# Continuation calculus

Bram Geron
Eindhoven University of Technology
bgeron@gmail.com

Herman Geuvers
Radboud University Nijmegen
Eindhoven University of Technology
herman@cs.ru.nl

Programs with control are usually modeled using lambda calculus extended with control operators. Instead of modifying lambda calculus, we consider a different model of computation. We introduce continuation calculus, or CC, a deterministic model of computation that is evaluated using only head reduction, and argue that it is suitable for modeling programs with control. It is demonstrated how to define programs, specify them, and prove them correct. This is shown in detail by presenting in CC a list multiplication program that prematurely returns when it encounters a zero. The correctness proof includes termination of the program.

In continuation calculus we can model both call-by-name and call-by-value. In addition, call-by-name functions can be applied to call-by-value results, and conversely.

## 1 Introduction

Lambda calculus has historically been the foundation of choice for modeling programs in pure functional languages. To capture features that are not purely functional, there is an abundance of variations in syntax and semantics: lambda calculus can be extended with special operators: $\mathscr{A}$, $\mathscr{C}$, $\mathscr{F}^{+/-}_{+/-}$, #, and call/cc, to incorporate control [4, 6, 7, 5] or one can move to a calculus like $\lambda$ [10, 2] that allows the encoding of control-like operators. Also, one must choose between the call-by-value and call-by-name reduction orders for the calculus to correspond to the modeled language. If one wants to study these calculi, one usually applies one of many CPS translations [11, 3] which allow simulation of control operators in a system without them. There is also a close connection between proofs in classical logic and control operators, as was first pointed out by Griffin [8], who extended the Curry-Howard proofs-as-programs principle to include rules of classical logic. The $\lambda$ -calculus of [10, 1] is also based on the relation between classical logical rules and control-like constructions in the type theory.

In this paper, we introduce a different kind of calculus for formalizing functional programs: *continuation calculus*. It is deterministic and Turing complete, yet its operational semantics are minimal: there is only head reduction and no stack, environment, or context. Control is natural to express, without additional operators.

We present continuation calculus as an untyped system. The study of a typed version, and possibly the connections with the rules of classical logic, is for future research. In the present paper we want to introduce the system, show how to write programs in it and prove properties about these programs, and show how control aspects and call-by-value (CBV) and call-by-name (CBN) naturally fit into the system.

Continuation calculus looks a bit like term rewriting and a bit like $\lambda$-calculus, and it has ideas from both. A term in CC is of the shape

$$n.t_1.\cdots.t_k,$$

where $n$ is a *name* and the $t_i$ are themselves terms. The "dot" is a binary operator that associates to the left. Note that terms do not contain variables. A *program P* is a list of *program rules* of the form

$$n.x_1.\cdots.x_k \longrightarrow u$$

© Bram Geron, Herman Geuvers
This work is licensed under the
Creative Commons Attribution License.

where the $x_i$ are all different variables and $u$ is a term over variables $x_1 \ldots x_k$. This program rule is said to *define* $n$, and we make sure that in a program $P$ there is *at most one* definition of $n$. Here, CC already deviates from term rewriting, where one would have, for example:

$$\begin{aligned} Add(0,m) &\longrightarrow m \\ Add(S(n),m) &\longrightarrow S(Add(n,m)) \end{aligned}$$

These syntactic case distinctions, or *pattern matchings*, are not possible in CC.

The meaning of the program rule $n.x_1.\cdots.x_k \longrightarrow u$ is that a term $n.t_1.\cdots.t_k$ evaluates to $u[\vec{x} := \vec{t}]$: the variables $\vec{x}$ in $t$ are replaced by the respective terms $\vec{t}$. A peculiarity of CC is that one cannot evaluate "deep in a term": we do not evaluate inside any of the $t_i$ and if we have a term $n.t_1.\cdots.t_m$, where $m > k$, this term does not evaluate. (This will even turn out to be a "meaningless" term.)

To give a better idea of how CC works, we give the example of the natural numbers: how they are represented in CC and how one can program addition on them. A natural number is either 0, or $S(m)$ for $m$ a natural number. We shall have a name *Zero* and a name $S$. The number $m$ will be represented by $S.(\cdots.(S.Zero)\cdots)$, with $m$ times $S$. So the numbers 0 and 3 are represented by the terms *Zero* and $S.(S.(S.Zero))$.

The only way to extract information from a natural $m$ is to "transfer control" to that natural. Execution should continue in some code $c_1$ when $m = 0$, and execution should continue in different code $c_2$ when $m = S(p)$. This becomes possible by postulating the following rules for *Zero* and $S$:

$$\begin{aligned} Zero.z.s &\longrightarrow z \\ S.x.z.s &\longrightarrow s.x \end{aligned}$$

We will now implement call-by-value addition in CC on these natural numbers. The idea of CC is that a function application does not just produce an output value, but passes it to the next function, the continuation. So we are looking for a term *AddCBV* that behaves as follows:

$$AddCBV.\langle\!\langle m\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle.r \twoheadrightarrow r.\langle\!\langle m+p\rangle\!\rangle \tag{1}$$

for all $m, p, r$, where $\twoheadrightarrow$ is the multi-step evaluation, and $\langle\!\langle l\rangle\!\rangle$ are the terms that represent a natural number $l$. Term $r$ indicates where evaluation should continue after the computation of $\langle\!\langle m+p\rangle\!\rangle$.

Equation (1) is the *specification of AddCBV*. We will use the following algorithm:

$$\begin{aligned} 0+p &= p \\ S(m)+p &= m+S(p) \end{aligned}$$

To program *AddCBV*, we have to give a rule of the shape $AddCBV.x.y.r \longrightarrow t$. We need to make a case distinction on the first argument $x$. If $x = Zero$, then the result of the addition is $y$, so we pass control to $r.y$. If $x = S.u$, then control should eventually transfer to $r.(AddCBV.u.(S.y))$. Let us write down a first approximation of *AddCBV*:

$$AddCBV.x.y.r \longrightarrow x.(r.y).t$$

The term $t$ is yet to be determined. Now control transfers to $r.y$ when $x = Zero$, or to $t.u$ when $x = S.u$. From $t.u$, control should eventually transfer to $AddCBV.u.(S.y).r$. Let us write down a naive second approximation of *Add*, in which we introduce a helper name $B$.

$$\begin{aligned} AddCBV.x.y.r &\longrightarrow x.(r.y).B \\ B.u &\longrightarrow AddCBV.u.(S.y).r \end{aligned}$$

Unfortunately, the second line is not a valid rule: $y$ and $r$ are variables in the right-hand side of $B$, but do not occur in its left-hand side. We can fix this by replacing $B$ with $B.y.r$ in both rules.

$$\begin{aligned} AddCBV.x.y.r &\longrightarrow x.(r.y).(B.y.r) \\ B.y.r.u &\longrightarrow AddCBV.u.(S.y).r \end{aligned}$$

This is a general procedure for representing data types and functions over data in CC. We can now prove the correctness of *AddCBV* by showing (simultaneously by induction on $m$) that

$$\begin{aligned} AddCBV.\langle\!\langle m\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle.r &\twoheadrightarrow r.\langle\!\langle m+p\rangle\!\rangle \\ B.\langle\!\langle p\rangle\!\rangle.r.\langle\!\langle m'\rangle\!\rangle &\twoheadrightarrow r.\langle\!\langle m'+p+1\rangle\!\rangle \end{aligned}$$

We formally define and characterize continuation calculus in the following sections. In Section 5, we define the meaning of $\langle\!\langle\cdot\rangle\!\rangle$, which allows us to give a specification for call-by-name addition, *AddCBN*:

$$AddCBN.\langle\!\langle m\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle \in \langle\!\langle m+p\rangle\!\rangle$$

This statement means that $AddCBN.\langle\!\langle m\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle$ is equivalent to and compatible with $S.(\cdots(S.Zero)\cdots)$, with $m+p$ times $S$. The precise meaning of this statement will be given in Definitions 32 and Remark 33.

The terms *AddCBV* and *AddCBN* are of a different kind. Nonetheless, we will see in Section 5.1 how call-by-value and call-by-name functions can be used together. We show additional examples with *FibCBV* and *FibCBN* in Section 5.1. Furthermore, we model and prove a program with call/cc in Sections 6 and 7.

The authors have made a program available to evaluate continuation calculus terms on `http://www.cs.ru.nl/~herman/ccalc/`. Evaluation traces of the examples are included.

## 2 Formalization

**Definition 1** (names)**.** There is an infinite set $\mathcal{N}$ of names. Concrete names are typically denoted as upper-case letters $(A,B,\ldots)$, or capitalized words $(True,False,And,\ldots)$; we refer to *any* name using $n$ and $m$.

*Interpretation.* Names are used by programs to refer to 'functionality', and will serve the role of constructors, function names, as well as labels within a function.

**Definition 2** (universe)**.** The set of terms $\mathcal{U}$ in continuation calculus is generated by:

$$\mathcal{U} ::= \mathcal{N} \mid \mathcal{U}.\mathcal{U}$$

where . (dot) is a binary constructor. The dot is neither associative nor commutative, and there shall be no overlap between names and dot-applications. We will often use $M,N,t,u$ to refer to terms. If we know that a term is a name, we often use $n,m$. We sometimes use lower-case words that describe its function, e.g. *abort*, or letters, e.g. $r$ for a 'return continuation'.

The dot is read left-associative: when we write $A.B.C$, we mean $(A.B).C$.

*Interpretation.* Terms by themselves do not denote any computation, nor do they have any value of themselves. We inspect value terms by 'dotting' other terms on them, and observing the reduction behavior. If for instance $b$ represents a boolean value, then $b.t.f$ reduces to $t$ if $b$ represents true; $b.t.f$ reduces to $f$ if $b$ represents false.

**Definition 3** (head, length)**.** All terms have a head, which is defined inductively:

$$\begin{aligned} \mathsf{head}(n\in\mathcal{N}) &= n \\ \mathsf{head}(a.b) &= \mathsf{head}(a). \end{aligned}$$

The head of a term is always a name.

The length of a term is determined by the number of dots traversed towards the head.

$$\begin{aligned}\mathsf{length}(n \in \mathcal{N}) &= 0\\ \mathsf{length}(a.b) &= 1 + \mathsf{length}(a).\end{aligned}$$

This definition corresponds to left-associativity: $\mathsf{length}(n.t_1.t_2.\cdots.t_k) = k$.

**Definition 4** (variables)**.** There is an infinite set $\mathcal{V}$ of variables. Terms are not variables, nor is the result of a dot application ever a variable.

Variables are used in CC rules as formal parameters to refer to terms. We will use lower-case letters or words, or $x, y, z$ to refer to variables.

Note that we use similar notations for both variables and terms. However, variables exist only in rules, so we expect no confusion.

**Definition 5** (rules)**.** Rules consist of a left-hand and a right-hand side, generated by:

$$\mathsf{LHS} ::= \mathcal{N} \mid \mathsf{LHS}.\mathcal{V} \qquad \text{where every variable occurs at most once}$$
$$\mathsf{RHS} ::= \mathcal{N} \mid \mathcal{V} \mid \mathsf{RHS}.\mathsf{RHS}$$

Therefore, any right-hand side without variables is a term in $\mathcal{U}$.

A combination of a left-hand and a right-hand side is a rule only when all variables in the right-hand side also occur in the left-hand side.

$$\mathsf{Rules} ::= \mathsf{LHS} \to \mathsf{RHS} \qquad \text{where all variables in } \mathsf{RHS} \text{ occur in } \mathsf{LHS}$$

A rule is said to *define* the name in its left-hand side; this name is also called the *head*. The *length* of a left-hand side is equal to the number of variables in it.

**Definition 6** (program)**.** A *program* is a finite set of rules, where no two rules define the same name. We denote a program by $P$.

$$\mathsf{Programs} = \{P \subseteq \mathsf{Rules} | P \text{ is finite and } \mathsf{head}(\cdot) \text{ is injective on the LHSes in } P)$$

The *domain* of a program is the set of names defined by its rules.

$$\mathsf{dom}(P) = \{\mathsf{head}(rule) \mid rule \in P\}$$

We will frequently extend programs: an *extension* of a program is a superset of that program.

**Definition 7** (evaluation)**.** A term can be evaluated under a program. Evaluation consists of zero or more sequential steps, which are all deterministic. For some terms and programs, evaluation never terminates.

We define the evaluation through the partial successor function $\mathsf{next}_P(\cdot) : \mathcal{U} \nrightarrow \mathcal{U}$. We define $\mathsf{next}_P(t)$ when $P$ defines $\mathsf{head}(t)$, and $\mathsf{length}(t)$ equals the length of the corresponding left-hand side.

$$\mathsf{next}_P(n.t_1.t_2.\cdots.t_k) = r[\vec{x} := \vec{t}] \qquad \text{when "}n.x_1.x_2.\cdots.x_k \longrightarrow r\text{"} \in P$$

It is allowed that $n = 0$:

$$\mathsf{next}_P(n) = r \qquad \text{when "}n \longrightarrow r\text{"} \in P$$

More informally, we write $M \to_P N$ when $\mathsf{next}_P(M) = N$. The reflexive and transitive closure of $\to_P$ will be denoted $\twoheadrightarrow_P$. When $M \twoheadrightarrow N$, then we call $N$ a *reduct* of $M$, and $M$ is said to be *defined*. When $\mathsf{next}_P(M)$ is not defined, we write that $M$ is *final*. Notation: $M \downarrow_P$. We also combine the notations: if $\mathsf{next}_P(M) = N$ and $\mathsf{next}_P(N)$ is undefined, we may write $M \to_P N \downarrow_P$. We will often leave the subscript $P$ implicit: $M \to N \downarrow$.

In Section 3, we divide the final terms in three groups: undefined terms, incomplete terms, and invalid terms. Thus, these are the three cases where $\mathsf{next}_P(M)$ is undefined.

**Definition 8** (termination)**.** A term $M$ is said to be *terminating* under a program $P$, notation $M \twoheadrightarrow\downarrow_P$, when it has a final reduct: $\exists N \in \mathcal{U} : M \twoheadrightarrow_P N \downarrow_P$. We often leave the subscript $P$ implicit.

## 3 Categorization of terms

A program divides all terms into four disjoint categories: undefined, incomplete, complete, and invalid. A term's evaluation behavior depends on its category, to which the term's *arity* is crucial.

**Definition 9.** The name $n$ has *arity* $k$ if $P$ contains a rule of the form $n.x_1.\cdots.x_k \longrightarrow q$.

A term $t$ has *arity* $k-i$ if it is of the form $n.q_1.\cdots.q_i$, where $n$ has arity $k$ ($k \geq i$).

**Definition 10.** Term $t$ is *defined in* $P$ if $\mathsf{head}(t) \in \mathsf{dom}(P)$, otherwise we say that $t$ is *undefined*.

Given a $t$ that is defined, we say that

- $t$ is *complete* if the arity of $t$ is 0
- $t$ is *incomplete* if the arity of $t$ is $j > 0$
- $t$ is *invalid* if is has no arity (that is, $t$ is of the form $n.q_1.\cdots.q_i$, where $n$ has arity $k < i$)

The four categories have distinct characteristics.

**Undefined terms.** Term $M$ is undefined iff $M.N$ is undefined. Extension of the program causes undefined terms to remain undefined or become incomplete, complete, or invalid.

*Interpretation.* Because variables are not part of a term in continuation calculus, we use undefined names instead for similar purposes, as exemplified by Theorem 12. This means that all CC terms are 'closed' in the lambda calculus sense.

The remaining three categories contain *defined terms*: terms with a head $\in \mathsf{dom}(P)$. Extension of the program does not change the category of defined terms.

**Incomplete terms.** If $M$ is incomplete, then $M.N$ can be incomplete or complete.

*Interpretation.* There are four important classes of incomplete terms.

- *Data terms (see Section 5).* If $d$ represents $c_k(v_1,\cdots,v_{n_k})$ of a data type $\mathscr{D}$ with $m$ constructors, then $\forall t_1 \ldots t_m : d.\vec{t} \twoheadrightarrow t_k.\vec{v}$. Examples:

$$\forall z,s : Zero.z.s \twoheadrightarrow z \qquad Zero \text{ represents } 0$$

$$\forall z,s : S.(S.(S.Zero)).z.s \twoheadrightarrow s.(S.(S.Zero)) \qquad S.(S.(S.Zero)) \text{ represents } S(S(S(0)))$$

- *Call-by-name function terms.* These are terms $f$ such that $f.v_1.\cdots.v_k$ is a data term $\in [\![\mathscr{D}]\!]$ for all $\vec{v}$ in the appropriate domain. Example using Figure 1:

$$\forall z,s : AddCBN.Zero.Zero.z.s \twoheadrightarrow z$$

$$\forall z,s : AddCBN.(S.Zero).(S.(S.Zero)).z.s \twoheadrightarrow s.(AddCBN.Zero.(S.(S.Zero)))$$

Recall that $AddCBN.(S.Zero).(S.(S.Zero))$ is a data term that represents 3. The second reduction shows that $1 +_{\text{CBN}} 2 = S(x)$, for some $x$ represented by $AddCBN.Zero.(S.(S.Zero))$.

- *Call-by-value function terms.* These are terms $f$ of arity $n+1$ such that for all $\vec{v}$ in a certain domain, $\forall r : f.v_1.\cdots.v_n.r \twoheadrightarrow r.t$ with data term $t$ depending only on $\vec{v}$, not on $r$. Example:

$$\forall r : AddCBV.(S.Zero).(S.(S.Zero)).r \twoheadrightarrow r.(S.(S.(S.Zero)))$$

- *Return continuations.* These represent the state of the program, parameterized over some values. Imagine a C program fragment "`return abs(2 - ?);`". If we were to resume execution from such fragment, then the program would run to completion, but it is necessary to first fill in the question mark. If $r$ represents the above program fragment, then $r.3$ represents the completed fragment "`return abs(2 - 3);`".

  If a return continuation has arity $n$, then it corresponds to a program fragment with $n$ question marks.

**Invalid terms.** All invalid terms will be considered equivalent. If $M$ is invalid, then $M.N$ is also invalid.

**Complete terms.** This is the set of terms that have a successor. If $M$ is complete, then $M.N$ is invalid.

**Common definitions**

$$\begin{aligned} \mathit{Zero}.z.s &\longrightarrow z \\ S.m.z.s &\longrightarrow s.m \\ \mathit{Nil}.\mathit{ifempty}.\mathit{iflist} &\longrightarrow \mathit{ifempty} \\ \mathit{Cons}.n.l.\mathit{ifempty}.\mathit{iflist} &\longrightarrow \mathit{iflist}.n.l \end{aligned}$$

**Call-by-value functions**

$$\begin{aligned} \mathit{AddCBV}.x.y.r &\longrightarrow x.(r.y).(\mathit{AddCBV}'.y.r) \\ \mathit{AddCBV}'.y.r.x' &\longrightarrow \mathit{AddCBV}.x'.(S.y).r \\[1ex] \mathit{FibCBV}.x.r &\longrightarrow x.(r.\mathit{Zero}).(\mathit{FibCBV}_1.r) \\ \mathit{FibCBV}_1.r.y &\longrightarrow y.(r.(S.\mathit{Zero})).(\mathit{FibCBV}_2.r.y) \\ \mathit{FibCBV}_2.r.y.y' &\longrightarrow \mathit{FibCBV}.y.(\mathit{FibCBV}_3.r.y') \\ \mathit{FibCBV}_3.r.y'.\mathit{fib}_y &\longrightarrow \mathit{FibCBV}.y'.(\mathit{FibCBV}_4.r.\mathit{fib}_y) \\ \mathit{FibCBV}_4.r.\mathit{fib}_y.\mathit{fib}_{y'} &\longrightarrow \mathit{AddCBV}.\mathit{fib}_y.\mathit{fib}_{y'}.r \end{aligned}$$

**Call-by-name functions**

$$\begin{aligned} \mathit{AddCBN}.x.y.z.s &\longrightarrow x.(y.z.s).(\mathit{AddCBN}'.y.s) \\ \mathit{AddCBN}'.y.s.x' &\longrightarrow s.(\mathit{AddCBN}.x'.y) \\[1ex] \mathit{FibCBN}.x.z.s &\longrightarrow x.z.(\mathit{FibCBN}_1.z.s) \\ \mathit{FibCBN}_1.z.s.y &\longrightarrow y.(s.\mathit{Zero}).(\mathit{FibCBN}_2.z.s.y) \\ \mathit{FibCBN}_2.z.s.y.y' &\longrightarrow \mathit{AddCBN}.(\mathit{FibCBN}.y).(\mathit{FibCBN}.y').z.s \end{aligned}$$

Figure 1: Continuation calculus representations of $+$ and $\mathsf{fib}$. The functions are applied in a different way, as shown in Figure 2. This incompatibility is already indicated by the different arity: $\mathsf{arity}(\mathit{AddCBV}) = 3 \neq \mathsf{arity}(\mathit{AddCBN}) = 4$, and $\mathsf{arity}(\mathit{FibCBV}) = 2 \neq \mathsf{arity}(\mathit{FibCBN}) = 3$. Figure 2 shows how to use the four functions.

## 4 Reasoning with CC terms

This section sketches the nature of continuation calculus through theorems. All proofs are included in the appendix.

### 4.1 Fresh names

**Definition 11.** When a name $\mathit{fr}$ does not occur in the program under consideration, then we call $\mathit{fr}$ a *fresh name*. Furthermore, all fresh names that we assume within theorems, lemmas, and propositions are understood to be different. When we say $\mathit{fr}$ is fresh *for some objects*, then it is additionally required that $\mathit{fr}$ is not mentioned in those objects.

We can always assume another fresh name, because programs are finite and there are infinitely many names.

*Interpretation.* Fresh names allow us to reason on arbitrary terms, much like free variables in lambda calculus.

**Theorem 12.** *Let $M,N$ be terms, and let name $\mathit{fr}$ be fresh. The following equivalences hold:*

$$\begin{aligned} M \twoheadrightarrow N &\iff \forall t \in \mathscr{U} : M[\mathit{fr} := t] \twoheadrightarrow N[\mathit{fr} := t] \\ M \twoheadrightarrow\downarrow &\iff \exists t \in \mathscr{U} : M[\mathit{fr} := t] \twoheadrightarrow\downarrow \end{aligned}$$

**Lemma 13** (determinism)**.** *Let $M, \vec{t}, \vec{u}$ be terms, and let $m,n$ be undefined names in $P$. If $M \twoheadrightarrow_P m.t_1.\cdots.t_k$ and $M \twoheadrightarrow_P n.u_1.\cdots.u_l$, then $m.t_1.\cdots.t_k = n.u_1.\cdots.u_l$.*

*Remark* 14. If $m$ or $n$ is defined, this may not hold. For instance, in the program "$A \longrightarrow B; B \longrightarrow C$", we have $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$, yet $B \neq C$.

## 4.2 Term equivalence

Besides syntactic equality ($=$), we introduce two equivalences on terms: common reduct ($=_P$) and observational equivalence ($\approx_P$).

**Definition 15.** Terms $M, N$ have a common reduct if $M \twoheadrightarrow t \twoheadleftarrow N$ for some term $t$. Notation: $M =_P N$.

**Proposition 16.** *Suppose $M =_P N\downarrow$. Then $M \twoheadrightarrow N$.*

Common reduct is a strong equivalence, comparable to $\beta$-conversion for lambda calculus. Terms $M \neq N$ can only have a common reduct if at least one of them is complete. This makes pure $=_P$ unsuitable for relating data or function terms, which are incomplete. In fact, $=_P$ is not a congruence.

To remedy this, we define an observational equivalence in terms of termination.

**Definition 17.** Terms $M$ and $N$ are *observationally equivalent* under a program $P$, notation $M \approx_P N$, when for all extension programs $P' \supseteq P$ and terms $X$:

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

We may write $M \approx N$ if the program is implicit.

Examples: $AddCBV.\langle m\rangle.\langle 0\rangle \approx AddCBV.\langle 0\rangle.\langle m\rangle$ and $\langle 0\rangle \approx \langle \mathsf{True}\rangle$, but $\langle 0\rangle \not\approx \langle 1\rangle$; see Section 5.

**Lemma 18.** *$\approx$ is a congruence. In other words, if $M \approx M'$ and $N \approx N'$, then $M.N \approx M'.N'$.*

**Characterization** The reduction behavior of complete terms divides them in three classes. Observational equivalence distinguishes the classes.

- *Nontermination.* When $M$ is nonterminating and the program is extended, $M$ remains nonterminating.

  If the reduction path of $M$ is finite, we call it *terminating*, and we may write $M \twoheadrightarrow\downarrow$. This is shorthand for $\exists N \in \mathscr{U} : M \twoheadrightarrow N\downarrow$.
- *Proper reduction to an incomplete or invalid term.* All such $M$ are observationally equivalent to an invalid term. When the program is extended, such terms remain in their execution class.
- *Proper reduction to an undefined term.* Observational equivalence distinguishes terms $M, N$ if the head of their final term is different. Therefore, there are infinitely many subclasses.

  When the program is extended, the final term may become defined. This can cause such $M$ to fall in a different class.

The following proposition and theorem show that $\approx$ distinguishes three equivalence classes: if $M \approx N$, then $M$ and $N$ are in the same class.

**Proposition 19.** *If $M \approx N$, then $M \twoheadrightarrow\downarrow \Leftrightarrow N \twoheadrightarrow\downarrow$.*

**Theorem 20.** *Let $M \approx N$ and $M \twoheadrightarrow fr.t_1.\cdots.t_k\downarrow$ with $fr \notin \mathsf{dom}(P)$. Then $N \twoheadrightarrow fr.u_1.\cdots.u_k\downarrow$ for some $\vec{u}$.*

**Retrieving observational equivalence** Complete terms with a common reduct are observationally equal. If $M, N$ are incomplete, but they have common reducts when extended with terms, then also $M \approx N$.

**Theorem 21.** *Let $M, N$ be terms with arity $k$. If $M.t_1.\cdots.t_k =_P N.t_1.\cdots.t_k$ for all $\vec{t}$, then $M \approx N$.*

**Corollary 22.** *Suppose $M =_P N$ and $\mathsf{arity}(M) = \mathsf{arity}(N) = 0$. Then $M \approx N$.*

*Remark* 23. $M \twoheadrightarrow N$ does not always imply $M \approx N$ if $\mathsf{arity}(N) > 0$. For instance, take the following program:

$$\begin{aligned} Goto.x &\longrightarrow x \\ Omega.x &\longrightarrow x.x \end{aligned}$$

Then $Goto.Omega \to Omega$, an incomplete term. We cannot 'fix' $Goto.Omega$ by appending another term: $Goto.Omega.Omega$ is invalid. Name $Goto$ is defined for one 'operand' term, and the superfluous $Omega$ term cannot be 'memorized' as with lambda calculus. On the other hand, $Omega.Omega \to Omega.Omega$ is nonterminating. Hence, $Goto.Omega \to Omega$ but $Goto.Omega \not\approx Omega$.

Note that $Goto.Omega \not\approx Omega$ is only possible because $\mathsf{arity}(Goto.Omega) \neq \mathsf{arity}(Omega)$.

## 4.3 Program substitution and union

**Definition 24** (fresh substitution)**.** Let $n_1 \dots n_k$ be names, and $m_1 \dots m_k$ be fresh for $M$, all different. Then $M[\vec{n} := \vec{m}]$ is equal to $M$ where all occurrences of $\vec{n}$ are simultaneously replaced by $\vec{m}$, respectively. The *fresh substitution* $P[\vec{n} := \vec{m}]$ replaces all $\vec{n}$ by $\vec{m}$ in both left and right hand sides of the rules of $P$.

We can combine two programs by applying a fresh substitution to one of them, and taking the union. As the following theorems shows, this preserves most interesting properties.

**Theorem 25.** *Suppose that $P' \supseteq P$ is an extension program, and $M,N$ are terms. Then the left hand equations hold. Let $\sigma$ denote a fresh substitution $[\vec{n} := \vec{m}]$. Then the right hand equations hold.*

$$\begin{array}{rcl@{\qquad\qquad}rcl} M \to_P N & \Longrightarrow & M \to_{P'} N & M \to_P N & \Longleftrightarrow & M\sigma \to_{P\sigma} N\sigma \\ M \twoheadrightarrow_P N & \Longrightarrow & M \twoheadrightarrow_{P'} N & M \twoheadrightarrow_P N & \Longleftrightarrow & M\sigma \twoheadrightarrow_{P\sigma} N\sigma \\ M \downarrow_P & \Longleftarrow & M \downarrow_{P'} & M \downarrow_P & \Longleftrightarrow & M\sigma \downarrow_{P\sigma} \\ M =_P N & \Longrightarrow & M =_{P'} N & M =_P N & \Longleftrightarrow & M\sigma =_{P\sigma} N\sigma \\ M \approx_P N & \Longrightarrow & M \approx_{P'} N & M \approx_P N & \Longleftrightarrow & M\sigma \approx_{P\sigma} N\sigma \end{array}$$

*Remark* 26. Names $\vec{n}$ are not mentioned in $M\sigma$ and $P\sigma$, so we can apply Theorem 25 with $\sigma^{-1}$ on $M\sigma$ and $P\sigma$.

**Theorem 27.** *Suppose that $P'$ extends $P$, but $\mathsf{dom}(P' \setminus P)$ are not mentioned in $M$, $N$, or $P$.*
*Then $M \approx_P N \Longleftrightarrow M \approx_{P'} N$.*

## 5 Data terms and functions

In this section, we show how to program some standard data in continuation calculus. We first give a canonical representation of data as CC terms. We then give essential semantic characteristics, and show that other terms have those characteristics as well. Observational equivalence guarantees that termination of the whole program is only dependent on those characteristics. In fact, it will prove possible to implement "call-by-name values", which delay computation until it is needed, by relying on those characteristics.

**Standard representation of data** In Section 1, we postulated terms for natural numbers in continuation calculus. We will now give this standard representation formally, as well as the representation of booleans and natural lists.

**Definition 28.** For a mathematical object $o$, we define a standard representation $\langle o\rangle$ of that object as a CC term, which we call a *data term*. We postulate that the rules in the middle column are included in programs that use the corresponding terms.

$$\begin{array}{rlrl} \langle\mathsf{True}\rangle &= \mathit{True} & \mathit{True}.t.f &\longrightarrow t \\ \langle\mathsf{False}\rangle &= \mathit{False} & \mathit{False}.t.f &\longrightarrow f \end{array}\Bigg\}\text{ booleans}$$

$$\begin{array}{rlrl} \langle 0\rangle &= \mathit{Zero} & \mathit{Zero}.z.s &\longrightarrow z \\ \langle m+1\rangle &= S.\langle m\rangle & S.x.z.s &\longrightarrow s.x \end{array}\Bigg\}\text{ naturals}$$

$$\begin{array}{rlrl} \langle[]\rangle &= \mathit{Nil} & \mathit{Nil}.e.c &\longrightarrow e \\ \langle m:l\rangle &= \mathit{Cons}.\langle m\rangle.\langle l\rangle & \mathit{Cons}.x.xs.e.c &\longrightarrow c.x.xs \end{array}\Bigg\}\text{ lists of naturals}$$

**Theorem 29.** $\langle\mathsf{True}\rangle \not\approx \langle\mathsf{False}\rangle$.

*Proof.* Observe that for all $t,f$, $\langle\mathsf{True}\rangle.t.f \to t$ and $\langle\mathsf{False}\rangle.t.f \to f$. Take two fresh names $t$ and $f$. Contraposition of Theorem 20 proves $\langle\mathsf{True}\rangle.t.f \not\approx \langle\mathsf{False}\rangle.t.f$. Because $\approx$ is a congruence with respect to dot, we can conclude $\langle\mathsf{True}\rangle \not\approx \langle\mathsf{False}\rangle$. □

Similar results hold for $\mathbb{N}$ and $\mathsf{List}_{\mathbb{N}}$, but we do not provide a proof here.

**A broader definition** The behavioral essence of these data terms is that they take a continuation for each constructor, and they continue execution in the respective continuation, augmented with the constructor arguments. For instance, $\langle 0\rangle.z.s \twoheadrightarrow z$ and $\langle n+1\rangle.z.s \twoheadrightarrow s.\langle n\rangle$. We can capture this essence in the following term sets; $[\![\mathbb{N}]\!]$ and $[\![\mathsf{List}_{\mathbb{N}}]\!]$ are the smallest sets satisfying the following equalities.

$$\begin{aligned} [\![\mathbb{B}]\!] &= \{M \in \mathscr{U} \mid \forall t,f \in \mathscr{U} : M.t.f \twoheadrightarrow t \vee M.t.f \twoheadrightarrow f\} \\ [\![\mathbb{N}]\!] &= \{M \in \mathscr{U} \mid (\forall z,s \in \mathscr{U} : M.z.s \twoheadrightarrow z) \\ &\qquad \vee \exists x \in [\![\mathbb{N}]\!]\ \forall z,s \in \mathscr{U} : M.z.s \twoheadrightarrow s.x\} \\ [\![\mathsf{List}_{\mathbb{N}}]\!] &= \{M \in \mathscr{U} \mid (\forall e,c \in \mathscr{U} : M.e.c \twoheadrightarrow e) \\ &\qquad \vee \exists x \in [\![\mathbb{N}]\!], xs \in [\![\mathsf{List}_{\mathbb{N}}]\!]\ \forall e,c \in \mathscr{U} : M.e.c \twoheadrightarrow c.x.xs\} \end{aligned}$$

*Remark* 30. These sets are dependent on the program. The sets are monotone with respect to program extension: if $M \in [\![\mathbb{B}]\!]$, $\in [\![\mathbb{N}]\!]$, or $\in [\![\mathsf{List}_{\mathbb{N}}]\!]$ for a program, then $M$ is also in the corresponding set for any extension program.

The sets include other terms besides $\langle\mathsf{True}\rangle$, $\langle\mathsf{False}\rangle$, $\langle n\rangle$, and $\langle l\rangle$. Consider the following program fragment, which implements the $\leq$ operator on natural numbers.

$$\begin{aligned} \mathit{Leq}.x.y.t.f &\longrightarrow x.t.(\mathit{Leq}'.y.t.f) \\ \mathit{Leq}'.y.t.f.x' &\longrightarrow \mathit{Leq}.y.x'.f.t \end{aligned}$$

Given naturals $m,p$ and this program fragment, $\mathit{Leq}.\langle m\rangle.\langle p\rangle \in [\![\mathbb{B}]\!]$. Even more, $\mathit{Leq}.\langle m\rangle.\langle p\rangle \approx \langle m \leq p\rangle$. In general, it follows from Theorem 21 that all $M \in [\![\mathbb{B}]\!]$ are observationally equivalent to $\langle\mathsf{True}\rangle$ or $\langle\mathsf{False}\rangle$. The appendix contains a proof of the analogous statement for $[\![\mathbb{N}]\!]$:

**Proposition 31.** *All terms in $[\![\mathbb{N}]\!]$ are observationally equivalent to $\langle k\rangle$ for some $k$.*

For further reasoning, it is useful to split up $[\![\mathbb{N}]\!]$ in parts as follows.

**Definition 32.** For a natural number $k$, the set $\langle\!\langle k\rangle\!\rangle$ is defined as $\{M \in [\![\mathbb{N}]\!] | M \approx \langle k\rangle\}$. We define $\langle\!\langle b\rangle\!\rangle$ and $\langle\!\langle l\rangle\!\rangle$ analogously for booleans $b$ and lists of naturals $l$.

With this definition, we may say $\mathit{Leq}.\langle 3\rangle.\langle 4\rangle \in \langle\!\langle\mathsf{True}\rangle\!\rangle$. In fact, if $a \in \langle\!\langle 3\rangle\!\rangle$ and $b \in \langle\!\langle 4\rangle\!\rangle$, then $\mathit{Leq}.a.b \in \langle\!\langle\mathsf{True}\rangle\!\rangle$.[1] To support this pattern of reasoning, we allow to lift $\langle\!\langle\cdot\rangle\!\rangle$, denoting a *term*. The

[1]To see this, observe $\mathit{Leq}.a.b \approx \mathit{Leq}.\langle 3\rangle.\langle 4\rangle \approx \langle\mathsf{True}\rangle$ by congruence, then use Theorems 20 and 12.

resulting statements are implicitly quantified universally and existentially, and are usable in proof chains.

*Remark* 33. For data terms, we would like to reason and compute with equivalence classes of representations, $\langle\!\langle k\rangle\!\rangle$, instead of with the representations themselves, $\langle k\rangle$. Of course, a CC program will always compute with a term (and not with an equivalence class of terms), but we would like this computation to only depend on the characterization of the equivalence class.

For example, we want to compute a CBN addition function *AddCBN*, such that for all $m, p \in \mathbb{N}$, $\forall t \in \langle\!\langle m\rangle\!\rangle \forall u \in \langle\!\langle p\rangle\!\rangle : \mathit{AddCBN}.t.u \in \langle\!\langle m+p\rangle\!\rangle$. As a specification, we want to summarize this as:

$$\mathit{AddCBN}.\langle\!\langle m\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle \in \langle\!\langle m+p\rangle\!\rangle$$

We will also summarize a statement of the form $\forall t_1 \in \langle\!\langle m\rangle\!\rangle\ \exists t_2 \in \langle\!\langle m\rangle\!\rangle\ \exists t_3 \in \langle\!\langle l\rangle\!\rangle : A.t_1 \twoheadrightarrow B.t_2.t_3$ with the shorthand $A.\langle\!\langle m\rangle\!\rangle \twoheadrightarrow B.\langle\!\langle m\rangle\!\rangle.\langle\!\langle l\rangle\!\rangle$. If we know $A.\langle\!\langle m\rangle\!\rangle \twoheadrightarrow B.\langle\!\langle m\rangle\!\rangle.\langle\!\langle l\rangle\!\rangle$ and $B.\langle\!\langle m\rangle\!\rangle.\langle\!\langle l\rangle\!\rangle \twoheadrightarrow C.\langle\!\langle m\rangle\!\rangle$, then we may logically conclude

$$\forall t_1 \in \langle\!\langle m\rangle\!\rangle\ \exists t_2 \in \langle\!\langle m\rangle\!\rangle\ \exists t_3 \in \langle\!\langle l\rangle\!\rangle\ \exists t_4 \in \langle\!\langle m\rangle\!\rangle : A.t_1 \twoheadrightarrow B.t_2.t_3 \twoheadrightarrow C.t_4\ ,$$

which we will summarize as $A.\langle\!\langle m\rangle\!\rangle \twoheadrightarrow B.\langle\!\langle m\rangle\!\rangle.\langle\!\langle l\rangle\!\rangle \twoheadrightarrow C.\langle\!\langle m\rangle\!\rangle$. Analogous statements of this form, and longer series, will be summarized in a similar way. In particular, it will suit us to also use $\rightarrow$ and $=_P$ in longer derivations.

**Example: delayed addition** We will program a different addition on natural numbers: one that delays work as long as possible, like in call-by-name programming languages. We use the following algorithm, for natural numbers $m, p$:

$$\begin{aligned} 0+p &= p \\ S(m)+p &= S(m+p) \end{aligned}$$

The resulting name *AddCBN* will be a 'call-by-name' function, with specification $\mathit{AddCBN}.\langle\!\langle m\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle \in \langle\!\langle m+p\rangle\!\rangle$, so we have to build a rule for *AddCBN*. Because $\mathit{AddCBN}.\langle\!\langle m\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle \in [\![\mathbb{N}]\!]$, $\mathsf{arity}(\mathit{AddCBN}) = 4$. We reduce the specification with a case distinction on the first argument.

$$\begin{aligned} \mathit{AddCBN}.\langle\!\langle 0\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle.z.s &=_P \langle\!\langle p\rangle\!\rangle.z.s, \qquad (\mathit{AddCBN}.\langle\!\langle 0\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle \text{ has the same specification as } \langle\!\langle p\rangle\!\rangle) \\ \mathit{AddCBN}.\langle\!\langle S(m)\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle.z.s &\twoheadrightarrow s.\langle\!\langle m+p\rangle\!\rangle \end{aligned}$$

We must make the case distinction by using the specified behavior of the first argument. This suggests a rule of the form $\mathit{AddCBN}.x.y.z.s \longrightarrow x.(y.z.s).(s.(\mathit{AddCBN}.x'.y))$. It almost works:

$$\begin{aligned} \mathit{AddCBN}.\langle\!\langle 0\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle.z.s &\twoheadrightarrow \langle\!\langle p\rangle\!\rangle.z.s \\ \mathit{AddCBN}.\langle\!\langle S(m)\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle.z.s &\twoheadrightarrow s.(\mathit{AddCBN}.x'.\langle\!\langle p\rangle\!\rangle).\langle\!\langle m\rangle\!\rangle \end{aligned}$$

However, variable $x'$ is not in the left-hand side, so this is not a valid rule. Furthermore, if $x = S(x')$, then $x'$ would be erroneously appended to $s.(\mathit{AddCBN}.x'.y)$. We fix *AddCBN* with a helper name $\mathit{AddCBN}'$, with specification $\mathit{AddCBN}'.\langle\!\langle p\rangle\!\rangle.s.\langle\!\langle m\rangle\!\rangle \twoheadrightarrow s.\langle\!\langle m+p\rangle\!\rangle$.

$$\begin{aligned} \mathit{AddCBN}.x.y.z.s &\longrightarrow x.(y.z.s).(\mathit{AddCBN}'.y.s)) \\ \mathit{AddCBN}'.y.s.x' &\longrightarrow s.(\mathit{AddCBN}.x'.y) \end{aligned}$$

This version conforms to the specification.

$$\begin{aligned} \mathit{AddCBN}.\langle\!\langle 0\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle.z.s &\twoheadrightarrow \langle\!\langle p\rangle\!\rangle.z.s = \langle\!\langle p\rangle\!\rangle \\ \mathit{AddCBN}.\langle\!\langle S(m)\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle.z.s &\twoheadrightarrow \mathit{AddCBN}'.\langle\!\langle p\rangle\!\rangle.s.\langle\!\langle m\rangle\!\rangle \\ &\rightarrow s.(\mathit{AddCBN}.\langle\!\langle m\rangle\!\rangle.\langle\!\langle p\rangle\!\rangle) = s.\langle\!\langle m+p\rangle\!\rangle \end{aligned}$$

| **Call-by-value** $\mathsf{fib}(7)$ | **Call-by-name** $\mathsf{fib}(7)$ |
|---|---|
| To apply $f$ to $\vec{x}$, evaluate $f.\vec{x}.r \twoheadrightarrow r.y$ for some $r$. Then $y$ is the result. | To apply $f$ to $\vec{x}$, write $f.\vec{x}$. This is directly a data term, no reduction happens. |
| *The result of* $\mathsf{fib}(7)$ *is 13, obtained in 362 reduction steps:* $FibCBV.7.fr \twoheadrightarrow fr.13$ | *By the specification of FibCBN, we know* $FibCBN.7 \in \langle\!\langle 13 \rangle\!\rangle$. *No reduction is involved.* |

Both 13 and $FibCBN.7$ can be used in other functions, like $+$. Because they are observationally equivalent, they can be substituted for each other in a term. That does not affect termination, or the head of the final term if that is undefined (Theorem 20). However, substituting 13 for $FibCBN.7$ may make the evaluation shorter.

| | |
|---|---|
| $13 +_{CBV} 0$ *is obtained in 41 steps:* $AddCBV.13.0.fr \twoheadrightarrow fr.13$ in 41 steps | $FibCBN.7 +_{CBV} 0$ *is obtained in 304 steps:* $AddCBV.(FibCBN.7).0.fr \twoheadrightarrow fr.13$ in 304 steps (263 more) |

Our implementation of *AddCBV* does not examine the right argument, as the converse addition shows.

| | |
|---|---|
| $AddCBV.0.13.fr \twoheadrightarrow fr.13$ in 2 steps | $AddCBV.0.(FibCBN.7).fr \twoheadrightarrow fr.(FibCBN.7)$ in 2 steps |

Figure 2: Calculating $\mathsf{fib}(7)$, $\mathsf{fib}(7)+0$, and $0+\mathsf{fib}(7)$ using call-by-value and call-by-name. Effectively, *FibCBN* delays computation until it is needed. A natural number $n$ stands for $\underbrace{S.(\cdots .(S}_{n\text{ times}}.Zero)\cdots)$.

## 5.1 Call-by-name and call-by-value functions

We regard two kinds of functions. We call them *call-by-name* and *call-by-value*, by analogy with the evaluation stategies for lambda calculus. Figure 1 defines a CBN and CBV version of addition on naturals and the Fibonacci function. Figure 2 shows how to use them. It also illustrates that the CBV function performs work eagerly, while the CBN function delays work until it is needed: hence the analogy.

- *Call-by-name functions* are terms $f$ such that $f.v_1.\cdots.v_k$ is a data term for all $\vec{v}$ in a certain domain. Example specifications for such $f$:

$$AddCBN.\langle\!\langle m \rangle\!\rangle.\langle\!\langle p \rangle\!\rangle \in \langle\!\langle m+p \rangle\!\rangle$$
$$FibCBN.\langle\!\langle m \rangle\!\rangle.\langle\!\langle p \rangle\!\rangle \in \langle\!\langle \mathsf{fib}(m) \rangle\!\rangle$$

- *Call-by-value functions* are terms $f$ of arity $n+1$ such that for all $\vec{v}$ in a certain domain, $\forall r : f.v_1.\cdots.v_n.r \twoheadrightarrow r.t$ with data term $t$ depending only on $\vec{v}$, not on $r$. Example specifications for such $f$:

$$\forall r : AddCBV.\langle\!\langle m \rangle\!\rangle.\langle\!\langle p \rangle\!\rangle.r \twoheadrightarrow r.\langle\!\langle m+p \rangle\!\rangle$$
$$\forall r : FibCBV.\langle\!\langle m \rangle\!\rangle.r \twoheadrightarrow r.\langle \mathsf{fib}(m) \rangle$$

  The output of *FibCBV* is always a standard representation. Because our implementation of *AddCBV* does not inspect the second argument, its output may not be a standard integer. An example of this is shown in Figure 2.

We leave formal proofs of the specifications for future work.

## 6 Modeling programs with control

To illustrate how control is fundamental to continuation calculus, we give an example program that multiplies a list of natural numbers, and show how an escape from a loop can be modeled without a special operator in the natural CC representation. We use an ML-like programming language for this example, and show the corresponding call-by-value program for CC.

The naive way to compute the product of a list is as follows:

```
let rec $listmult_1\ l$ = match $l$ with
  | $[] \to 1$
  | $(x : xs) \to x \cdot listmult_1\ xs$
```

$$\begin{aligned} ListMult.l.r &\longrightarrow l.(r.(S.Zero)).(C.r) \\ C.r.x.xs &\longrightarrow ListMult.xs.(PostMult.x.r) \\ PostMult.x.r.y &\longrightarrow Mult.x.y.r \end{aligned}$$

Note that if $l$ contains a zero, then the result is always zero. One might wish for a more efficient version that skips all numbers after zero.

```
let rec $listmult_2\ l$ = match $l$ with
  | $[] \to 1$
  | $(x : xs) \to$ match $x$ with
      | $0 \to 0$
      | $x' + 1 \to x \cdot listmult_2\ l$
```

$$\begin{aligned} ListMult.l.r &\longrightarrow l.(r.(S.Zero)).(B.r) \\ B.r.x.xs &\longrightarrow x.(r.Zero).(C.r.x.xs) \\ C.r.x.xs.x' &\longrightarrow ListMult.xs.(PostMult.x.r) \\ PostMult.x.r.y &\longrightarrow Mult.x.y.r \end{aligned}$$

However, $listmult_2$ is not so efficient either: if the list is of the form $[x_1 + 1, \cdots, x_k + 1, 0]$, then we only avoid multiplying $0 \cdot listmult_2\,[]$. The other multiplications are all of the form $n \cdot 0 = 0$. We also want to avoid execution of those surrounding multiplications. We can do so if we extend ML with the call/cc operator, which creates alternative exit points that are invokable as a function.

```
let $listmult_3\ l$ =
  call/cc $(\lambda abort.$
    $A\ l$
    where $A$ = function
      | $[] \to 1$
      | $(x : xs) \to$ $\boxed{\begin{array}{l}\text{match } x \text{ with} \\ \quad \mid 0 \to abort\ 0 \\ \quad \mid x' + 1 \to \boxed{x \cdot A\ xs}_{(C)}\end{array}}_{(B)}$ )
```

*The boxes are not syntax, but are used to relate $listmult_3$ to Figure 3.*

While $listmult_3$ is not readily expressible in actual ML or lambda calculus, it is natural to express in CC: we list the program in Figure 3.

These programs are a CPS translation of $listmult_3$, with one exception: the variable *abort* in Figure 3 corresponds to the partial application of *abort* to 0 in $listmult_3$. Note that in CC, *abort* is obtained simply by constructing *r.Zero*. The variable $r$ globally corresponds to the return continuation that is implicit in ML. Continuation calculus requires to explicitly thread variables through the continuations.

## 7 Correctness of ListMult

This section proves that *ListMult* in Figure 3 is correct. The idea is to assume that a program contains the listed definitions, and *Mult* behaves according to the specification; then Theorem 36 proves the specification of *ListMult* in that program.

**Continuation calculus**

— *Assume* $m, m', p \in \mathbb{N}$, $l \in \mathsf{List}_{\mathbb{N}}$, $r, r_0 \in \mathscr{U}$.
$\mathit{ListMult}.xs.r \longrightarrow A.xs.r.(r.\mathit{Zero})$
**Theorem.** $\mathit{ListMult}.\langle\!\langle l \rangle\!\rangle.r \twoheadrightarrow r.\langle\!\langle \mathsf{product}\ l \rangle\!\rangle$

— Assume $r.\mathit{Zero} =_P r_0$.
$A.xs.r.\mathit{abort} \longrightarrow xs.(r.(S.\mathit{Zero})).(B.r.\mathit{abort})$
**Lemma.** $A.\langle\!\langle l \rangle\!\rangle.r.r_0 =_P r.\langle\!\langle \mathsf{product}\ l \rangle\!\rangle$

$B.r.\mathit{abort}.x.xs \longrightarrow x.\mathit{abort}.(C.r.\mathit{abort}.x.xs)$
$\Longrightarrow \quad B.r.r_0.\langle\!\langle m \rangle\!\rangle.\langle\!\langle l \rangle\!\rangle =_P r.\langle\!\langle m \cdot \mathsf{product}\ l \rangle\!\rangle$

$C.r.\mathit{abort}.x.xs.x' \longrightarrow A.xs.(\mathit{PostMult}.x.r).\mathit{abort}$
$\Longrightarrow \quad C.r.r_0.\langle\!\langle m \rangle\!\rangle.\langle\!\langle l \rangle\!\rangle.x' =_P r.\langle\!\langle m \cdot \mathsf{product}\ l \rangle\!\rangle$

$\mathit{PostMult}.x.r.y \longrightarrow \mathit{Mult}.x.y.r$
$\Longrightarrow \quad \mathit{PostMult}.\langle\!\langle m \rangle\!\rangle.r.\langle\!\langle p \rangle\!\rangle \twoheadrightarrow r.\langle\!\langle m \cdot p \rangle\!\rangle$

$\mathit{Mult}.x.y.r \longrightarrow y.(r.\mathit{Zero}).(\mathit{PostMult}.x.(\mathit{PostAdd}.x.r))$
**Assumption.** $\mathit{Mult}.\langle\!\langle m \rangle\!\rangle.\langle\!\langle p \rangle\!\rangle.r \twoheadrightarrow r.\langle\!\langle m \cdot p \rangle\!\rangle$

**Usage.** $\mathit{ListMult}.\langle\!\langle [3,1,2] \rangle\!\rangle.r \twoheadrightarrow r.\langle\!\langle 6 \rangle\!\rangle$

**Haskell equivalent**

— *Assume* $l, xs \in [\mathbb{N}]$, $x, x', y \in \mathbb{N}$.
$\mathit{listmult}_4\ l\ r = A\ l\ r\ (r\ 0)$
$\Longrightarrow \quad \mathit{listmult}_4\ l\ r = r\ (\mathsf{product}\ l)$

$A\ l\ r\ \mathit{abort} = \mathsf{case}\ l\ \mathsf{of}$
$\quad |\ [] \to r\ 1$
$\quad |\ x : xs \to B\ r\ \mathit{abort}\ x\ xs$
$\Longrightarrow \quad A\ l\ r\ (r\ 0) = r\ (\mathsf{product}\ l)$

$B\ r\ \mathit{abort}\ x\ xs = \mathsf{case}\ x\ \mathsf{of}$
$\quad |\ 0 \to \mathit{abort}$
$\quad |\ y+1 \to C\ r\ \mathit{abort}\ x\ xs\ y$
$\Longrightarrow \quad B\ r\ (r\ 0)\ x\ xs = r\ (x \cdot \mathsf{product}\ xs)$

$C\ r\ \mathit{abort}\ x\ xs\ x' = A\ xs\ (\mathit{PostMult}\ x\ r)\ \mathit{abort}$
$\Longrightarrow \quad C\ r\ (r\ 0)\ x\ xs\ x' = r\ (x \cdot \mathsf{product}\ xs)$

$\mathit{PostMult}\ x\ r\ y = r\ (x \cdot y)$

**Usage.** $6 == \mathit{listmult}_4\ [3,1,2]\ \mathit{id}$

Figure 3: Left: 'fast' list multiplication in continuation calculus (CC). Right: Haskell program with equivalent semantics. Statements after $\Longrightarrow$ serve to guide the reader. The theorem and lemma are proven in Section 7.

We need two lemmas. Firstly, we show that name $A$ conforms to its specification. This is done by induction on list $l$. Furthermore, we need a lemma on the quick exit of *PostMult*.

**Lemma 34.** *The specification of A is satisfied. That is, assume* $l \in \mathsf{List}_{\mathbb{N}}$, $r, r_0 \in \mathscr{U}$ *such that* $r.\langle 0 \rangle =_P r_0$. *Then* $A.\langle\!\langle l \rangle\!\rangle.r.r_0 =_P r.\langle\!\langle \mathsf{product}\ l \rangle\!\rangle$.

*Proof.* We use induction on $l$, and make a three-way case distinction.

*Case* 1. Base case: $l = []$. Then:

$$\begin{array}{lll} & A.\langle\!\langle [] \rangle\!\rangle.r.r_0 & \\ \to & \langle\!\langle [] \rangle\!\rangle.(r.(S.\mathit{Zero})).(B.r.r_0) & \text{by definition} \\ \twoheadrightarrow & r.(S.\mathit{Zero}) & \text{by definition of } \langle\!\langle [] \rangle\!\rangle \\ = & r.\langle\!\langle \mathsf{product}\ [] \rangle\!\rangle & S.\mathit{Zero} \in \langle\!\langle 1 \rangle\!\rangle = \langle\!\langle \mathsf{product}\ [] \rangle\!\rangle \end{array}$$

*Case* 2. $l = (0 : l')$. Then:

$$\begin{array}{lll} & A.\langle\!\langle 0 : l' \rangle\!\rangle.r.r_0 & \\ \to & \langle\!\langle 0 : l' \rangle\!\rangle.(r.(S.\mathit{Zero})).(B.r.r_0) & \text{by definition of } A \\ \twoheadrightarrow & B.r.r_0.\langle\!\langle 0 \rangle\!\rangle.\langle\!\langle l' \rangle\!\rangle & \text{by definition of } \langle\!\langle 0 : l' \rangle\!\rangle \\ \to & \langle\!\langle 0 \rangle\!\rangle.r_0.(C.r.r_0.\langle\!\langle 0 \rangle\!\rangle.\langle\!\langle l' \rangle\!\rangle) & \text{by definition of } B \\ \twoheadrightarrow & r_0 & \text{by definition of } \langle\!\langle 0 \rangle\!\rangle \\ =_P & r.\mathit{Zero} & \text{by assumption} \\ = & r.\langle\!\langle \mathsf{product}\ (0 : l') \rangle\!\rangle & \mathit{Zero} \in \langle\!\langle 0 \rangle\!\rangle = \langle\!\langle \mathsf{product}\ (0 : l') \rangle\!\rangle \end{array}$$

*Case* 3. $l = (m+1 : l')$. Then:

| | | |
|---|---|---|
| | $A.\langle\!\langle m+1 : l'\rangle\!\rangle.r.r_0$ | |
| $\to$ | $\langle\!\langle m+1 : l'\rangle\!\rangle.(r.(S.Zero)).(B.r.r_0)$ | by definition of $A$ |
| $\twoheadrightarrow$ | $B.r.r_0.\langle\!\langle m+1\rangle\!\rangle.\langle\!\langle l'\rangle\!\rangle$ | by definition of $\langle\!\langle m+1 : l'\rangle\!\rangle$ |
| $\to$ | $\langle\!\langle m+1\rangle\!\rangle.r_0.(C.r.r_0.\langle\!\langle m+1\rangle\!\rangle.\langle\!\langle l'\rangle\!\rangle)$ | by definition of $B$ |
| $\twoheadrightarrow$ | $C.r.r_0.\langle\!\langle m+1\rangle\!\rangle.\langle\!\langle l'\rangle\!\rangle.\langle\!\langle m\rangle\!\rangle$ | by definition of $\langle\!\langle m+1\rangle\!\rangle$ |
| $\to$ | $A.\langle\!\langle l'\rangle\!\rangle.(PostMult.\langle\!\langle m+1\rangle\!\rangle.r).r_0$ | by definition of $C$ |
| $=_P$ | $PostMult.\langle\!\langle m+1\rangle\!\rangle.r.\langle\!\langle \mathsf{product}\ l'\rangle\!\rangle$ | by induction if $r_0 =_P PostMult.\langle\!\langle m+1\rangle\!\rangle.r.\langle 0\rangle$ |
| $\to$ | $Mult.\langle\!\langle m+1\rangle\!\rangle.\langle\!\langle \mathsf{product}\ l'\rangle\!\rangle.r$ | by definition of *Postmult* |
| $\twoheadrightarrow$ | $r.\langle\!\langle (m+1)\cdot \mathsf{product}\ l'\rangle\!\rangle$ | spec *Mult* |
| $=$ | $r.\langle \mathsf{product}\ (m+1 : l')\rangle$ | mathematics |

This chain proves that $A.\langle\!\langle m+1 : l'\rangle\!\rangle.r.r_0 =_P r.\langle\!\langle \mathsf{product}\ (m+1 : l')\rangle\!\rangle$.

The third case requires Lemma 35, which is proved below. This completes the induction, yielding:

$$A.\langle\!\langle l\rangle\!\rangle.r.r_0 =_P r.\langle\!\langle \mathsf{product}\ l\rangle\!\rangle \text{ for all } l \in \mathsf{List}_{\mathbb{N}}, r \in \mathscr{U}\ . \qquad \square$$

**Lemma 35.** *Let $x \in \mathscr{U}$, $l \in \mathsf{List}_{\mathbb{N}}$, $r, r_0 \in \mathscr{U}$ and $r.\langle 0\rangle =_P r_0$. Then $PostMult.x.r.\langle 0\rangle =_P r_0$.*

*Proof.* By the following chain.

| | | |
|---|---|---|
| | $PostMult.x.r.\langle 0\rangle$ | |
| $\to$ | $Mult.x.\langle 0\rangle.r$ | by definition of *PostMult* |
| $\to$ | $\langle 0\rangle.(r.Zero).(PostMult.x.(PostAdd.x.r))$ | by definition of *Mult* |
| $\to$ | $r.Zero = r.\langle 0\rangle$ | by definition of $\langle 0\rangle$ |
| $=_P$ | $r_0$ | by assumption $\square$ |

**Theorem 36.** *The specification of ListMult is satisfied. That is: assume $l \in \mathsf{List}_{\mathbb{N}}$, $r \in \mathscr{U}$. Then $ListMult.\langle\!\langle l\rangle\!\rangle.r \twoheadrightarrow r.\langle\!\langle \mathsf{product}\ l\rangle\!\rangle$.*

*Proof.* We fill in $r_0 = r.Zero$ in the specification of $A$; then $r.Zero =_P r.\langle 0\rangle$ is satisfied by definition of $\langle 0\rangle$.

$$A.\langle\!\langle l\rangle\!\rangle.r.(r.Zero) =_P r.\langle\!\langle \mathsf{product}\ l\rangle\!\rangle \qquad \text{for all } l \in \mathsf{List}_{\mathbb{N}}, r \in \mathscr{U}$$

If we temporarily take $r$ to be a fresh name, then we can change $=_P$ into $\twoheadrightarrow$ with Proposition 16.

$$A.\langle\!\langle l\rangle\!\rangle.r.(r.Zero) \twoheadrightarrow r.\langle\!\langle \mathsf{product}\ l\rangle\!\rangle \qquad \text{for all } l \in \mathsf{List}_{\mathbb{N}}$$

We can generalize this again with Theorem 12:

$$A.\langle\!\langle l\rangle\!\rangle.r.(r.Zero) \twoheadrightarrow r.\langle\!\langle \mathsf{product}\ l\rangle\!\rangle \qquad \text{for all } l \in \mathsf{List}_{\mathbb{N}}, r \in \mathscr{U}$$

Now our main correctness result follows rather straightforwardly.

| | | |
|---|---|---|
| | $ListMult.\langle\!\langle l\rangle\!\rangle.r$ | |
| $\to$ | $A.\langle\!\langle l\rangle\!\rangle.r.(r.Zero)$ | by definition |
| $\twoheadrightarrow$ | $r.\langle\!\langle \mathsf{product}\ l\rangle\!\rangle$ | as just shown $\square$ |

## 8 Conclusions and Future work

We have defined a deterministic calculus that is suitable for modeling programs with control. The calculus has simple operational semantics, and can model both call-by-value and call-by-name programs in such a way that CBN and CBV subprograms can be combined. Call-by-push-value [9] is another calculus in which CBV and CBN lambda calculus can be embedded. We believe that CBV and CBN subprograms can be meaningfully combined in call-by-push-value, but we have not found this in the literature.

In the present paper, we have not yet exploited types. In the future, we will develop a typed version of continuation calculus, which also guarantees the termination of well-typed terms. Another way to look at types is by giving a standard representation of data terms as terms in continuation calculus. In this paper, we have shown how to do this for booleans, natural numbers and lists; in future work we will extend this to other (algebraic, higher order, . . . ) data types. Also, we will develop a generic procedure to transform functions that are defined by pattern matching and equations into terms of continuation calculus.

The determinism in continuation calculus suggests that we can model assignment and side effects using a small number of extra names with special reduction rules. However, such an extension may not preserve observational equivalence. We want to examine if an extension provides a pragmatic model for imperative-functional garbage-collected languages, such as OCaml.

## References

[1] Zena M. Ariola & Hugo Herbelin (2003): *Minimal Classical Logic and Control Operators*. In: *Automata, Languages and Programming*, *Lecture Notes in Computer Science* 2719, Springer Berlin Heidelberg, pp. 871–885, doi:10.1007/3-540-45061-0_68.

[2] Zena M. Ariola & Hugo Herbelin (2008): *Control reduction theories: the benefit of structural substitution*. *Journal of Functional Programming* 18, pp. 373–419, doi:10.1017/S0956796807006612.

[3] Olivier Danvy & Andrzej Filinski (1992): *Representing Control: a Study of the CPS Transformation*. *Mathematical Structures in Computer Science* 2, pp. 361–391, doi:10.1017/S0960129500001535.

[4] R. Kent Dyvbig, Simon Peyton Jones & Amr Sabry (2007): *A monadic framework for delimited continuations*. *Journal of Functional Programming* 17, pp. 687–730, doi:10.1017/S0956796807006259.

[5] Matthias Felleisen (1988): *The theory and practice of first-class prompts*. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, pp. 180–190, doi:10.1145/73560.73576.

[6] Matthias Felleisen & Daniel P. Friedman (1986): *Control operators, the SECD-machine, and the $\lambda$-calculus*. In: *3rd Working Conference on the Formal Description of Programming Concepts*, North-Holland Publishing, pp. 193–219.

[7] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker & Bruce Duba (1987): *A syntactic theory of sequential control*. *Theoretical Computer Science* 52(3), pp. 205–237, doi:10.1016/0304-3975(87)90109-5.

[8] Timothy G. Griffin (1990): *A formulae-as-type notion of control*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, pp. 47–58, doi:10.1145/96709.96714.

[9] Paul Blain Levy (1999): *Call-by-Push-Value: A Subsuming Paradigm*. In: *Typed Lambda Calculi and Applications*, *Lecture Notes in Computer Science* 1581, Springer Berlin Heidelberg, pp. 228–243, doi:10.1007/3-540-48959-2_17.

[10] Michel Parigot (1992): $\lambda$ *-calculus: An algorithmic interpretation of classical natural deduction*. In: *Logic Programming and Automated Reasoning*, *Lecture Notes in Computer Science* 624, Springer Berlin Heidelberg, pp. 190–201, doi:10.1007/BFb0013061.

[11] G.D. Plotkin (1975): *Call-by-name, call-by-value and the $\lambda$-calculus*. *Theoretical Computer Science* 1(2), pp. 125–159, doi:10.1016/0304-3975(75)90017-1.

## A Proofs

We first prove the theorems in Section 4.1, then those in Section 4.3, and finally those in Section 4.2. The theorems within a subsection are not proved in order, and are interspersed with lemmas.

### A.1 General

**Proposition 37.** *Let name $fr$ be not mentioned in term $M$ and program $P$. Then $fr$ is not mentioned in any reduct of $M$.*

*Proof.* By induction and by definition of $\mathsf{next}(M)$. $\square$

**Theorem 38.** *Let $M, N \in \mathscr{U}$, $P$ a program, and $fr$ a name not mentioned in $P$. The following equivalences hold:*

$$\begin{aligned} M \to N &\iff \forall t \in \mathscr{U} : M[fr := t] \to N[fr := t] && (1) \\ M\downarrow &\iff \exists t \in \mathscr{U} : M[fr := t]\downarrow && (2) \\ M \twoheadrightarrow N &\iff \forall t \in \mathscr{U} : M[fr := t] \twoheadrightarrow N[fr := t] && (3) \\ M \twoheadrightarrow\downarrow &\iff \exists t \in \mathscr{U} : M[fr := t] \twoheadrightarrow\downarrow && (4) \end{aligned}$$

This theorem implies Theorem 12.

*Proof.*

($\Leftarrow$1). Fill in $t = fr$.

($\Rightarrow$1). Since $\mathsf{next}_P(M)$ exists, $\mathsf{head}(M)$ must be in the domain of $P$. Because $fr \notin \mathsf{dom}(P)$, we know $\mathsf{head}(M) \neq fr$. Let $M = n.u_1.\cdots.u_k$ and "$n.x_1.\cdots.x_k \to r$" $\in P$, where $n$ is a name. Then $M[fr := t] = n.u_1[fr := t].\cdots.u_k[fr := t] \to r[\vec{x} := \vec{u}[fr := t]]$. Since $fr$ is not mentioned in $r$, the last term is equal to $r[\vec{x} := \vec{u}][fr := t] = N[fr := t]$.

($\Leftarrow$2). Assume $M[fr := t]\downarrow$. Then $\mathsf{head}(M[fr := t]) \notin \mathsf{dom}(P)$ or $\mathsf{length}(M[fr := t]) \neq \mathsf{arity}(\mathsf{head}(M[fr := t]))$. If $\mathsf{head}(M) = fr$, then $M\downarrow$, so assume $\mathsf{head}(M) \neq fr$. Then $\mathsf{head}(M) = \mathsf{head}(M[fr := t])$ and $\mathsf{length}(M) = \mathsf{length}(M[fr := t])$, so also $M\downarrow$.

($\Rightarrow$2, $\Leftarrow$3, $\Rightarrow$4). Fill in $t = fr$.

($\Rightarrow$3). $\twoheadrightarrow$ is the reflexive and transitive closure of $\to$.

($\Leftarrow$4). Suppose that $M[fr := t] \twoheadrightarrow N\downarrow$ in $k$ steps. If on the contrary $M$ is not terminating, then $M \twoheadrightarrow M'$ in $k+1$ steps. By repeated application of ($\Rightarrow$1), also $M[fr := t] \twoheadrightarrow M'[fr := t]$ in $k+1$ steps. Contradiction. $\square$

*Proof of Lemma 13 (determinism).* We assumed that $M \twoheadrightarrow m.\vec{t}\downarrow$ and $M \twoheadrightarrow n.\vec{u}\downarrow$. So $m.\vec{t}$ and $n.\vec{u}$ are the term at the end of the execution path of $M$; we see that they must be equal. $\square$

*Proof of Proposition 16 ($M \twoheadrightarrow t \twoheadleftarrow N\downarrow$ then $M \twoheadrightarrow N$).* By assumption, $M \twoheadrightarrow t \twoheadleftarrow N$. If $N \twoheadrightarrow t$ in 1 or more steps, then we could not have had $N\downarrow$. Thus $N \twoheadrightarrow t$ in 0 steps: $N = t$. $\square$

## A.2 Program substitution and union

These are proofs of theorems in Section 4.3.

*Proof of Theorem 25, equivalences 1/3.* Let $M = h.t_1.\cdots.t_k$, where $h$ is a name. We have the following cases.

1. $h \notin dom(P)$. All names in $\mathsf{dom}(P\sigma) \setminus \mathsf{dom}(P)$ are fresh, so also $h\sigma \notin \mathsf{dom}(P\sigma)$. We see that both $\mathsf{next}_P(M)$ and $\mathsf{next}_{P\sigma}(M\sigma)$ are undefined.

2. $h \in dom(P)$. Then some rule "$h.x_1.\cdots.x_l \longrightarrow r$" is in $P$, while "$h\sigma.x_1.\cdots.x_l \longrightarrow r\sigma$" is in $P\sigma$. If $k \neq l$, then both $\mathsf{next}_P(M)$ and $\mathsf{next}_{P\sigma}(M\sigma)$ are undefined.

   If $k = l$, then $\mathsf{next}_P(M) = r[\vec{x} := \vec{t}]$, and $\mathsf{next}_{P\sigma}(M\sigma) = r\sigma[\vec{x} := \vec{t}\sigma]$. We note that the domains of $\sigma$ and $[\vec{x} := \vec{t}\sigma]$ are disjoint because names are never variables. We can therefore do the substitutions in parallel: $r\sigma[\vec{x} := \vec{t}\sigma] = r\left[\vec{n} := \vec{m}, \vec{x} := \vec{t}\sigma\right]$. Because the result of $\sigma$ is never in $\mathsf{dom}(\sigma)$ (all $m_i$ are fresh), we can even put $[\vec{n} := \vec{m}]$ at the end: $r\left[\vec{n} := \vec{m}, \vec{x} := \vec{t}\sigma\right] = r\left[\vec{x} := \vec{t}\sigma\right][\vec{n} := \vec{m}] = r\left[\vec{x} := \vec{t}\sigma\right]\sigma$. Also, because the result of $\sigma$ is never in $\mathsf{dom}(\sigma)$ , we know that $\sigma\sigma = \sigma$. We find that $r\left[\vec{x} := \vec{t}\sigma\right]\sigma = r\left[\vec{x} := \vec{t}\right]\sigma = N\sigma$. This completes the proof. □

*Proof of Theorem 25, equivalences 2/4.* The second equivalence is by transitivity of equivalence 1. The fourth equivalence is then trivial. □

*Proof of Theorem 25, implications 1–4.* $\mathsf{next}_P(M)$ exists iff a rule $\in P$ defines it; by $P \subseteq P'$ that rule also defines $\mathsf{next}_{P'}(M)$. This proves implication 1 and 3. The second implication follows using the structure of $M \twoheadrightarrow_P N$. Then the fourth implication follows trivially. □

*Proof of Theorem 25, implication 5.* We have to show that for all $P'' \supseteq P'$ and $X \in \mathscr{U}$, $X.M \twoheadrightarrow\downarrow_{P''} \Longleftrightarrow X.N \twoheadrightarrow\downarrow_{P''}$. This follows from $M \approx_P N$ because $P \subseteq P' \subseteq P''$. □

*Proof of Theorem 25, equivalence 5.* We show the left-implication. The right implication then follows from $M\sigma\sigma^{-1} \approx_{P\sigma\sigma^{-1}} N\sigma\sigma^{-1} \Leftarrow M\sigma \approx_{P\sigma} N\sigma$, because $\sigma\sigma^{-1}$ is the identity substitution.

So suppose $M\sigma \approx_{P\sigma} N\sigma$, and let program $Q \supseteq P$ and term $X$ be given. We prove $X.M \twoheadrightarrow\downarrow_Q \Leftrightarrow X.N \twoheadrightarrow\downarrow_Q$ by the following chain:

$$\begin{array}{lll} & X.M \twoheadrightarrow\downarrow_Q & \\ \Leftrightarrow & X\sigma.M\sigma \twoheadrightarrow\downarrow_{Q\sigma} & \text{(Theorem 25 equivalence 2/3)} \\ \Leftrightarrow & X\sigma.N\sigma \twoheadrightarrow\downarrow_{Q\sigma} & (M \approx_{P\sigma} N \text{and } Q\sigma \supseteq P\sigma) \\ \Leftrightarrow & X.N \twoheadrightarrow\downarrow_Q & \text{(Theorem 25 equivalence 2/3)} \end{array}$$
□

**Lemma 39.** *Assume program $P' \supseteq P$ and $M \in \mathscr{U}$ such that $\mathsf{dom}(P' \setminus P)$ is not mentioned in $P$ or $M$. Then $M \twoheadrightarrow\downarrow_P \Leftrightarrow M \twoheadrightarrow\downarrow_{P'}$.*

*Proof.* Regard $\mathsf{next}_P(M)$ and $\mathsf{next}_{P'}(M)$. The names in $\mathsf{dom}(P' \setminus P)$ are not mentioned in $M$, so either both $\mathsf{next}_P(M)$ and $\mathsf{next}_{P'}(M)$ are defined and equal, or both are undefined. The names in $\mathsf{dom}(P' \setminus P)$ are still not mentioned in $M$'s successor, so the previous sentence applies to all reducts of $M$. We find that $M$ has a final reduct in $P$ iff it has one in $P'$, hence $M \twoheadrightarrow\downarrow_P \Leftrightarrow M \twoheadrightarrow\downarrow_{P'}$. □

*Proof of Theorem 27.* The right-implication is already proven by Theorem 25, so we prove the left-implication.

Suppose program $P' \supseteq P$, but $\mathsf{dom}(P' \setminus P)$ is not mentioned in $M,N$. Suppose furthermore program $Q \supseteq P$ and $X \in \mathscr{U}$. Then we have to prove $X.M \twoheadrightarrow\downarrow_Q \Leftrightarrow X.N \twoheadrightarrow\downarrow_Q$. $Q$ is not required to be a superset of $P'$; it may even define some names differently than $P'$.

Although we know that $\Delta = \mathsf{dom}(P' \setminus P)$ is not used in $M$ or $N$, any name $\in \Delta$ could be used in $X$. We want to compare $X.M$ and $X.N$ on an extension program of $Q$, so we will make sure that $X$ does not accidentally refer to names in $\Delta$. We will rename all $d \in \Delta$ within $X$ and $P'$.

Take a substitution $\sigma = [d_i := d_i' | d_i \in \Delta]$ that renames all $d \in \Delta$ to fresh names for $M,N,X,P',Q$. We know that $M = M\sigma$, $N = N\sigma$, and $P = P\sigma$, because all $d \in \Delta$ are not mentioned in $M$, $N$, or $P$. Now note that $(X.M)\sigma = X\sigma.M$ and $(X.N)\sigma = X\sigma.N$ do not contain a name in $\Delta$, nor does any such name occur in $Q\sigma$.

Take $Q' = P' \cup Q\sigma$. Then $Q'$ is a program because $\mathsf{dom}(Q\sigma \setminus P)$ has no overlap with $\mathsf{dom}(P' \setminus P) = \Delta$. Furthermore, $Q'$ is an extension program of both $P'$ and $Q\sigma$. We apply Lemma 39 to see that

$$\begin{aligned} & X\sigma.M \twoheadrightarrow\downarrow_{Q\sigma} \Leftrightarrow X\sigma.M \twoheadrightarrow\downarrow_{Q'} \\ \text{and}\quad & X\sigma.N \twoheadrightarrow\downarrow_{Q\sigma} \Leftrightarrow X\sigma.N \twoheadrightarrow\downarrow_{Q'}\,. \end{aligned} \tag{2}$$

We can thus make the following series of bi-implications.

$$\begin{array}{rcll} X.M \twoheadrightarrow\downarrow_Q & \Leftrightarrow & X\sigma.M \twoheadrightarrow\downarrow_{Q\sigma} & \text{(Theorem 25)} \\ & \Leftrightarrow & X\sigma.M \twoheadrightarrow\downarrow_{Q'} & (2) \\ & \Leftrightarrow & X\sigma.N \twoheadrightarrow\downarrow_{Q'} & (M \approx_{P'} N, P' \subseteq Q') \\ & \Leftrightarrow & X\sigma.N \twoheadrightarrow\downarrow_{Q\sigma} & (2) \\ & \Leftrightarrow & X.N \twoheadrightarrow\downarrow_Q & \text{(Theorem 25)} \end{array}$$

Because we showed $X.M \twoheadrightarrow\downarrow_Q \Leftrightarrow X.N \twoheadrightarrow\downarrow_Q$, we can conclude that $M \approx_P N$. □

## A.3 Term equivalence

**Proposition 40.** *$=_P$ is an equivalence relation.*

*Proof.* Suppose $A =_P C$ and $C =_P E$, then there exist $B,D$ such that $A \twoheadrightarrow B \twoheadleftarrow C \twoheadrightarrow D \twoheadleftarrow E$. Suppose that $C \twoheadrightarrow B$ in $k$ steps and $C \twoheadrightarrow D$ in $l$ steps. Without loss of generality, $k \leq l$. By determinism of $\rightarrow$,

$$C \underset{k\text{ steps}}{\twoheadrightarrow} B \underset{l-k\text{ steps}}{\twoheadrightarrow} D.$$

Then $A \twoheadrightarrow B \twoheadrightarrow D$. We see that $D$ is a common reduct of $A$ and $E$. □

**Proposition 41.** *$\approx$ is an equivalence relation.*

*Proof.* Reflexivity and symmetry are trivial. We have to prove transitivity: if $M \approx_P N$ and $N \approx_P O$, and $P \subseteq P'$, then $X.M \twoheadrightarrow\downarrow_{P'} \Leftrightarrow X.O \twoheadrightarrow\downarrow_{P'}$. We know from the premises that $X.M \twoheadrightarrow\downarrow_{P'} \Leftrightarrow X.N \twoheadrightarrow\downarrow_{P'}$ and $X.N \twoheadrightarrow\downarrow_{P'} \Leftrightarrow X.O \twoheadrightarrow\downarrow_{P'}$. □

**Lemma 42.** *If $X \twoheadrightarrow Y$, then $X \twoheadrightarrow\downarrow \Leftrightarrow Y \twoheadrightarrow\downarrow$.*

*Proof.* By induction on the number of steps $s$ in $X \twoheadrightarrow Y$. If $X = Y$, then trivial, so assume $s \geq 1$. This implies the existence of term $X'$ such that $X \rightarrow X'$.

If there exists $Z$ such that $Y \twoheadrightarrow Z\downarrow$, then $X \twoheadrightarrow Y \twoheadrightarrow Z\downarrow$. Reversely, assume $X \twoheadrightarrow Z\downarrow$ for some $Z$. Because $X \neq Y$ and by determinism of $\rightarrow$ we know $X \rightarrow X' \twoheadrightarrow Z\downarrow$ and $X \rightarrow X' \twoheadrightarrow Y$. By induction on $X' \twoheadrightarrow Y$ we get $Y \twoheadrightarrow Z\downarrow$. □

*Proof of Proposition 19 ($M \approx N$ then $M \twoheadrightarrow\downarrow \Leftrightarrow N \twoheadrightarrow\downarrow$).* Take a fresh name $X$, and define $P' = P \cup \{X.t \longrightarrow t\}$.

$$\begin{array}{rlll} M \twoheadrightarrow\downarrow_P & \Leftrightarrow & M \twoheadrightarrow\downarrow_{P'} & \text{(evaluation of } M \text{ never contains a head in } \mathsf{dom}(P' \setminus P)) \\ & \Leftrightarrow & X.M \twoheadrightarrow\downarrow_{P'} & (X.M \rightarrow_{P'} M,\ \rightarrow \text{ deterministic}) \\ & \Leftrightarrow & X.N \twoheadrightarrow\downarrow_{P'} & (M \approx_P N) \\ & \Leftrightarrow & N \twoheadrightarrow\downarrow_{P'} & (X.N \rightarrow_{P'} N,\ \rightarrow \text{ deterministic}) \\ & \Leftrightarrow & N \twoheadrightarrow\downarrow_P & \text{(evaluation of } N \text{ never contains a head in } \mathsf{dom}(P' \setminus P)) \end{array}$$

□

**Lemma 43.** *If $X \rightarrow Y$, then $X.\vec{t}\downarrow$ for $k > 0$.*

*Proof.* $\mathsf{next}(M)$ exists iff the length of the corresponding left-hand side is equal to $\mathsf{length}(M)$, and $\mathsf{length}(X.t_1.\cdots.t_k) = \mathsf{length}(X) + k$. The corresponding left-hand side is the same for $X$ and $X.\vec{t}$. □

*Proof of Lemma 18 ($\approx$ is a congruence).* Let $P' \supseteq P$ be an extension program. We must prove that for all $X, X.(M.N) \twoheadrightarrow\downarrow_{P'} \Leftrightarrow X.(M'.N') \twoheadrightarrow\downarrow_{P'}$. Extend $P'$ to $P'' = P' \cup \{A.m.n \longrightarrow X.(m.n), B.n.m \longrightarrow X.(m.n)\}$. Note that by Lemma 42,

$$\begin{array}{rlllll} & X.(M.N) \twoheadrightarrow\downarrow_{P''} & \Leftrightarrow & A.M.N \twoheadrightarrow\downarrow_{P''} & \Leftrightarrow & B.N.M \twoheadrightarrow\downarrow_{P''} \\ \text{and} & X.(M'.N') \twoheadrightarrow\downarrow_{P''} & \Leftrightarrow & A.M'.N' \twoheadrightarrow\downarrow_{P''} & \Leftrightarrow & B.N'.M' \twoheadrightarrow\downarrow_{P''}, \end{array}$$

so we can make the following chain:

$$\begin{array}{rll} & X.(M.N) \twoheadrightarrow\downarrow_{P''} & \\ \Leftrightarrow & A.M.N \twoheadrightarrow\downarrow_{P''} & \\ \Leftrightarrow & A.M.N' \twoheadrightarrow\downarrow_{P''} & (N \approx N') \\ \Leftrightarrow & B.N'.M \twoheadrightarrow\downarrow_{P''} & \\ \Leftrightarrow & B.N'.M' \twoheadrightarrow\downarrow_{P''} & (M \approx M') \\ \Leftrightarrow & X.(M'.N') \twoheadrightarrow\downarrow_{P''}. & \end{array}$$

Now by Lemma 39, $X.(M.N) \twoheadrightarrow\downarrow_{P'} \Leftrightarrow X.(M'.N') \twoheadrightarrow\downarrow_{P'}$, which was to be shown. □

**Lemma 44.** *Let $M, N \in \mathcal{U}$ and $k \geq 0$. Let names $\mathit{fr}_1, \cdots, \mathit{fr}_k$ be not mentioned in $M, N, P$. Suppose $M.\mathit{fr}_1.\cdots.\mathit{fr}_k \rightarrow M' \twoheadrightarrow t \twoheadleftarrow N' \leftarrow N.\mathit{fr}_1.\cdots.\mathit{fr}_k$. Let name $\mathfrak{n}$ be not mentioned in $M, N, P$. Then $\forall X \in \mathcal{U}$ : $X[\mathfrak{n} := M] \twoheadrightarrow\downarrow \Leftrightarrow X[\mathfrak{n} := N] \twoheadrightarrow\downarrow$.*

*Proof.* Suppose that $X[\mathfrak{n} := M] \twoheadrightarrow X'\downarrow$ in $n$ steps. We will show that $X[\mathfrak{n} := N] \twoheadrightarrow\downarrow$. The other direction holds by symmetry. The proof goes by induction on $n$.

Because $\mathsf{next}(M.\mathit{fr}_1.\cdots.\mathit{fr}_k)$ and $\mathsf{next}(N.\mathit{fr}_1.\cdots.\mathit{fr}_k)$ exist, we know that $\mathsf{arity}(M) = \mathsf{arity}(N) = k$. Regard $\mathsf{head}(X)$. We distinguish four cases:

*Case* 1. $\mathsf{head}(X) = \mathfrak{n}$, *and* $\mathsf{length}(X) \neq k$. Then $\mathsf{arity}(X[\mathfrak{n} := N])$ is undefined or not zero, hence $X[\mathfrak{n} := N]\downarrow$.

*Case* 2. $\mathsf{head}(X) = \mathfrak{n}$, *and* $\mathsf{length}(X) = k$. Then there exist $u_1, \cdots, u_k$ such that $X = \mathfrak{n}.u_1.\cdots.u_k$. Then:

$$\begin{aligned} X[\mathfrak{n} := M] &= M.u_1[\mathfrak{n} := M].\cdots.u_l[\mathfrak{n} := M] \\ &\to M'[\vec{fr} := \vec{u}[\mathfrak{n} := M]] && (\vec{fr} \text{ fresh for } M,P) \\ &= M'[\vec{fr} := \vec{u}][\mathfrak{n} := M] && (\mathfrak{n} \text{ fresh for } M') \\ &\twoheadrightarrow t[\vec{fr} := \vec{u}][\mathfrak{n} := M] && (\vec{fr} \text{ fresh for } M,P) \end{aligned}$$

Analogously, $X[\mathfrak{n} := N] \twoheadrightarrow t[\vec{fr} := \vec{u}][\mathfrak{n} := N]$. We know $t[\vec{fr} := \vec{u}][\mathfrak{n} := M] \twoheadrightarrow X' \downarrow$ in at most $n-1$ steps, and $\mathfrak{n}$ is not mentioned in $t[\vec{fr} := \vec{u}]$, so using the induction hypothesis we get $t[\vec{fr} := \vec{u}][\mathfrak{n} := N] \twoheadrightarrow\downarrow$. Hence, $X[\mathfrak{n} := N] \twoheadrightarrow\downarrow$.

*Case* 3. $\mathsf{head}(X) \neq \mathfrak{n}$, *and* $\mathsf{arity}(X) \neq 0$ *or undefined.* Then $\mathsf{arity}(X[\mathfrak{n} := M]) = \mathsf{arity}(X[\mathfrak{n} := N]) = \mathsf{arity}(X) \neq 0$ or undefined, hence $X[\mathfrak{n} := M] \downarrow$ and $X[\mathfrak{n} := N] \downarrow$.

*Case* 4. $\mathsf{head}(X) \neq \mathfrak{n}$, *and* $\mathsf{arity}(X) = 0$. Then $\mathsf{next}(X[\mathfrak{n} := M]) = \mathsf{next}(X)[\mathfrak{n} := M]$ and $\mathsf{next}(X[\mathfrak{n} := N]) = \mathsf{next}(X)[\mathfrak{n} := N]$. We assumed $X[\mathfrak{n} := M] \twoheadrightarrow X' \downarrow$ in $n$ steps, so $\mathsf{next}(X)[\mathfrak{n} := M] \twoheadrightarrow X' \downarrow$ in at most $n-1$ steps. We can therefore apply the induction hypothesis to find $\mathsf{next}(X)[\mathfrak{n} := N] \twoheadrightarrow\downarrow$. ☐

*Proof of Theorem 21.* Suppose $P' \supseteq P$ is an extension program. We must prove $X.M \twoheadrightarrow\downarrow \Leftrightarrow X.N \twoheadrightarrow\downarrow$.

Take $\vec{fr}$ fresh for $P', X, M, N$. Because $\mathsf{arity}(M.fr_1.\cdots.fr_k) = \mathsf{arity}(N.fr_1.\cdots.fr_k) = 0$, they both have a successor, say $M'$ and $N'$. By definition of $=_P$ and determinism of $\to$, we know $M.fr_1.\cdots.fr_k \to M' \twoheadrightarrow t \twoheadleftarrow N' \leftarrow N.fr_1.\cdots.fr_k$. Then by Lemma 44, we know $X.M \twoheadrightarrow\downarrow_{P'} \Leftrightarrow X.N \twoheadrightarrow\downarrow_{P'}$. ☐

*Proof of Theorem 20 ($M \twoheadrightarrow fr.t_1.\cdots.t_k$, $M \approx N$ then $N \twoheadrightarrow fr.u_1.\cdots.u_k$).* Because $fr \notin \mathsf{dom}(P)$, we know $M \twoheadrightarrow fr \downarrow$. By Proposition 19, $N \twoheadrightarrow N' \downarrow$.

Suppose on the contrary that $\mathsf{head}(N') \neq fr$ or $\mathsf{length}(N') \neq k$. We will deduce an impossibility. Make an extension program $P' = P \cup \{fr.x_1.\cdots.x_k \longrightarrow fr.x_1.\cdots.x_k\}$. Then $M \twoheadrightarrow fr.t_1.\cdots.t_k$ is nonterminating under $P'$. But by definition of next, $N' \downarrow_{P'}$. This contradicts with Proposition 19 and Theorem 27, which prove that $N \twoheadrightarrow N'$ is nonterminating. Hence we conclude $N' = fr.u_1.\cdots.u_k$ for some terms $\vec{u}$. ☐

*Proof of Proposition 31.* Let $[\![\mathbb{N}_0]\!] = \{M \in \mathscr{U} \mid \forall z, s \in \mathscr{U} : M.z.s \twoheadrightarrow z\}$ and $[\![\mathbb{N}_{k+1}]\!] = \{M \in \mathscr{U} \mid \exists x \in [\![\mathbb{N}_k]\!]\ \forall z, s \in \mathscr{U} : M.z.s \twoheadrightarrow s.x\}$. Then every $[\![\mathbb{N}_k]\!] \subseteq [\![\mathbb{N}]\!]$, and $\cup_{k\in\mathbb{N}}[\![\mathbb{N}_k]\!]$ satisfies the defining equation of $[\![\mathbb{N}]\!]$, so $[\![\mathbb{N}]\!] = \cup_{k\in\mathbb{N}}[\![\mathbb{N}_k]\!]$.

Suppose $M \in [\![\mathbb{N}]\!]$, then $M \in$ some $[\![\mathbb{N}_k]\!]$. We proceed by induction on $k$. If $k = 0$, then Theorem 21 shows $M \approx \langle 0 \rangle$. If $k \geq 1$, then by the induction hypothesis there is some $x \in [\![\mathbb{N}_{k-1}]\!]$ such that for all $z, s$, $M.z.s \twoheadrightarrow s.x$. Observe that $M.z.s =_P S.x.z.s$ for all $z, s$, so Theorem 21 shows us $M \approx S.x$. We get $S.x \approx S.\langle k-1 \rangle = \langle k \rangle$ by the induction hypothesis and Lemma 18, from which we get the result. ☐