_{a_i} : D^i_{a_i} \quad r_1 : D^i_1[B/D] \ldots r_{j-1} : D^i_{j_1}[B/D]}{\mathbf{ItCBV}^{i,j}_{D \to B}.\vec{f}.c.x_j \ldots x_{a_i}.r_1 \ldots r_{j-1} : \bot}$$

The program rules are

$$\mathbf{ItCBV}_{D \to B}.\vec{f}.c.x \quad \longrightarrow \quad x.(\mathbf{ItCBV}^{1,1}_{D \to B}.\vec{f}.c).\ldots.(\mathbf{ItCBV}^{n,1}_{D \to B}.\vec{f}.c)$$

and for $i \in [1,n]$ and $j \in [1,a_i]$:

$$\mathbf{ItCBV}^{i,j}_{D \to B}.\vec{f}.c.x_j \ldots x_{a_i}.r_1 \ldots r_{j-1} \quad \longrightarrow \quad LHS$$

$$LHS = \begin{cases} \mathbf{ItCBV}_{D \to B}.\vec{f}.(\mathbf{ItCBV}^{i,j+1}_{D \to B}.\vec{f}.c.x_{j+1} \ldots x_{a_i}.r_1 \ldots r_{j-1}).x_j & if\ x_j : D \\ \mathbf{ItCBV}^{i,j+1}_{D \to B}.\vec{f}.c.x_{j+1} \ldots x_{a_i}.r_1 \ldots r_{j-1}.x_j & otherwise \end{cases}$$

$$\mathbf{ItCBV}^{i,a_i+1}_{D \to B}.\vec{f}.c.r_1 \ldots r_{a_i} \quad \longrightarrow \quad f_i.r_1 \ldots r_{a_i}.c$$

The technical subtlety in the call-by-value reduction rule lies in the fact that, in case data-type D has a constructor with more than one recursive sub-term (e.g. in the case of binary trees, where we have 'join', taking two sub-trees), we have to evaluate *all recursive sub-terms*. The reduction rule makes sure that we do that and reduce to a complete value before calling the function. The following lemma helps in better understanding the terms $\mathbf{ItCBV}^{i,j}_{D \to B}.\vec{f}.c.$ in Definition 3.19.

Lemma 3.20 *For $j \in [1 \ldots a_i]$, given $x_j : D^i_j \ldots x_{a_i} : D^i_{a_i}$ and $r_1 : D^i_1[B/D] \ldots r_{j-1} : D^i_{j_1}[B/D]$, the reduct of $\mathbf{ItCBV}^{i,j}_{D \to B}.\vec{f}.c.x_j \ldots x_{a_i}.r_1 \ldots r_{j-1}$ is of type $\bot$.*

Proof For $j = a_i$, the result is immediate, for other j, the result follows from the result for $j+1$, making a case distinction between $D^i_j[B/D] = D^i_j$ or $D^i_j[B/D] = B$. $\square$

The following now easily follows.

Lemma 3.21 *The rules given in Definition 3.19 are well-typed.*

Example 3.22 *For the iterator from* **Nat** *to* **Nat***, this amounts to the following*

$$\mathbf{ItCBV_{Nat\to Nat}}.f_1.f_2.c.x \longrightarrow x.(\mathbf{ItCBV_{Nat\to Nat}^{Zero,1}}.f_1.f_2.c).(\mathbf{ItCBV_{Nat\to Nat}^{Succ,1}}.f_1.f_2.c)$$

$$\mathbf{ItCBV_{Nat\to Nat}^{Zero,1}}.f_1.f_2.c \longrightarrow f_1.c$$

$$\mathbf{ItCBV_{Nat\to Nat}^{Succ,1}}.f_1.f_2.c.x_1 \longrightarrow \mathbf{ItCBV_{Nat\to Nat}}.f_1.f_2.(\mathbf{ItCBV_{Nat\to Nat}^{Succ,2}}.f_1.f_2.c).x_1$$

$$\mathbf{ItCBV_{Nat\to Nat}^{Succ,2}}.f_1.f_2.c.r_1 \longrightarrow f_2.r_1.c$$

This can be compressed a bit if we replace $\mathbf{ItCBV_{Nat\to Nat}^{Zero,1}}.f_1.f_2.c$ *by* $f_1.c$.

Another simplification that we can do is to replace some auxiliary names that are introduced in the iteration scheme by a λ-term. For example we can replace $\mathbf{ItCBV_{Nat\to Nat}^{Succ,2}}$ *by the new 'name'* $(\lambda f_1, f_2, c, r_1 \mapsto f_2.r_1.c)$. *The convention for such a name is that*

$$(\lambda f_1, f_2, c, r_1 \mapsto f_2.r_1.c).f_1.f_2.c.r_1 \longrightarrow f_2.r_1.c.$$

So, the arity of the new name is the number of arguments of the λ and its program rule is given by the body. Now the rules for $\mathbf{ItCBV_{Nat\to Nat}}$ *simplify to*

$$\mathbf{ItCBV_{Nat\to Nat}}.f_1.f_2.c.x \longrightarrow x.(f_1.c).(\mathbf{ItCBV_{Nat\to Nat}^{Succ,1}}.f_1.f_2.c)$$

$$\mathbf{ItCBV_{Nat\to Nat}^{Succ,1}}.f_1.f_2.c.x_1 \longrightarrow \mathbf{ItCBV_{Nat\to Nat}}.f_1.f_2.((\lambda f_1, f_2, c, r_1 \mapsto f_2.r_1.c).f_1.f_2.c).x_1$$

In Section 4, we show more examples, notably in Example 4.2 we give the call-by-value iterator for **List**$_A$ and we show how to program the 'length' function with it.

3.5 Rules for programming with data types in CC

Starting from the constructors for first order data types and the call-by-name and call-by-value iterators we can program new functions from existing ones. However, due to the fact that we are in CC and not in λ-calculus, we need some additional 'glue' to make flexible use of the iteration scheme to define functions.

Example 3.23 *Given* $\mathbf{ItCBV_{Nat\to Nat}}$ *we can define* **AddCBV** *as follows.*

$$\mathbf{AddCBV}.m.n.c := \mathbf{ItCBV_{Nat\to Nat}}.(F_1.m).F_2.c.n$$

where F_1 and F_2 are defined by

$$F_1.x.c \longrightarrow c.x$$
$$F_2.x.c \longrightarrow c.(\mathbf{Succ}.x)$$

So, we need 2 auxiliary functions to define **AddCBV** *in terms of* $\mathbf{ItCBV_{Nat\to Nat}}$. *In terms of Example 3.23, we need the names* $(\lambda x, c \mapsto c.x)$, *which is F_1 and* $(\lambda x, c \mapsto c.(\mathbf{Succ}.x))$, *which is F_2.*

The example shows that, to really profit from the expressivity of the iteration schemes, we must allow the addition of 'simple' functions. These are functions that have a non-circular definition.

Definition 3.24 *A non-circular program rule is a rule of the form*

$$n.x_1.\ldots.x_k \longrightarrow q,$$

where the names occurring in q are restricted to the constructors (Definition 3.14) and the iterators (CBN, Definition 3.17 and CBV, Definition 3.19).

We define $\mathbb{P}$ as the set of program rules that contains constructors for all data types (Definition 3.14), iterators for all data types (CBN, Definition 3.17 and CBV, 3.19) and arbitrary many non-circular rules.

So, $\mathbb{P}$ is an "open set": it contains constructors and iterators for all (infinitely many) data-types that we can define, and it includes arbitrary many "non-circular rules" that can be added when desired. This is needed to *really* define functions using the iteration schemes.

3.6   Translating between call-by-name and call-by-value

We can mediate between the call-by-name and the call-by-value representations of data by defining a function **StoreNat** : **Nat** $\to \neg\neg$**Nat** and a function **UnstoreNat** : $\neg\neg$**Nat** $\to$ **Nat**. The function **StoreNat** act as a *storage operator* in the sense of Krivine ([6]) in the sense that for t : **Nat** with $t \approx \langle n \rangle$ and c : **Nat** $\to \bot$,

$$\textbf{StoreNat}.t.c \longrightarrow\!\!\!\!\rightarrow c.\langle n \rangle.$$

So, **StoreNat** first evaluates the argument t of type **Nat** completely before passing it on to the continuation c. The term **StoreNat**.t.c can be defined as **AddCBV**.t.**Zero**.c, but we can also define it directly by

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

It is easy to verify that **StoreNat** : **Nat** $\to \neg\neg$**Nat**. (Note that $\textbf{A},\textbf{B} : \neg$**Nat** $\to \neg$**Nat**.)

In the reverse direction, we have **UnstoreNat** : $\neg\neg$**Nat** $\to$ **Nat**, defined by, given $f : \neg\neg$**Nat**,

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

Then **UnstoreNat** : $\neg\neg$**Nat** $\to$ **Nat**. (Note that **UseNat** : $\bot \to \neg$**Nat** $\to \neg$**Nat**.)

Lemma 3.25 *For all t : **Nat** and $n \in \mathbb{N}$ with $t \approx \langle n \rangle$, $\textbf{StoreNat}.t.c \longrightarrow\!\!\!\!\rightarrow c.\langle n \rangle$.*
For all $n \in \mathbb{N}$, $\textbf{UnstoreNat}.(\textbf{StoreNat}.\langle n \rangle) \approx \langle n \rangle$.

Proof For the first, we note that, if $t \approx \langle n \rangle$, then: (i) in case $n = 0$, $t.z.s \longrightarrow z$; (ii) in case $n = m + 1$, $t.z.s \longrightarrow s.q$ for some q with $q \approx \langle m \rangle$. Then we prove the following by induction on n (for all p): **StoreNat**. $\langle n \rangle .(\mathbf{B}^p.r) \longrightarrow r.(\mathbf{Succ}^{n+p}.\mathbf{Zero})$.

For the second, we prove **UnstoreNat**.(**StoreNat**. $\langle n \rangle$).$z.s =_P \langle n \rangle .z.s$, which is sufficient by Corollary 3.10. We compute:

$$
\begin{aligned}
\mathbf{UnstoreNat}.(\mathbf{StoreNat}. \langle n \rangle).z.s &\longrightarrow \mathbf{StoreNat}. \langle n \rangle .(\mathbf{UseNat}.z.s) \\
&\longrightarrow \mathbf{UseNat}.z.s. \langle n \rangle \\
&\longrightarrow \langle n \rangle .z.s
\end{aligned}
$$

Thus, **UnstoreNat**.(**StoreNat**. $\langle n \rangle$).$z.s =_P \langle n \rangle .z.s$ which was what we had to prove. $\square$

The map **StoreNat** can be seen as adding a double negation, whereas **UnstoreNat** can be seen as a classical double negation law, **UnstoreNat** : $\neg\neg\mathbf{Nat} \to \mathbf{Nat}$. Note that the fact that $\neg\neg\mathbf{Nat} \to \mathbf{Nat}$ is inhabited is not a surprise, because **Nat** is a negative type (ending in $\to \bot$. The precise connection with classical logic remains to be studied.

The storage and 'unstorage' operators can most likely also be defined for other data types.

More interesting to study further is the fact that we can combine call-by-name and call-by-value functions. We detail this for natural numbers.

Example 3.26 *If we have* $f_1 : \mathbf{Nat}$ *and* $f_2 : \mathbf{Nat} \to \mathbf{Nat}$, $c_1 : \bot$, $c_2 : \mathbf{Nat} \to \bot$ *and* $n : \mathbf{Nat}$, *then*

$$\mathbf{ItCBN_{Nat \to Nat}}.f_1.f_2.n.c_1.c_2 : \bot$$

gives a call-by-name iteration. However, one can also first define $\hat{f}_1 : \neg\neg\mathbf{Nat}$, $\hat{f}_2 : \mathbf{Nat} \to \neg\neg\mathbf{Nat}$ *and* $\widehat{(c_1, c_2)} : \mathbf{Nat} \to \bot$ *by*

$$
\begin{aligned}
\hat{f}_1.c &\longrightarrow c.f_1 \\
\hat{f}_2.n.c &\longrightarrow c.(f_2.n) \\
\widehat{(c_1, c_2)}.n &\longrightarrow n.c_1.c_2
\end{aligned}
$$

Then, for $n : \mathbf{Nat}$, *we have*

$$\mathbf{ItCBV_{Nat \to Nat}}.\hat{f}_1.\hat{f}_2.\widehat{(c_1, c_2)}.n : \bot$$

which gives call-by-value iteration. So, using this transformation (from f_1 *to* $\hat{f}_1$ *etc.) one can use the call-by-name functions to compute call-by-value.*

4 Examples of iterators and programs

Example 4.1 *This is the call-by-name iterator for* $\mathbf{List}_A$ *to* $\mathbf{Nat}$:

$$
\begin{aligned}
\mathbf{ItCBN_{List_A \to Nat}}.f_1.f_2.x.c_1.c_2 \quad &\longrightarrow \\
x.(\mathbf{ItCBN^{Nil}_{List_A \to Nat}}&.f_1.f_2.c_1.c_2).(\mathbf{ItCBN^{Cons}_{List_A \to Nat}}.f_1.f_2.c_1.c_2) \\
\mathbf{ItCBN^{Nil}_{List_A \to Nat}}.f_1.f_2.c_1.c_2 \quad &\longrightarrow \quad f_1.c_1.c_2 \\
\mathbf{ItCBN^{Cons}_{List_A \to Nat}}.f_1.f_2.c_1.c_2.x_1.x_2 \quad &\longrightarrow \quad f_2.x_1.(\mathbf{ItCBN_{List_A \to Nat}}.f_1.f_2.x_2).c_1.c_2
\end{aligned}
$$

Example 4.2 *The scheme of Definition 3.19 yields the following call-by-value iterator for* $\mathbf{List}_A$ *to B:*

$$\mathbf{ItCBV}_{\mathbf{List}_A\to B}.f_1.f_2.c.x \quad\longrightarrow\quad x.(\mathbf{ItCBV}^{\mathsf{Nil},1}_{\mathbf{List}_A\to B}.f_1.f_2.c).(\mathbf{ItCBV}^{\mathsf{Cons},1}_{\mathbf{List}_A\to B}.f_1.f_2.c)$$

$$\mathbf{ItCBV}^{\mathsf{Nil},1}_{\mathbf{List}_A\to B}.f_1.f_2.c \quad\longrightarrow\quad f_1.c$$

$$\mathbf{ItCBV}^{\mathsf{Cons},1}_{\mathbf{List}_A\to B}.f_1.f_2.c.x_1.x_2 \quad\longrightarrow\quad \mathbf{ItCBV}^{\mathsf{Cons},2}_{\mathbf{List}_A\to B}.f_1.f_2.c.x_2.x_1$$

$$\mathbf{ItCBV}^{\mathsf{Cons},2}_{\mathbf{List}_A\to B}.f_1.f_2.c.x_2.r_1 \quad\longrightarrow\quad \mathbf{ItCBV}_{\mathbf{List}_A\to B}.f_1.f_2.(\mathbf{ItCBV}^{\mathsf{Cons},3}_{\mathbf{List}_A\to B}.f_1.f_2.c.r_1).x_2$$

$$\mathbf{ItCBV}^{\mathsf{Cons},3}_{\mathbf{List}_A\to B}.f_1.f_2.c.r_1.r_2 \quad\longrightarrow\quad f_2.r_1.r_2.c$$

We note that the number of program rules we need is highly dependent on the arity of the constructors **Nil** *and* **Cons**. *Since* **Nil** *has no parameters, one rule is enough to define the operation on* **Nil**. **Cons** *on the other hand has two parameters. Because of this we get three program rules: one for evaluating each parameter of the constructor and one general rule that redirects every parameter to the corresponding program rule.*

We now show the use of the iterators by providing the implementation of the function **Length**. We use the iterators for $\mathbf{List}_A$ from Example 4.1 and Example 4.2.

$$\mathbf{LengthCBN}.x.c_1.c_2 \quad\longrightarrow\quad \mathbf{ItCBN}_{\mathbf{List}_A\to\mathbf{Nat}}.\mathbf{LengthCBN}^1.\mathbf{LengthCBN}^2.x.c_1.c_2$$

$$\mathbf{LengthCBN}^1.c_1.c_2 \quad\longrightarrow\quad \mathbf{Zero}.c_1.c_2$$

$$\mathbf{LengthCBN}^2.x.n.c_1.c_2 \quad\longrightarrow\quad \mathbf{Succ}.n.c_1.c_2$$

$$\mathbf{LengthCBV}.x.c \quad\longrightarrow\quad \mathbf{ItCBV}_{\mathbf{List}_A\to\mathbf{Nat}}.\mathbf{LengthCBV}^1.\mathbf{LengthCBV}^2.c.x$$

$$\mathbf{LengthCBV}^1.c \quad\longrightarrow\quad c.\mathbf{Zero}$$

$$\mathbf{LengthCBV}^2.x.n.c \quad\longrightarrow\quad c.(\mathbf{Succ}.n)$$

We prove for $\mathbb{N}$ that the two iterator schemes (CBN and CBV) indeed compute the desired results. We expect that this proof can easily be extended to prove the semantics of our iteration schemes for any first-order algebraic data-type. We leave this for future work.

We assume D to be a data type which has a representation in Continuation Calculus, $\mathbf{D}$, with a representation such that $\langle d\rangle : \mathbf{D}$, for $d : D$. We now define what it means that a function over a data-type is represented in CC.

Definition 4.3 *We say that* $f_1 : \neg\neg\mathbf{D}$ *CBV-represents* $d : D$ *and* $f_2 : \mathbf{D}\to\neg\neg\mathbf{D}$ *CBV-represents* $F : D\to D$, *if for all* $c : \mathbf{D}\to\perp$ *and* $n\in\mathbb{N}$ *we have*

$$f_1.c \longrightarrow\!\!\!\twoheadrightarrow c.\langle d\rangle \tag{1}$$

$$f_2.\langle n\rangle.c \longrightarrow\!\!\!\twoheadrightarrow c.\langle F(n)\rangle \tag{2}$$

The following Theorem states the semantic correctness of $\mathbf{ItCBV}_{\mathbf{Nat}\to D}$ in CC. The proof can be found in the Appendix section 6.

Theorem 4.4 *If* f_1 *CBV-represents* $d : D$ *and* f_2 *CBV-represents* $F : D\to D$, *then* $\mathbf{ItCBV}_{\mathbf{Nat}\to D}.f_1.f_2$ *CBV-represents* $\mathbf{It}(d,F)$, *that is: for all* $c : \mathbf{D}\to\perp$ *and all* $n\in\mathbb{N}$ *we have*

$$\mathbf{ItCBV}_{\mathbf{Nat}\to D}.f_1.f_2.c.\langle n\rangle \longrightarrow\!\!\!\twoheadrightarrow c.\langle(\mathbf{It}(d,F))(n)\rangle$$

Definition 4.5 *If D has m constructors, we say that $f_1 : \mathbf{D}$ CBN-represents $d : D$ and $f_2 : \mathbf{D} \to \mathbf{D}$ CBN-represents $F : D \to D$, if for all $c_i : \mathbf{D}^i$ and $n \in \mathbb{N}$ we have (writing $\vec{c} = c_1 \ldots c_m$):*

$$f_1.\vec{c} \approx \langle d \rangle .\vec{c} \tag{1}$$

$$f_2.\langle n \rangle .\vec{c} \approx \langle F(n) \rangle .\vec{c} \tag{2}$$

The following Theorem states the semantic correctness of $\mathbf{ItCBN_{Nat \to D}}$ in CC. The proof can be found in the Appendix section 6.

Theorem 4.6 *If D has m constructors, f_1 CBN-represents $d : D$ and f_2 CBN-represents $F : D \to D$, then $\mathbf{ItCBN_{Nat \to D}}.f_1.f_2$ CBN-represents $\mathbf{It}(d,F)$. That is, for all $c_i : \mathbf{D}^i$ and all $n \in \mathbb{N}$:*

$$\mathbf{ItCBN_{Nat \to D}}.f_1.f_2.\langle n \rangle .\vec{c} \approx \langle (\mathbf{It}(d,F))(n) \rangle .\vec{c}$$

5 Future Work and Conclusions

As future work, we want to understand better the relation with classical logic, which we have briefly indicated in Section 4. Here we have also defined storage (and unstorage) operators, which we would like to define in general for all data types. The possibility to combine call-by-value and call-by-name in a flexible way, which is directed by the types, is an interesting feature, which needs further study. The fact that computation is completely deterministic and that the function definition of f itself determines whether f is cbv or cbn, makes the this combining of cbv and cbn very perspicuous.

The continuations in this paper are limited, and do not include delimited continuations. (More examples using continuations van be found in [2].) It would be interesting to see if delimited continuations can be added.

In a forthcoming paper [4] we prove the termination of all CC terms written using the program rules of $\mathbb{P}$. This is done by translating CC with these rules to a typed λ-calculus with (cbv and cbn) iterators. We wish to further study the precise translations and connections between CC and (typed) λ-calculus.

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] Herman Geuvers. A typed λ-calculus with cbn and cbv iterators. to appear, 2014.

[5] 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.

[6] Jean-Louis Krivine. Classical logic, storage operators and second-order lambda-calculus. *Ann. Pure Appl. Logic*, 68(1):53–78, 1994.

[7] A. Miquel. Classical realizability with forcing and the axiom of countable choice. `http://perso.ens-lyon.fr/alexandre.miquel/habilitation/forcing.pdf`, 2009.

6 Appendix

Proof of Theorem4.4

Proof We recall the definition of $\mathbf{ItCBV_{Nat \to B}}B$:

$$\mathbf{ItCBV_{Nat \to B}}.f_1.f_2.c.x \longrightarrow x.(\mathbf{ItCBV_{Nat \to B}^{Zero,1}}.f_1.f_2.c).(\mathbf{ItCBV_{Nat \to B}^{Succ,1}}.f_1.f_2.c) \tag{3a}$$

$$\mathbf{ItCBV_{Nat \to B}^{Zero,1}}.f_1.f_2.c \longrightarrow f_1.c \tag{3b}$$

$$\mathbf{ItCBV_{Nat \to B}^{Succ,1}}.f_1.f_2.c.x_1 \longrightarrow \mathbf{ItCBV_{Nat \to B}}.f_1.f_2.(\mathbf{ItCBV_{Nat \to B}^{Succ,2}}.f_1.f_2.c).x_1 \tag{3c}$$

$$\mathbf{ItCBV_{Nat \to B}^{Succ,2}}.f_1.f_2.c.r_1 \longrightarrow f_2.r_1.c \tag{3d}$$

By induction on n we prove that for all n, $P(n)$ holds, with

$$P(n) := \mathbf{ItCBV_{Nat \to B}}.f_1.f_2.c.\langle n \rangle \longrightarrow\!\!\!\rightarrow c.\langle (\mathbf{It}(x,F))(n) \rangle$$

$P(0)$ holds, because:

$$
\begin{aligned}
\mathbf{ItCBV_{Nat \to B}}.f_1.f_2.c.\langle 0 \rangle \quad &\xrightarrow{(3a)} \quad \langle 0 \rangle.(\mathbf{ItCBV_{Nat \to B}^{Zero,1}}.f_1.f_2.c).(\mathbf{ItCBV_{Nat \to B}^{Succ,1}}.f_1.f_2.c) \\
&= \quad \mathbf{Zero}.(\mathbf{ItCBV_{Nat \to B}^{Zero,1}}.f_1.f_2.c).(\mathbf{ItCBV_{Nat \to B}^{Succ,1}}.f_1.f_2.c) \\
&\xrightarrow{Zero} \quad \mathbf{ItCBV_{Nat \to B}^{Zero,1}}.f_1.f_2.c \\
&\xrightarrow{(3b)} \quad f_1.c \\
&\xrightarrow{(1)} \quad c.\langle x \rangle \\
&= \quad c.\langle (\mathbf{It}(x,F))(0) \rangle
\end{aligned}
$$

Assume $P(n)$ holds.

$P(n+1)$ holds, because:

$$
\begin{aligned}
\mathbf{ItCBV_{Nat \to B}}.f_1.f_2.c.\langle n+1 \rangle \quad &\xrightarrow{(3a)} \quad \langle n+1 \rangle.(\mathbf{ItCBV_{Nat \to B}^{Zero,1}}.f_1.f_2.c).(\mathbf{ItCBV_{Nat \to B}^{Succ,1}}.f_1.f_2.c) \\
&= \quad \mathbf{Succ}.\langle n \rangle.(\mathbf{ItCBV_{Nat \to B}^{Zero,1}}.f_1.f_2.c).(\mathbf{ItCBV_{Nat \to B}^{Succ,1}}.f_1.f_2.c) \\
&\xrightarrow{Succ} \quad \mathbf{ItCBV_{Nat \to B}^{Succ,1}}.f_1.f_2.c.\langle n \rangle \\
&\xrightarrow{(3c)} \quad \mathbf{ItCBV_{Nat \to B}}.f_1.f_2.(\mathbf{ItCBV_{Nat \to B}^{Succ,2}}.f_1.f_2.c).\langle n \rangle \\
&\xrightarrow{P(n)} \quad \mathbf{ItCBV_{Nat \to B}^{Succ,2}}.f_1.f_2.c.\langle (\mathbf{It}(x,F))(n) \rangle \\
&\xrightarrow{(3d)} \quad f_2.\langle (\mathbf{It}(x,F))(n) \rangle.c \\
&\xrightarrow{(2)} \quad c.\langle F((\mathbf{It}(x,F))(n)) \rangle \\
&= \quad c.\langle F(F^n(x)) \rangle \\
&= \quad c.\langle F^{n+1}(x) \rangle \\
&= \quad c.\langle (\mathbf{It}(x,F))(n+1) \rangle
\end{aligned}
$$

Proof of Theorem4.6

Proof We recall the definition of $\textbf{ItCBN}_{\textbf{Nat} \to B}$:

$$\textbf{ItCBN}_{\textbf{Nat} \to B}.f_1.f_2.x.\vec{c} \longrightarrow x.(\textbf{ItCBN}^{\mathsf{Zero}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c}).(\textbf{ItCBN}^{\mathsf{Succ}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c}) \tag{3a}$$

$$\textbf{ItCBN}^{\mathsf{Zero}}_{\textbf{Nat} \to B}.f_1.f_2.c \longrightarrow f_1.\vec{c} \tag{3b}$$

$$\textbf{ItCBN}^{\mathsf{Succ}}_{\textbf{Nat} \to B}.f_1.f_2.c.x_1 \longrightarrow f_2.(\textbf{ItCBN}_{\textbf{Nat} \to B}.f_1.f_2.x_1).\vec{c} \tag{3c}$$

By induction on n we prove that for all n, $P(n)$ holds, with

$$P(n) \; := \; \textbf{ItCBN}_{\textbf{Nat} \to B}.f_1.f_2.\langle n \rangle.\vec{c} \approx \langle (\textbf{It}(x, F))(n) \rangle.\vec{c}$$

$P(0)$ holds, because:

$$
\begin{aligned}
\textbf{ItCBN}_{\textbf{Nat} \to B}.f_1.f_2.\langle 0 \rangle.\vec{c} \quad &\xrightarrow{(3a)} \quad \langle 0 \rangle.(\textbf{ItCBN}^{\mathsf{Zero}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c}).(\textbf{ItCBN}^{\mathsf{Succ}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c}) \\
&= \quad \textbf{Zero}.(\textbf{ItCBN}^{\mathsf{Zero}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c}).(\textbf{ItCBN}^{\mathsf{Succ}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c}) \\
&\xrightarrow{\mathsf{Zero}} \quad \textbf{ItCBN}^{\mathsf{Zero}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c} \\
&\xrightarrow{(3b)} \quad f_1.\vec{c} \\
&\xrightarrow{(1)} \quad \langle x \rangle.\vec{c} \\
&= \quad \langle (\textbf{It}(x, F))(0) \rangle.\vec{c}
\end{aligned}
$$

Assume $P(n)$ holds.

$P(n+1)$ holds, because:

$$
\begin{aligned}
\textbf{ItCBN}_{\textbf{Nat} \to B}.f_1.f_2.\langle n+1 \rangle.\vec{c} \quad &\xrightarrow{(3a)} \quad \langle n+1 \rangle.(\textbf{ItCBN}^{\mathsf{Zero}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c}).(\textbf{ItCBN}^{\mathsf{Succ}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c}) \\
&= \quad \textbf{Succ}.\langle n \rangle.(\textbf{ItCBN}^{\mathsf{Zero}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c}).(\textbf{ItCBN}^{\mathsf{Succ}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c}) \\
&\xrightarrow{\mathsf{Succ}} \quad \textbf{ItCBN}^{\mathsf{Succ}}_{\textbf{Nat} \to B}.f_1.f_2.\vec{c}.\langle n \rangle \\
&\xrightarrow{(3c)} \quad f_2.(\textbf{ItCBN}_{\textbf{Nat} \to B}.f_1.f_2.\langle n \rangle).\vec{c} \\
&\overset{P(n),[2.3]}{\approx} \quad f_2.\langle (\textbf{It}(x, F))(n) \rangle.\vec{c} \\
&\overset{(2)}{\approx} \quad \langle F((\textbf{It}(x, F))(n)) \rangle.\vec{c} \\
&= \quad \langle F(F^n(x)) \rangle.\vec{c} \\
&= \quad \langle F^{n+1}(x) \rangle.\vec{c} \\
&= \quad \langle (\textbf{It}(x, F))(n+1) \rangle.\vec{c}
\end{aligned}
$$

In the proof, we say that $f_2.(\textbf{ItCBN}_{\textbf{Nat} \to B}.f_1.f_2.\langle n \rangle).\vec{c} \approx f_2.\langle (\textbf{It}(x, F))(n) \rangle.\vec{c}$. This may may not be immediately obvious, as the sub-term $\textbf{ItCBN}_{\textbf{Nat} \to B}.f_1.f_2.\langle n \rangle$ is incomplete. However, it is an immediate consequence of Lemma2.3: If M, N are terms of arity k, and $M.t_1.\ldots.t_k =_p N.t_1.\ldots.t_k$ for all $\vec{t}$, then $M \approx N$.