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. We show this 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 reversely.

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 $\mathsf{call/cc}$, to incorporate control [4, 5, 6, 7] or one can move to a calculus like $\lambda\mu$ [10, 2] that allow 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 it. There is also a close connection between proofs in classical logic and control operators, as was first pointed out by [8], who extended the Curry-Howard proofs-as-programs principle to include rules of classical logic. The $\lambda\mu$-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 control is natural to express, without additional operators. We present continuation calculus as an untyped system which is Turing complete. 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 and call-by-name naturally fit into the system.

Continuation calculus looks a bit like term rewriting and a bit like λ-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 terms again. The "dot" is a binary operator that associates to the left. Note that terms do not contain variables, because there are no variables. A *program P* is a list of *program rules* of the form

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

Submitted to:
COS 2013

where the x_i are all different variables and u is a term over variables $x_1 \dots x_k$. 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 don't 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 over 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 we want execution to continue with another code c_2 when $m = S(p)$. We achieve this 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 now represent 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 that to the next function, the continuation. So we are looking for a term *AddCBV* that behaves as follows:

$$AddCBV.\langle m\rangle.\langle p\rangle.r \twoheadrightarrow r.\langle m+p\rangle$$

for all r, where $\langle\cdot\rangle$ is the encoding of natural numbers as terms of CC and $\twoheadrightarrow$ is the multi-step evaluation. This 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 $AddCBV.x.y.r \to t$. 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 $x = Succ.u$, 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 named 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

$$AddCBV.\langle m\rangle.\langle p\rangle.r \twoheadrightarrow r.\langle m+p\rangle,$$
$$B.\langle p\rangle.r.\langle m\rangle \twoheadrightarrow r.\langle m+p+1\rangle.$$

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

$$AddCBN.\langle m\rangle.\langle p\rangle = \langle m+p\rangle.$$

This statement means that $AddCBN.\langle m\rangle.\langle p\rangle$ implements the interface of $\langle m+p\rangle$.

The terms *AddCBV* and *AddCBN* are fundamentally incompatible. Nonetheless, we will see in Section 5.2 how call-by-value and call-by-name functions can be used together. We show additional examples with *FibCBV* and *FibCBN* in Section 5.2. Furthermore, we model and prove a program with call/cc in Sections 6 and 7.

The authors make available a program to evaluate continuation calculus terms on `https://bitbucket.org/bgeron/continuation-calculus-paper/`. Evaluation traces of the examples are included.

2 Formalization

Definition 1 (names)**.** There is a 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. The set of names is assumed infinite when we analyze continuation calculus, although this is not necessary for the evaluation of programs.

Interpretation. Names are used by programs to refer to 'functionality', and will serve the role of constructors, function names, as well 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 not 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 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:

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

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.

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

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 $\mathscr{V}$ 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 some names 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:

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

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

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

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

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 with P.

$$\mathsf{Program} = \mathscr{P}(\mathsf{Rules}) \qquad \text{where } \mathsf{head}(\cdot) \text{ is injective on the LHSes in Program}$$

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

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

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 $\mathsf{next}_P(\cdot) : \mathscr{U} \nrightarrow \mathscr{U}$. 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.

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

It is allowed that $n = 0$:

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

More informally, we write $M \to_P N$ when $\mathsf{next}_P(M) = N$. The reflexive transitive closure of $\to_P$ will be denoted $\twoheadrightarrow_P$. When $M \twoheadrightarrow N$, then we call N a *reduct* of M. When $\mathsf{next}_P(M)$ is undefined, we write 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: $M \to N\downarrow$.

Definition 8 (termination)**.** A term M is said to be *terminating* under a program, notation $M \twoheadrightarrow\downarrow$, when it has a final reduct: $\exists N \in \mathscr{U} : M \twoheadrightarrow N\downarrow$.

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 embeds $c_k(v_1,\cdots,v_{n_k})$ of a data type with m constructors, then $\forall t_1 \ldots t_m : d.\vec{t} \twoheadrightarrow t_k.\vec{v}$. Examples:

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

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

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

  Note that $AddCBN.(S.Zero).(S.(S.Zero))$ is a data term embedding 3.
  This shows that $1 +_{\mathrm{CBN}} 2 = S(x)$, for some x embedded 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 are 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

$$Zero.z.s \longrightarrow z$$
$$S.m.z.s \longrightarrow s.m$$
$$Nil.ifempty.iflist \longrightarrow ifempty$$
$$Cons.n.l.ifempty.iflist \longrightarrow iflist.n.l$$

Call-by-value functions

$$AddCBV.x.y.r \longrightarrow x.(r.y).(AddCBV'.y.r)$$
$$AddCBV'.y.r.x \longrightarrow AddCBV.x.(S.y).r$$

$$FibCBV.x.r \longrightarrow x.(r.Zero).(FibCBV_1.r)$$
$$FibCBV_1.r.y \longrightarrow y.(r.(S.Zero)).(FibCBV_2.r.y)$$
$$FibCBV_2.r.y.y' \longrightarrow FibCBV.y.(FibCBV_3.r.y')$$
$$FibCBV_3.r.y'.fib_y \longrightarrow FibCBV.y'.(FibCBV_4.r.fib_y)$$
$$FibCBV_4.r.fib_y.fib_{y'} \longrightarrow AddCBV.fib_y.fib_{y'}.r$$

Call-by-name functions

$$AddCBN.x.y.z.s \longrightarrow x.(y.z.s).(AddCBN'.y.s)$$
$$AddCBN'.y.s.x' \longrightarrow s.(AddCBN.x'.y)$$

$$FibCBN.x.z.s \longrightarrow x.z.(FibCBN_1.z.s)$$
$$FibCBN_1.z.s.y \longrightarrow y.(s.Zero).(FibCBN_2.z.s.y)$$
$$FibCBN_2.z.s.y.y' \longrightarrow AddCBN.(FibCBN.y).(FibCBN.y').z.s$$

Figure 1: Continuation calculus embeddings of $+$ and fib. The functions are applied in a different way, as shown in Figure 2. This incompatibility is already indicated by the different arity: $\mathsf{arity}(AddCBV) = 3 \neq \mathsf{arity}(AddCBN) = 4$, and $\mathsf{arity}(FibCBV) = 2 \neq \mathsf{arity}(FibCBN) = 3$. 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 fr does not occur in the program under consideration, then we call fr a *fresh name*. Furthermore, all fresh names that we assume within theorems, lemmas, and propositions are understood to be different. When we say fr is fresh *for some objects*, then it is additionally required that 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 fr be fresh. The following equivalences hold:*

$$M \twoheadrightarrow N \Longleftrightarrow \forall t \in \mathscr{U} : M[fr := t] \twoheadrightarrow N[fr := t]$$
$$M \twoheadrightarrow\!\downarrow \Longleftrightarrow \exists t \in \mathscr{U} : M[fr := t] \twoheadrightarrow\!\downarrow$$

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 β-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. M is *terminating* in P when the reduction path of M is finite: $\exists t : M \twoheadrightarrow_P t\downarrow_P$.
Notation: $M \twoheadrightarrow\downarrow_P$.

Definition 18. 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: $FibCBV.\langle m\rangle.\langle 0\rangle \approx FibCBV.\langle 0\rangle.\langle m\rangle$ and $Zero \approx Nil$, but $Zero \not\approx S.Zero$.

Lemma 19. *$\approx$ 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*; we call M *nonterminating*. When the program is extended, term 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 $\exists N \in \mathscr{U} : M \twoheadrightarrow N\downarrow$.

- *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.

- *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 this.

Proposition 20. *If $M \approx N$, then $M \twoheadrightarrow\downarrow \Leftrightarrow N \twoheadrightarrow\downarrow$.*

Theorem 21. *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 $M \approx N$.

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

The generalization goes as follows.

Theorem 23. *Let M,N be terms with arity k, and names $\vec{fr}$ be fresh. If $M.fr_1.\cdots.fr_k =_P N.fr_1.\cdots.fr_k$, then $M \approx N$.*

Remark 24. $M \twoheadrightarrow N$ does not always imply $M \approx N$ if $\mathsf{arity}(N) > 0$. For instance, take the following program:

$$Goto.x \longrightarrow x$$
$$Omega.x \longrightarrow x.x$$

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 1 '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 not $Goto.Omega \approx Omega$.

4.3 Program substitution and union

Definition 25 (fresh substitution)**.** Let $n_1 \ldots n_k$ be names, and $m_1 \ldots m_k$ be fresh for M, hence 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 one of them, and taking the union. As the following theorems shows, this keeps most interesting properties.

Theorem 26. *Suppose that $P' \supseteq P$ is an extension program, and M,N are terms. Then the left hand equations hold. Let σ denote a fresh substitution $[\vec{n} := \vec{m}]$. Then the right hand equations hold.*

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

Remark 27. Names $\vec{n}$ are not mentioned in $M\sigma$ and $P\sigma$, so we can apply Theorem 26 with σ^{-1} on $M\sigma$ and $P\sigma$.

Theorem 28. *Suppose that P' extends P, but $\mathsf{dom}(P' \setminus P)$ are not mentioned in M or N.*
Then $M \approx_P N \Longleftrightarrow M \approx_{P'} N$.

5 Data terms and functions

In this section, we give an embedding of some standard data in continuation calculus. We define the data terms by interface: a term represents a certain boolean, natural, or list if extensions of that term have a specified reduct. There is no upper bound on the distance of this reduct: Figure 2 shows a computation that takes 41 steps on one data term, and 304 steps on an observationally equivalent data term.

Later in this section, we introduce two kinds of function terms. We call them *call-by-name* and *call-by-value*, by analogy with the concepts in lambda calculus.

5.1 Data terms interface

The booleans have the simplest interface. Term M belongs to the embedded booleans iff it satisfies one of two conditions: $\forall t, f \in \mathscr{U} : M.t.f \twoheadrightarrow t$ or $\forall t, f \in \mathscr{U} : M.t.f \twoheadrightarrow f$. If M satisfies the first condition, then M models True; if M satisfies the second condition, then M models False.

Definition 29. We may write $\langle \mathsf{True} \rangle$ or $\langle \mathsf{False} \rangle$ for a term that satisfies the respective condition. We will say that M *embeds* True or False, respectively.

There are three observations to be made about $[\![\mathbb{B}]\!]$. First, it is important that always $\langle \mathsf{True} \rangle \not\approx \langle \mathsf{False} \rangle$.

Proof. Take fresh names t, f, then $t \neq f$. Then $\langle \mathsf{True} \rangle.t.f \twoheadrightarrow t\downarrow$ and $\langle \mathsf{False} \rangle.t.f \twoheadrightarrow f\downarrow$. By contraposition of Theorem 21, we know $\langle \mathsf{True} \rangle \not\approx \langle \mathsf{False} \rangle$. □

Second, it is sufficient to regard *fresh names*: it is equivalent to define $[\![\mathbb{B}]\!]$ as all M such that $M.t.f \twoheadrightarrow t$ or $M.t.f \twoheadrightarrow f$. In such definitions, t and f should be understood not mentioned in the program nor in M. If t, f are not introduced, we will intend them fresh. The new definition for $[\![\mathbb{B}]\!]$ reads as follows:

$$[\![\mathbb{B}]\!] = \{M \in \mathscr{U} \,|\, M.t.f \twoheadrightarrow t \vee M.t.f \twoheadrightarrow f\} \tag{1}$$

Third, and perhaps most importantly: the set of booleans depends on the program. $\mathscr{U}$ may contain no booleans, two booleans, or infinitely many booleans. But assume the following program fragment:

$$\begin{aligned} \mathit{True}.t.f &\longrightarrow t \\ \mathit{False}.t.f &\longrightarrow f \end{aligned}$$

Then the set of embedded booleans $[\![\mathbb{B}]\!]$ contains both *True* and *False*. (To create more booleans, one may define *True*′ and *False*′ similarly.) Furthermore, Theorem 23 shows that all $M \in [\![\mathbb{B}]\!]$ are observationally equivalent to *True* or *False*.

We will now define and give meaning to the term sets $[\![\mathbb{N}]\!]$ and $[\![\mathsf{List}_{\mathbb{N}}]\!]$ that contain embeddings of natural numbers and lists of natural numbers.

Naturals numbers We define $[\![\mathbb{N}]\!]$ as the least set satisfying:

$$[\![\mathbb{N}]\!] = \{M \in \mathscr{U} \,|\, M.z.s \twoheadrightarrow z \vee \exists n \in [\![\mathbb{N}]\!] : M.z.s \twoheadrightarrow s.n\}$$

By a slight abuse of notation, we postulate that n shall not depend on z or s. The unabridged definition reads:

$$[\![\mathbb{N}]\!] = \{M \in \mathscr{U} \,|\, (\forall z,s \in \mathscr{U} : M.z.s \twoheadrightarrow z) \vee \exists n \in [\![\mathbb{N}]\!]\ \forall z,s \in \mathscr{U} : M.z.s \twoheadrightarrow s.n\}$$

If $n \in \mathbb{N}$, then we may write $\langle n \rangle$ for a term that satisfies the appropriate condition. All classes $\langle n \rangle$ are implementable by the following fragment.

$$\begin{aligned} \mathit{Zero}.z.s &\longrightarrow z \qquad && \text{such that } \mathit{Zero} = \langle 0 \rangle \\ S.n.z.s &\longrightarrow s.n \qquad && \text{such that } S.\langle n \rangle = \langle n+1 \rangle \end{aligned}$$

Proposition 30. *All terms in* $[\![\mathbb{N}]\!]$ *are observationally equivalent to* $\underbrace{S.(\cdots.(S}_{k \text{ times}}.\mathit{Zero})\cdots)$ *for some* k.

The proof is in the appendix.

Lists of naturals We define $[\![\mathsf{List}_{\mathbb{N}}]\!]$ as the least set satisfying:

$$[\![\mathsf{List}_{\mathbb{N}}]\!] = \{M \in \mathscr{U} \,|\, M.e.c \twoheadrightarrow e \vee \exists x \in [\![\mathbb{N}]\!], xs \in [\![\mathsf{List}_{\mathbb{N}}]\!] : M.e.c \twoheadrightarrow c.x.xs\}$$

If $l \in [\![\mathsf{List}_{\mathbb{N}}]\!]$, then we may write $\langle l \rangle$ for a term that satisfies the appropriate condition.

Remark 31. We may write multiple $\langle \cdot \rangle$ in composite expressions or phrases, for example:

$$a.\langle b \rangle.M \twoheadrightarrow c.\langle d \rangle \twoheadrightarrow \langle e \rangle.N$$

In this expression, $\langle b \rangle, \langle d \rangle, \langle e \rangle$ stand for terms embedding the mathematical objects b, d, e. However, there are multiple terms that can be used for $\langle b \rangle, \langle d \rangle, \langle e \rangle$. The occurrence $\langle b \rangle$ in the leftmost expression $a.\langle b \rangle.M$ should be understood universally quantified over those terms. The occurrences $\langle d \rangle$ and $\langle e \rangle$ in the rest of the phrase are existentially quantified, and may depend on the choice of $\langle b \rangle$.

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:* $\quad$ *FibCBV*.7.*fr* $\twoheadrightarrow$ $\mathit{FibCBV}_1.\mathit{fr}.6$ in 2 steps $\twoheadrightarrow$ $\mathit{FibCBV}_2.\mathit{fr}.6.5$ in 2 steps $\to$ $\mathit{FibCBV}.6.(\mathit{FibCBV}_3.\mathit{fr}.5)$ $\twoheadrightarrow$ $\mathit{FibCBV}_3.\mathit{fr}.5.8$ in 210 steps $\to$ $\mathit{FibCBV}.5.(\mathit{FibCBV}_4.\mathit{fr}.8)$ $\twoheadrightarrow$ $\mathit{FibCBV}_4.\mathit{fr}.8.5$ in 119 steps $\to$ *AddCBV*.8.5.*fr* $\twoheadrightarrow$ *fr*.13 in 26 steps	*By the specification of FibCBN and Proposition 30, we know* $\mathit{FibCBN}.7 \approx 13$.

Both 13 and $\mathit{FibCBN}.7$ can be used in other functions, like $+$.

$13 +_{CBV} 0$ *is obtained in 41 steps:* $\quad$ *AddCBV*.13.0.*fr* $\twoheadrightarrow$ *fr*.13 in 41 steps	$\mathit{FibCBN}.7 +_{CBV} 0$ *is obtained in 304 steps:* $\quad$ $\mathit{AddCBV}.(\mathit{FibCBN}.7).0.\mathit{fr}$ $\twoheadrightarrow$ *fr*.13 in 304 steps
$0 +_{CBV} 13$ *is obtained in 2 steps:* $\quad$ *AddCBV*.0.13.*fr* $\to$ $0.(\mathit{fr}.13).(\mathit{AddCBV}'.13.\mathit{fr})$ $\to$ *fr*.13	$0 +_{CBV} \mathit{FibCBN}.7$ *is obtained in 2 steps:* $\quad$ $\mathit{AddCBV}.0.(\mathit{FibCBN}.7).\mathit{fr}$ $\to$ $0.(\mathit{fr}.(\mathit{FibCBN}.7)).(\mathit{AddCBV}'.(\mathit{FibCBN}.7).\mathit{fr})$ $\to$ $\mathit{fr}.(\mathit{FibCBN}.7)$
$13 +_{CBN} 0$ *is* $\mathit{AddCBN}.13.0$*, without reduction. Analogous for* $0 +_{CBN} 13$.	$\mathit{FibCBN}.7 +_{CBN} 0$ *is* $\mathit{AddCBN}.(\mathit{FibCBN}.7).0$*, without reduction. Analogous for* $0 +_{CBN} \mathit{FibCBN}.7$.

Figure 2: Calculating $\mathsf{fib}(7)$, $\mathsf{fib}(7)+0$, and $0+\mathsf{fib}(7)$ using all combinations of call-by-value and call-by-name. The call-by-value result $\mathit{fib}(7)$ takes a lot of steps to obtain, whereas the call-by-name result is direct. However, the CBV sum $\mathit{fib}(7)+0$ with is only 'slow' for the CBN result. Effectively, the CBN $\mathit{fib}(7)$ delays computation until it is needed. A decimal number n stands for $\underbrace{S.(\cdots.(S}_{n \text{ times}}.\mathit{Zero})\cdots)$.

5.2 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 concepts in 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. We illustrate this by deriving a function *AddCBN*. We use an algorithm that allows us to postpone as much computation as possible:

  $$0+p = p$$
  $$S(m)+p = S(m+p)$$

  As *AddCBN* will be a call-by-name function, we require that $AddCBN.x.y \in [\![\mathbb{N}]\!]$. We use the following specification:

  $$AddCBN.\langle m\rangle.\langle p\rangle = \langle m+p\rangle$$

  All terms in $[\![\mathbb{N}]\!]$ have arity 2, so the rule for *AddCBN* will be of the shape $AddCBN.x.y.z.s \longrightarrow t$. Furthermore, we require

  $$AddCBN.\langle 0\rangle.\langle 0\rangle.z.s \twoheadrightarrow z \text{ and}$$
  $$AddCBN.\langle S(m)\rangle.\langle p\rangle.z.s \twoheadrightarrow s.\langle m+p\rangle.$$

  The specification of $\langle p\rangle$ helps us here:

  $$\langle 0\rangle.z.s \twoheadrightarrow z$$
  $$\langle S(p)\rangle.z.s \twoheadrightarrow s.\langle p\rangle$$

  Note that $\langle p\rangle.z.s$ fulfills the requirement on $AddCBN.\langle 0\rangle.\langle p\rangle.z.s$. A first estimate is as follows:

  $$AddCBN.x.y.z.s \longrightarrow x.(y.z.s).(s.(AddCBN.x'.y))$$

  But 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 appended to $s.(AddCBN.x'.y)$. We fix this with a helper name:

  $$AddCBN.x.y.z.s \longrightarrow x.(y.z.s).(AddCBN.y.s))$$
  $$AddCBN.y.s.x' \longrightarrow s.(AddCBN.x'.y).$$

  This fulfills the specification, which means $AddCBN.\langle m\rangle.\langle p\rangle$ is a data term. We have a similar specification for *FibCBN*:

  $$FibCBN.\langle m\rangle = \langle \mathsf{fib}(m)\rangle$$

- *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:

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

  When a function's definition has "nested function calls", e.g. $\mathsf{fib}(y)+\mathsf{fib}(y')$ for fib on inputs ≥ 2, then a call-by-value function definition typically needs administrative names to prepare appropriate return continuations r. In the example, $FibCBV_2$ prepares for $\mathsf{fib}(y)$, $FibCBV_3$ prepares for $\mathsf{fib}(y')$, and $FibCBV_4$ reorders the results appropriately for $fib_y + fib_{y'}$. Such administrative names are not necessary when using only call-by-name functions.

We leave the 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. We use a call-by-value 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:

$$\begin{array}{l}\mathsf{let\ rec}\ \mathit{listmult}_1 = \\ \quad \mid [] \to 1 \\ \quad \mid (x : xs) \to x \cdot \mathit{listmult}_1\ xs\end{array}$$

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

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.

$$\begin{array}{l}\mathsf{let\ rec}\ \mathit{listmult}_2 = \\ \quad \mid [] \to 1 \\ \quad \mid (x : xs) \to \mathsf{match}\ x\ \mathsf{with} \\ \qquad \mid 0 \to 0 \\ \qquad \mid x' + 1 \to x \cdot \mathit{listmult}_2\ l\end{array}$$

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

However, $\mathit{listmult}_2$ is not so efficient either: if only the last number in the list is zero, then we skip exactly 1 multiplication. The other multiplications are all of the form $0 \cdot n = 0$. 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.

$$\begin{array}{l}\mathsf{let}\ \mathit{listmult}_3\ l = \\ \quad \mathsf{call/cc}\ (\lambda \mathit{abort}. \\ \qquad A\ l \\ \qquad \mathsf{where}\ A = \\ \qquad\quad \mid [] \to 1 \\ \qquad\quad \mid (x : xs) \to \boxed{\begin{array}{l}\mathsf{match}\ x\ \mathsf{with} \\ \quad \mid 0 \to \mathit{abort}\ 0 \\ \quad \mid x' + 1 \to \boxed{x \cdot A\ xs}_{(C)}\end{array}}_{(B)})\end{array}$$

The boxes are not syntax, but are used to relate $\mathit{ListMult}_3$ to Figure 3.

While $\mathit{listmult}_3$ 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 $\mathit{listmult}_3$, with one exception: here the variable *abort* corresponds to the partial application of *abort* to 0 in $\mathit{listmult}_3$. The variable r 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. That is: if program P contains the listed definitions, and we assume that *Mult* behaves according to the specification, then the specification of *ListMult* holds: $\mathit{ListMult}.\langle l \rangle.r \twoheadrightarrow r.\langle \mathsf{product}\ l \rangle$ for all $l \in \mathsf{List}_{\mathbb{N}}, r \in \mathscr{U}$.

The proof consists of two parts. First, we prove that name A conforms to its specification. This is done by induction on l. The proof requires Lemma 36. Afterwards, it is straightforward to prove *ListMult* correct using the specification of A.

Continuation calculus

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

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

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

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

$PostMult.x.r.y \longrightarrow Mult.x.y.r$
$\Longrightarrow \quad PostMult.\langle x\rangle.r.\langle y\rangle \twoheadrightarrow r.\langle x \cdot y\rangle$

$Mult.x.y.r \longrightarrow y.(r.Zero).(PostMult.x.(PostAdd.x.r))$
Assumption. $Mult.\langle x\rangle.\langle y\rangle.r \twoheadrightarrow r.\langle x \cdot y\rangle$

Usage. $ListMult.\langle [3,1,2]\rangle.r \twoheadrightarrow r.\langle 6\rangle$

Haskell equivalent

— *Assume* $l, xs \in [\mathsf{Nat}]$, $x, x', y \in \mathsf{Nat}$.
$listmult_4\ l\ r = A\ l\ r\ (r\ 0)$
$\Longrightarrow \quad listmult_4\ l\ r = r\ (\mathsf{product}\ l)$

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

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

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

$Postmult\ x\ r\ y = r\ (x \cdot y)$

Usage. $6 == listmult_4\ [3,1,2]\ id$

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.

Lemma 32. *The specification of A is satisfied. That is, assume* $l \in \mathsf{List}_{\mathbb{N}}$, $r, r_0 \in \mathscr{U}$ *such that* $r.Zero =_P r_0$. *Then* $A.\langle l\rangle.r.r_0 =_P r.\langle product\ l\rangle$.

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

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

	$A.\langle []\rangle.r.r_0$	
$\to$	$\langle []\rangle.(r.(S.Zero)).(B.r.r_0)$	by definition
$\twoheadrightarrow$	$r.(S.Zero)$	by definition of $\langle []\rangle$
$=$	$r.\langle \mathsf{product}\ []\rangle$	$S.Zero$ embeds $1 = \mathsf{product}\ []$

Case 2. $l = (0 : xs)$. Then:

	$A.\langle 0 : xs\rangle.r.r_0$	
$\to$	$\langle 0 : xs\rangle.(r.(S.Zero)).(B.r.r_0)$	by definition of A
$\twoheadrightarrow$	$B.r.r_0.\langle 0\rangle.\langle xs\rangle$	by definition of $\langle 0 : xs\rangle$
$\to$	$\langle 0\rangle.r_0.(C.r.r_0.\langle 0\rangle.\langle xs\rangle)$	by definition of B
$\twoheadrightarrow$	r_0	by definition of $\langle 0\rangle$
$=_P$	$r.Zero$	by assumption
$=$	$r.\langle \mathsf{product}\ (0 : xs)\rangle$	$Zero$ embeds $0 = \mathsf{product}\ (0 : xs)$

Case 3. $l = (x'+1 : xs)$. Then:

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

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

The third case requires Lemma 36, which is proved below. This completes the induction, yielding:

$$A.\langle l\rangle.r.r_0 =_P r.\langle \mathsf{product}\ l\rangle \text{ for all } l, r\ .$$ □

Lemma 33. *Let $x' \in \mathbb{N}$, $xs \in \mathsf{List}_{\mathbb{N}}$, $r, r_0 \in \mathscr{U}$ and $r.Zero =_P r_0$. Then $PostMult.\langle x'+1\rangle.r.Zero =_P r_0$.*

Proof. By the following chain.

	$PostMult.\langle x'+1\rangle.r.Zero$	
$\to$	$Mult.\langle x'+1\rangle.Zero.r$	by definition of *PostMult*
$\to$	$Zero.\,(r.Zero)\,.\,(PostMult.\langle x'+1\rangle.\,(PostAdd.\langle x'+1\rangle.r))$	by definition of *Mult*
$\to$	$r.Zero$	by definition of *Zero*
$=_P$	r_0	by assumption □

Theorem 34. *The specification of ListMult is satisfied. That is, assume $l \in \mathsf{List}_{\mathbb{N}}$, $r \in \mathscr{U}$. Then $ListMult.\langle l\rangle.r \twoheadrightarrow r.\langle product\ l\rangle$.*

Proof. We fill in $r_0 = r.Zero$ in the specification of A; $r.Zero =_P r.Zero$ is trivially satisfied.

$$A.\langle l\rangle.r.\,(r.Zero) =_P r.\langle \mathsf{product}\ l\rangle \text{ for all } l, r\ .$$

If we temporarily take r to be a fresh name, then Proposition 16 shows

$$A.\langle l\rangle.r.\,(r.Zero) \twoheadrightarrow r.\langle \mathsf{product}\ l\rangle \text{ for all } l\ .$$

However, Theorem 12 allows us to generalize this again:

$$A.\langle l\rangle.r.\,(r.Zero) \twoheadrightarrow r.\langle \mathsf{product}\ l\rangle \text{ for all } l, r\ .$$

Now our main correctness result follows rather straightforwardly.

	$ListMult.\langle l\rangle.r$	
$\to$	$A.\langle l\rangle.r.\,(r.Zero)$	by definition
$\twoheadrightarrow$	$r.\langle \mathsf{product}\ l\rangle$	as just shown. □

8 Conclusions and Future work

We have defined a simple deterministic calculus that is suitable for modeling programs with control, and for modeling both call-by-value and call-by-name programs in an integrated way, that allows combininng CBN and CBV subprograms. 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 integrated 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 encoding 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: *ICALP*, *LNCS* 2719, Springer, pp. 871–885.

[2] Zena M. Ariola & Hugo Herbelin (2008): *Control Reduction Theories: the Benefit of Structural Substitution*. *Journal of Functional Programming* 18(3), pp. 373–419.

[3] O. Danvy & A. Filinski (1992): *Representing Control: A Study of the CPS Transformation*. *Mathematical Structures in Computer Science* 2(4), pp. 361–391.

[4] R Kent Dyvbig, Simon Peyton Jones & Amr Sabry (2007): *A monadic framework for delimited continuations*. *Journal of Functional Programming* 17(6), pp. 687–730, doi:10.1017/S0956796807006259.

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

[6] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker & Bruce F. Duba (1987): *A Syntactic Theory of Sequential Control*. *Theoretical Computer Science* 52, pp. 205–237.

[7] Mattias 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.

[8] Timothy G. Griffin (1990): *A Formulae-as-Types Notion of Control*. In: *POPL*, ACM, pp. 47–58.

[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, pp. 228–243, doi:10.1007/3-540-48959-2_17.

[10] Michel Parigot (1992): *$\lambda\mu$-calculus: An Algorithmic Interpretation of Classical Natural Deduction*. In: *Logic Programming and Automated Reasoning, LPAR'92*, *LNCS* 624, pp. 190–201.

[11] G.D. Plotkin (1975): *Call-by-name, call-by-value and the λ-calculus*. *Theoretical computer science* 1(2), pp. 125–159.

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 35. *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 $\mathsf{next}(M)$. □

Theorem 36. *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.

(⇐1). Fill in $t = fr$.

(⇒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 $r[\vec{x} := \vec{u}][fr := t] = N[fr := t]$.

(⇐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 $M\downarrow$.

(⇒2, ⇐3, ⇒4). Fill in $t = fr$.

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

(⇐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 (⇒1), also $M[fr := t] \twoheadrightarrow M'[fr := t]$ in $k+1$ steps. Contradiction. □

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. □

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: $N = t$. □

A.2 Program substitution and union

These are proofs of theorems in Section 4.3.

Proof of Theorem 26, equivalences 1/3. Let $M = h.t_1.\cdots.t_k$, where h is a name. We have the following cases.

1. $h \notin \mathit{dom}(P)$. 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. $h \in \mathit{dom}(P)$. 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 σ 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 σ 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 σ 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 26, equivalences 2/4. The second equivalence is by transitivity of equivalence 1. The fourth equivalence is then trivial. □

Proof of Theorem 26, 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 26, implication 5. We have to show that for all $P'' \supseteq P'$ and $X \in \mathscr{U}$, $X.M \twoheadrightarrow\!\!\downarrow_{P''} \iff X.N \twoheadrightarrow\!\!\downarrow_{P''}$. This follows from $M \approx_P N$ because $P \subseteq P' \subseteq P''$. □

Proof of Theorem 26, 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:

$$
\begin{array}{lll}
 & X.M \twoheadrightarrow\!\!\downarrow_Q & \\
\Leftrightarrow & X\sigma.M\sigma \twoheadrightarrow\!\!\downarrow_{Q\sigma} & \text{(Theorem 26 equivalence 2/3)} \\
\Leftrightarrow & X\sigma.N\sigma \twoheadrightarrow\!\!\downarrow_{Q\sigma} & (M \approx_{P\sigma} N \text{and } Q\sigma \supseteq P\sigma) \\
\Leftrightarrow & X.N \twoheadrightarrow\!\!\downarrow_Q & \text{(Theorem 26 equivalence 2/3)}
\end{array}
$$

□

Lemma 37. *Assume program $P' \supseteq P$ and $M \in \mathscr{U}$ such that $\mathit{dom}(P' \setminus P)$ 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 28. The right-implication is already proven by Theorem 26, 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 Δ. 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 Δ, 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 40 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.

$$\begin{array}{lcll} X.M \twoheadrightarrow\!\downarrow_Q & \Leftrightarrow & X\sigma.M \twoheadrightarrow\!\downarrow_{Q\sigma} & \text{(Theorem 26)} \\ & \Leftrightarrow & X\sigma.M \twoheadrightarrow\!\downarrow_{Q'} & (2) \\ & \Leftrightarrow & X\sigma.N \twoheadrightarrow\!\downarrow_{Q'} & (M \approx_{P'} N, P' \subseteq Q') \\ & \Leftrightarrow & X\sigma.N \twoheadrightarrow\!\downarrow_{Q\sigma} & (2) \\ & \Leftrightarrow & X.N \twoheadrightarrow\!\downarrow_Q & \text{(Theorem 26)} \end{array}$$

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 38. *$=_P$ 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 $\to$,

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

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

Proposition 39. *$\approx$ 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 40. *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 $X \to X'$.

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 $\to$ we know $X \to X' \twoheadrightarrow Z\downarrow$ and $X \to X' \twoheadrightarrow Y$. By induction on $X' \twoheadrightarrow Y$ we get $Y \twoheadrightarrow Z\downarrow$. □

Proof of Proposition 20 ($M \approx N$ then $M \twoheadrightarrow\downarrow \Leftrightarrow N \twoheadrightarrow\downarrow$). Take a fresh name X, and define $P' = P \cup \{X.t \longrightarrow t\}$.

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

□

Lemma 41. *If $X \to Y$, 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 19 ($\approx$ 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 43,

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

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

Proof of Proposition 22. Implied by Theorem 23, which is proven shortly. □

Lemma 42. *Let $M, N \in \mathscr{U}$ and $k \geq 0$. Let names $fr_1, \cdots, fr_k$ be not mentioned in M, N, P. Suppose $M.fr_1.\cdots.fr_k \to M' \twoheadrightarrow t \twoheadleftarrow N' \leftarrow N.fr_1.\cdots.fr_k$. Let name $\mathfrak{n}$ be not mentioned in M, N, P. Then $\forall X \in \mathscr{U}$: $X[\mathfrak{n} := M] \twoheadrightarrow\downarrow \Leftrightarrow X[\mathfrak{n} := N] \twoheadrightarrow\downarrow$.*

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 $\mathsf{next}(M.fr_1.\cdots.fr_k)$ and $\mathsf{next}(N.fr_1.\cdots.fr_k)$ 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 23. Suppose $P' \supseteq P$ is an extension program. We must prove $X.M \twoheadrightarrow\downarrow \Leftrightarrow X.N \twoheadrightarrow\downarrow$.

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 $M.fr_1.\cdots.fr_k \to M' \twoheadrightarrow t \twoheadleftarrow N' \leftarrow N.fr_1.\cdots.fr_k$. Make a further extension program $P'' = P' \cup \{Y.m \longrightarrow X.m\}$ for some fresh name Y. Then by Lemma 45, we know $Y.M \twoheadrightarrow\downarrow_{P''} \Leftrightarrow Y.N \twoheadrightarrow\downarrow_{P''}$. By determinism of $\to$, we know $X.M \twoheadrightarrow\downarrow_{P''} \Leftrightarrow X.N \twoheadrightarrow\downarrow_{P''}$, and by Lemma 40, we know $X.M \twoheadrightarrow\downarrow_{P'} \Leftrightarrow X.N \twoheadrightarrow\downarrow_{P'}$. □

Proof of Theorem 21 ($M \twoheadrightarrow fr.t_1.\cdots.t_k$, $M \approx N$ then $N \twoheadrightarrow fr.u_1.\cdots.u_k$). Because $fr \notin \mathsf{dom}(P)$, we know $M \twoheadrightarrow fr\downarrow$. By Proposition 20, $N \twoheadrightarrow N'\downarrow$.

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 20 and Theorem 28, 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 30. Let $[\![\mathbb{N}_0]\!] = \{M \in \mathscr{U} \mid M.z.s \twoheadrightarrow z\}$ and $[\![\mathbb{N}_{k+1}]\!] = \{M \in \mathscr{U} \mid \exists n \in [\![\mathbb{N}_k]\!] : M.z.s \twoheadrightarrow s.n\}$. 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 Proposition 22 shows $M \approx Zero$. If $k \geq 1$, there is some $n \in [\![\mathbb{N}_{k-1}]\!]$ such that for all z, s, $M.z.s \twoheadrightarrow s.n$. Observe that $M.z.s =_P S.n.z.s$ for all z, s, so Theorem 23 shows us $M \approx S.n$. We get $S.n \approx S.(\underbrace{S.(\cdots.(S}_{k-1 \text{ times}}.Zero)\cdots))$ by the induction hypothesis, which gives us the result. □