Context-based Proofs of Termination for Typed Delimited-control Operators

Małgorzata Biernacka Dariusz Biernacki

Institute of Computer Science
University of Wrocław
Wrocław, Poland
{mabi,dabi}@ii.uni.wroc.pl

Abstract

We present direct proofs of termination of evaluation for typed delimited-control operators *shift* and *reset* using a variant of Tait's method with context-based reducibility predicates. We address both call by value and call by name, and for each reduction strategy we consider a type-and-effect system à la Danvy and Filinski as well as a system with a fixed answer type. The call-by-value type-and-effect system we present is a refinement of Danvy and Filinski's original type system, whereas the call-by-name type-and-effect system is new. From the normalization proofs, we extract call-by-value and call-by-name evaluators in continuation-passing style with two layers of continuations; by construction, these evaluators are instances of normalization by evaluation.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives

General Terms Languages, Theory

Keywords Delimited Continuations, Reduction Semantics, Type System, Reducibility Predicates, Continuation-Passing Style

1. Introduction

Static delimited continuations, accessible through the control operators *shift* and *reset*, were introduced by Danvy and Filinski for expressing the success-failure continuation model of backtracking in direct style [18, 19] and they have found numerous practical and theoretical applications ever since [12, 15, 20, 21, 24, 25, 32, 35, 36, 39]. While delimited continuations are gaining currency for practitioners and implementors, their theoretical foundations are also being actively studied and developed [1–4, 10, 27, 28, 30, 31]. Recently, Ariola et al. [1] and Kameyama and Asai [29] have proved strong normalization of, respectively, several monomorphic calculi and a polymorphic calculus for static delimited continuations by using reduction-preserving CPS translations to the corresponding pure calculi.

In this article, our goal is to prove termination of evaluation for *shift* and *reset* directly, using a variant of Tait's method of reducibility predicates [37, 40] for both call by value and call by name. Specifically, we adapt the method that we have proposed in an earlier work and successfully applied to the simply typed lambda calculus enriched with abortive control operators *callcc*, *abort* and Felleisen's $\mathcal{C}$ [9]. Our context-based approach takes as the starting point a reduction semantics (i.e., a small-step operational semantics with explicit representation of evaluation contexts [22, 23]) which is especially fitted for languages with control operators where reduction rules that manipulate the current context (or, "the rest of the computation") can be conveniently specified. We treat evaluation contexts as an independent syntactic category (rather than informally as "terms with a hole") for which we define typing rules and reducibility predicates. The formalization of the problem and the proof of termination then relies on the fact that each program can be represented explicitly by a term in context. Moreover, a subsequent application of program extraction from the (constructive) proof shows how continuations arise as the computational content of the reducibility predicate defined on contexts and provides a logical confirmation of the connection between contexts and continuations [16, 17].

In case of delimited-control operators *shift* and *reset*, the standard reduction semantics uses two layers of contexts [10]. In this work, we show how the context-based method can be extended to account for such a language under both the call-by-value and the call-by-name reduction strategy. By introducing typed contexts we obtain two new type systems for *shift* and *reset*, one for each reduction strategy. They are derived from the respective CPS translations and are sound with respect to reduction (the subject reduction property holds in each case). Our call-by-value type system is a refinement of the original type system of Danvy and Filinski [18] in that it admits a form of implicit context answer type polymorphism. For each of the termination proofs we present its computational content, i.e., an evaluator in continuation-passing style with two layers of continuations that is an instance of normalization by evaluation [6, 8, 11]. The programs compute weak head normal forms of well-typed lambda terms with *shift* and *reset* according to the given evaluation strategy and are provably correct with respect to the corresponding reduction semantics.

We also discuss another useful typing discipline for *shift* and *reset*—a system with a fixed answer type, for which termination holds as well and can be proved along the same lines provided the answer type is a base type [1].

PPDP'09, September 7–9, 2009, Coimbra, Portugal.

2. Call-by-Value Delimited Continuations

2.1 Reduction Semantics

In this section we present the call-by-value reduction semantics for the lambda calculus extended with delimited-control operators *shift* and *reset* [10].

We introduce three syntactic categories of terms, evaluation (reduction) contexts and metacontexts. The syntax of contexts encodes the chosen reduction strategy—here, left-to-right call by value:

$$\begin{array}{lrcl} (terms) & t & ::= & x \mid \lambda x.t \mid t\ t \mid \mathcal{S}k.t \mid \\ & & & \langle t \rangle \mid k \hookleftarrow t \mid E \hookleftarrow t \\ (CBV\ contexts) & E & ::= & \bullet \mid v\ E \mid E\ t \mid E' \hookleftarrow E \\ (metacontexts) & F & ::= & \Box \mid E \cdot F \\ (values) & v & ::= & \lambda x.t \end{array}$$

The grammar of terms extends lambda terms with the *shift* construct $\mathcal{S}k.t$ (where the operator $\mathcal{S}$ is a binder and the continuation variable k is bound in t), the delimited term $\langle t \rangle$ (where the delimiter $\langle \cdot \rangle$ is called *reset*), the application of a continuation variable to a term $k \hookleftarrow t$ (akin to the *throw* operator known from SML of New Jersey [26]) and the application of a captured context E to a term denoted $E \hookleftarrow t$. This last construct is not present in source terms but it can appear in the course of evaluation when the *shift* operator captures a context. Therefore, we distinguish terms with no subterm of the form $E \hookleftarrow t$ and we call them *plain terms*. Continuation variables k are separate from object variables x and they can only appear in the *shift* construct or the *throw* construct.

We define the sets of free and bound variables (of both kinds) in a term in the usual way, and we distinguish *closed* terms, i.e., terms with no free variables (of any kind). As is also standard, we identify terms that differ only in the names of their bound variables.

Both contexts and metacontexts can be regarded as terms with a hole. Contexts are standard and in our approach they are represented inside-out, i.e.: $\bullet$ represents the empty context, $v\ E$ represents the "term with a hole" $E[v\ [\,]]$, $E\ t$ represents $E[[\,]\ t]$ and $E' \hookleftarrow E$ represents $E[E' \hookleftarrow [\,]]$. A metacontext is a stack of contexts: the empty metacontext is denoted $\Box$ and the metacontext obtained by pushing a context E on top of a metacontext F is denoted $E \cdot F$. Each context on the stack is separated from the rest by a delimiter, and thus $\Box$ represents the term with a hole $\langle [\,] \rangle$, and $E \cdot F$ represents the term with a hole $F[\langle E[\,] \rangle]$. Formally, we can define the meaning of contexts (metacontexts) by the function *plug* ($plug_m$) mapping a term and a context (metacontext) to the term such a pair represents:

$$\begin{array}{rcl} plug\ (t, \bullet) & = & t \\ plug\ (t, v\ E) & = & plug\ (v\ t, E) \\ plug\ (t_0, E\ t_1) & = & plug\ (t_0\ t_1, E) \\ plug\ (t, E' \hookleftarrow E) & = & plug\ (E' \hookleftarrow t, E) \\ \\ plug_m\ (t, \Box) & = & t \\ plug_m\ (t, E \cdot F) & = & plug_m\ (\langle plug\ (t, E) \rangle, F) \end{array}$$

We write the result of plugging the term t in the context E in the usual way: $E[t]$. Similarly, t plugged in the metacontext F is written $F[t]$. We say a context is closed, if all terms occurring in it are closed. A metacontext is closed if all contexts occurring in it are closed.

Given the grammar of terms, contexts and metacontexts, we now define a *program* in the call-by-value language as a triple consisting of a term, a call-by-value context and a metacontext:

$$(programs) \quad p ::= \langle t, E, F \rangle$$

The program $\langle t, E, F \rangle$ represents the term obtained by plugging the term t into the context E and metacontext F, i.e., the term $plug_m\ (\langle plug\ (t, E) \rangle, F)$ or again $F[\langle E[t] \rangle]$. With such triples we can represent all terms in such a way that we explicitly show boundaries in a program: we distinguish the current term, its surrounding context up to the nearest enclosing *reset*, and the rest of the program beyond the *reset*. This definition allows various triples to represent the same program, i.e., the function $plug_m\ (plug\ (\cdot, \cdot), \cdot)$ applied to different triples may give the same term as a result. In other words, all such triples represent different *decompositions* of the same program. Computationally, we will identify them by considering programs as abstraction classes of the equivalence relation between triples defined as follows:

$$\langle t_0, E_0, F_0 \rangle \sim \langle t_1, E_1, F_1 \rangle := F_0[\langle E_0[t_0] \rangle] = F_1[\langle E_1[t_1] \rangle]$$

where the equality on the right-hand side denotes syntactic equality modulo alpha renaming. For example, the program $\langle (\lambda x.r)\, s, \bullet, E \cdot F \rangle$ can be otherwise represented by a program $\langle \lambda x.r, (\bullet\ s), E \cdot F \rangle$, or by $\langle s, ((\lambda x.r)\ \bullet), E \cdot F \rangle$, or by $\langle \langle (\lambda x.r)\ s \rangle, E, F \rangle$. It should be noted that programs are in fact terms, only represented in a way that allows one to easily see their decomposition into the three components. Such a representation is useful when performing reductions and well suited for defining reducibility predicates in Sections 2.3 and 3.3.

The call-by-value notion of reduction for this language is given by the following set of rules:

$$\begin{array}{lrcl} (\beta_v) & \langle (\lambda x.r)\, v, E, F \rangle & \to_{\mathrm{v}} & \langle r\{v/x\}, E, F \rangle \\ (shift) & \langle \mathcal{S}k.t, E, F \rangle & \to_{\mathrm{v}} & \langle t\{E/k\}, \bullet, F \rangle \\ (reset) & \langle \langle v \rangle, E, F \rangle & \to_{\mathrm{v}} & \langle v, E, F \rangle \\ (throw) & \langle E' \hookleftarrow v, E, F \rangle & \to_{\mathrm{v}} & \langle v, E', E \cdot F \rangle \end{array}$$

where v is a value and the notation $r\{v/x\}$ stands for the usual metaoperation of capture-avoiding substitution of v for variable x in r. Similarly, $t\{E/k\}$ denotes the metaoperation of capture-avoiding substitution of context E for continuation variable k in r. Terms of the form $(\lambda x.r)\, v$ are the familiar call-by-value *β-redexes*. Reduction of the operator *shift* (rule $(shift)$) $\mathcal{S}k.t$ takes place in any context E (i.e., in the surrounding program fragment up to the nearest enclosing reset) and it consists in capturing that context and substituting it for the continuation variable k in t. The metacontext remains unchanged during this operation. In turn, whenever a captured context is applied to a value in the rule $(throw)$, it becomes the current context for that value and the previous current context is pushed on the metacontext. Finally, the reduction for *reset* (rule $(reset)$) takes place whenever the term under the delimiter is evaluated to a value—in that case, the *reset* is no longer needed to delimit the context and is therefore dropped. We say a *redex* is the term component of the program occurring on the left-hand side of each of the contraction rules above.

Thanks to the unique-decomposition property of the calculus, the relation $\to_{\mathrm{v}}$ is deterministic and it is a function on programs as abstraction classes of the relation $\sim$.

PROPERTY 1 (Unique decomposition (CBV)). *For all terms t, t either is a value, or it decomposes* uniquely *into a CBV context E, a metacontext F and a potential redex*[1] *r, i.e., $t = F[\langle E[r] \rangle]$.*

Finally, we define the evaluation relation as the reflexive-transitive closure of one-step reduction ($\to_{\mathrm{v}}^*$). The result of the evaluation is a (program) value of the form $p_v := \langle v, \bullet, \Box \rangle$. A program value consists simply of a term value in the empty context and the empty metacontext. Note that—according to the interpretation of program triples—a program value is a term consisting of a value (i.e., a lambda abstraction) delimited by a reset.

[1] A potential redex is either a proper redex that can be contracted, or a "stuck" term (i.e., a term that neither is a value nor can be further reduced). The type systems considered in this article ensure that there are no stuck terms in the course of reduction of (closed) programs.

Terms:

$$S \quad ::= \quad b \mid S \,{}_U\!\to_V T$$

$$\frac{}{\Gamma,\, x : S; \Delta \mid T \vdash x : S \mid T} \qquad \frac{\Gamma,\, x : S; \Delta \mid U \vdash t : T \mid V}{\Gamma; \Delta \mid W \vdash \lambda x.t : S \,{}_U\!\to_V T \mid W}$$

$$\frac{\Gamma; \Delta \mid X \vdash t_0 : S \,{}_U\!\to_W T \mid V \qquad \Gamma; \Delta \mid W \vdash t_1 : S \mid X}{\Gamma; \Delta \mid U \vdash t_0\, t_1 : T \mid V}$$

$$\frac{\Gamma; \Delta \mid U \vdash t : U \mid S}{\Gamma; \Delta \mid T \vdash \langle t \rangle : S \mid T} \qquad \frac{\Gamma; \Delta,\, k : S \rhd T \mid V \vdash t : V \mid U}{\Gamma; \Delta \mid T \vdash \mathcal{S}k.t : S \mid U}$$

$$\frac{\Gamma; \Delta,\, k : S \rhd T \mid U \vdash t : S \mid V}{\Gamma; \Delta,\, k : S \rhd T \mid U \vdash k \hookleftarrow t : T \mid V} \qquad \frac{\Gamma; \Delta \vdash E : S \rhd T \qquad \Gamma; \Delta \mid U \vdash t : S \mid V}{\Gamma; \Delta \mid U \vdash E \hookleftarrow t : T \mid V}$$

Contexts:

$$C \quad ::= \quad S \rhd T$$

$$\frac{}{\Gamma; \Delta \vdash \bullet : S \rhd S} \qquad \frac{\Gamma; \Delta \vdash E : T \rhd U \qquad \Gamma; \Delta \mid V \vdash t : S \mid W}{\Gamma; \Delta \vdash E\, t : (S \,{}_U\!\to_V T) \rhd W}$$

$$\frac{\Gamma; \Delta \mid W \vdash v : S \,{}_U\!\to_V T \mid W \qquad \Gamma; \Delta \vdash E : T \rhd U}{\Gamma; \Delta \vdash v\, E : S \rhd V} \qquad \frac{\Gamma; \Delta \vdash E' : S \rhd T \qquad \Gamma; \Delta \vdash E : T \rhd U}{\Gamma; \Delta \vdash E' \hookleftarrow E : S \rhd U}$$

Metacontexts:

$$D \quad ::= \quad \neg S$$

$$\frac{}{\Gamma; \Delta \vdash \Box : \neg S} \qquad \frac{\Gamma; \Delta \vdash E : S \rhd T \qquad \Gamma; \Delta \vdash F : \neg T}{\Gamma; \Delta \vdash E \cdot F : \neg S}$$

Programs:

$$\frac{\Gamma; \Delta \mid T \vdash F[\langle E[t] \rangle] : S \mid T}{\Gamma; \Delta \vdash \langle t, E, F \rangle : S}$$

Figure 1. Type system à la Danvy and Filinski for call by value

In the remainder of this section, whenever we consider terms, contexts, metacontexts, or programs, we refer to their well-typed counterparts.

2.2 Type System à la Danvy and Filinski

The type system of Danvy and Filinski is the most liberal of monomorphic type systems proposed for the language with *shift* and *reset* and it has been derived from the call-by-value CPS definitional interpreter for this language [18]. The slightly modified system we propose is shown in Figure 1.

In our modification of Danvy and Filinski's type system, we not only assign types to terms, but also to contexts and metacontexts. In typing judgments, Γ is the typing context of object variables (i.e., a list of pairs $x : S$) and Δ is the typing context of continuation variables (i.e., a list of pairs $k : S \rhd T$). Contexts are assigned types of the form $S \rhd T$, where S is the type of the hole of the context and T is its answer type. Metacontexts are assigned types of the form $\neg S$, where S is the type of the hole of the metacontext. A typing judgment for terms uses effect annotations and is of the form $\Gamma; \Delta | T \vdash t : S | U$ and can be interpreted in the following way: under the typing assumptions Γ and Δ, term t can be put in a context of type $S \rhd T$ and a metacontext of type $\neg U$ (in general, evaluating a well-typed term may use its surrounding context of type $S \rhd T$ to produce a value of type U, where T and U can be different). Operationally, the effect annotations can be understood as the type of the surrounding context before (T) and after (U) evaluating t. Because lambda abstractions encode "frozen" computations that can be activated by application to an argument, the type of a lambda abstraction is annotated with additional two types: the function type $S \,{}_U\!\to_V T$ denotes the type of a function that can be applied to an argument of type S in a context of type $T \rhd U$ and a metacontext of type $\neg V$ (in general, applying a well-typed function to an argument of type S may use the surrounding context of type $T \rhd U$ to produce a value of type V, where U and V can be different). We notice that in the conclusion of the typing rules for the necessary pure (i.e., control-effect free expressions) x, $\lambda x.t$ and $\langle t \rangle$ the two effect annotations are equal, but otherwise arbitrary. Such terms can be put in any context and always return a value to the surrounding context. If for such a term t a judgment $\Gamma; \Delta \mid T \vdash t : S \mid T$ is derivable for some type T, then $\Gamma; \Delta \mid T' \vdash t : S \mid T'$ is derivable for any other type T'. [2] Consequently, since a program represents

[2] In general, t is a pure expression, as defined by Asai and Kameyama [5], if $\Gamma; \Delta \mid T \vdash t : S \mid T$ is derivable for any type T.

a delimited term, its type does not depend on the effect annotations in the premise of the typing rule for programs.

What is characteristic of Danvy and Filinski's original type system is that, unlike in our type system, continuation variables are assigned functional types (where the two annotations on the arrow must be the same type—continuations captured by *shift* are static [13] and hence contain a control delimiter) and do not need an explicit *throw* construct. The original typing rule for *shift* reads as follows:

$$\frac{\Gamma; \Delta,\ k\ :\ S\ _W \rightarrow _W\ T \mid V\ \vdash\ t\ :\ V \mid U}{\Gamma; \Delta \mid T\ \vdash\ \mathcal{S}k.t\ :\ S \mid U}$$

A consequence of this design choice is that a captured continuation can be used only in contexts of the specified and chosen upfront answer type W, resulting in the lack of context answer type polymorphism that causes rejection of some interesting programs by this type system [5]. Let us discuss this issue using a classical example—the following program listing list prefixes, traditionally presented to illustrate the power of Danvy and Filinski's type system [10]:[3]

$$\begin{array}{l} \mathit{letrec\ prefix}\ xs = \\ \quad \mathit{match}\ xs\ \mathit{with} \\ \quad\quad nil \rightarrow \mathcal{S}k.nil \\ \quad\quad \mid\ \ y :: ys \rightarrow y :: (\mathcal{S}k.(k\ nil) :: \langle k\ (\mathit{prefix}\ ys)\rangle) \\ \mathit{in}\ \lambda xs.\langle \mathit{prefix}\ xs\rangle \end{array}$$

In order to analyze this program, we introduce typing rules for recursion and lists (again, derived from the CBV evaluator in CPS):

$$\frac{\begin{array}{c}\Gamma,\ f\ :\ U\ _W \rightarrow _X\ V,\ x\ :\ U; \Delta \mid W\ \vdash\ t_1\ :\ V \mid X \\ \Gamma,\ f\ :\ U\ _W \rightarrow _X\ V; \Delta \mid T\ \vdash\ t_2\ :\ S \mid Y\end{array}}{\Gamma; \Delta \mid T\ \vdash\ \mathit{letrec}\ f\ x\ =\ t_1\ \mathit{in}\ t_2\ :\ S \mid Y}$$

$$\frac{}{\Gamma; \Delta \mid T\ \vdash\ nil\ :\ S\ \mathit{list} \mid T}$$

$$\frac{\Gamma; \Delta \mid U\ \vdash\ t_1\ :\ S \mid V \qquad \Gamma; \Delta \mid T\ \vdash\ t_2\ :\ S\ \mathit{list} \mid U}{\Gamma; \Delta \mid T\ \vdash\ t_1 :: t_2\ :\ S\ \mathit{list} \mid V}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \mid W\ \vdash\ t\ :\ V\ \mathit{list} \mid U \\ \Gamma; \Delta \mid T\ \vdash\ t_1\ :\ S \mid W \\ \Gamma,\ x\ :\ V,\ xs\ :\ V\ \mathit{list}; \Delta \mid T\ \vdash\ t_2\ :\ S \mid W\end{array}}{\Gamma; \Delta \mid T\ \vdash\ \mathit{match}\ t\ \mathit{with}\ nil \rightarrow t_1 \mid x :: xs \rightarrow t_2\ :\ S \mid U}$$

We can observe that the captured continuation k in the induction case of the function *prefix* is used in two different contexts. In Danvy and Filinski's original type system, the first application requires k to be of type $S\ \mathit{list}\ _{S\ \mathit{list}\ \mathit{list}} \rightarrow _{S\ \mathit{list}\ \mathit{list}}\ S\ \mathit{list}$, whereas the second requires k to be of type $S\ \mathit{list}\ _{S\ \mathit{list}} \rightarrow _{S\ \mathit{list}}\ S\ \mathit{list}$. The program, therefore, does not type-check in Danvy and Filinski's original type system. Recently, Asai and Kameyama have proposed a generalization of Danvy and Filinski's type system introducing let-polymorphism for *shift* and *reset* [5] that addresses the above issue, but it turns out that we obtain a sufficient form of polymorphism (context answer type polymorphism), if captured continuations are not represented as functions, but as contexts (after all, continuations are not functions) accompanied by the explicit *throw* construct. Indeed, a context (and hence, a continuation variable) does not need to be assigned a full function type; the types annotating the arrow are redundant (witness the typing rules for *shift* and *throw* in Figure 1). Therefore, we obtain a type system with a form of polymorphism—a captured continuation can be applied in contexts of any answer type.

Adjusting the program listing list prefixes accordingly (continuation applications are performed by *throw*):

$$\begin{array}{l} \mathit{letrec\ prefix}\ xs = \\ \quad \mathit{match}\ xs\ \mathit{with} \\ \quad\quad nil \rightarrow \mathcal{S}k.nil \\ \quad\quad \mid\ \ y :: ys \rightarrow y :: (\mathcal{S}k.(k \hookleftarrow nil) :: \langle k \hookleftarrow (\mathit{prefix}\ ys)\rangle) \\ \mathit{in}\ \lambda xs.\langle \mathit{prefix}\ xs\rangle \end{array}$$

we see that k captured in *prefix* is simply given the type $S\ \mathit{list} \rhd S\ \mathit{list}$ and it can freely be used in different contexts. Thus, the function *prefix* is well typed and the whole expression listing list prefixes is of type $S\ \mathit{list}\ _T \rightarrow _T\ S\ \mathit{list}\ \mathit{list}$, for any types S and T.

Our type system is thus a minimal refinement of Danvy and Filinski's original type system that accepts the prefixes example as well as the other examples discussed by Asai and Kameyama [5]. The refinement we obtain is a natural consequence of representing captured continuations as typed contexts.

We now turn to proving some essential properties of the system. The relations between typing terms, contexts, metacontexts and programs is established by the following three lemmas:

LEMMA 1 (Decomposition of well-typed terms). *The following hold:*

1. *If* $\Gamma; \Delta \mid S\ \vdash\ E[t]\ :\ S \mid T$, *then* $\Gamma; \Delta \mid V\ \vdash\ t\ :\ U \mid T$ *and* $\Gamma; \Delta\ \vdash\ E\ :\ U \rhd V$ *for some types* U *and* V.
2. *If* $\Gamma; \Delta \mid T\ \vdash\ F[\langle t\rangle]\ :\ S \mid T$, *then* $\Gamma; \Delta \mid V\ \vdash\ t\ :\ V \mid U$ *and* $\Gamma; \Delta\ \vdash\ F\ :\ \neg U$ *for some types* U *and* V.
3. *If* $\Gamma; \Delta\ \vdash\ \langle t, E, F\rangle\ :\ S$, *then* $\Gamma; \Delta \mid U\ \vdash\ t\ :\ T \mid V$, $\Gamma; \Delta\ \vdash\ E\ :\ T \rhd U$ *and* $\Gamma; \Delta\ \vdash\ F\ :\ \neg V$ *for some types* T, U *and* V.

LEMMA 2 (Recomposition of well-typed terms). *The following hold:*

1. *If* $\Gamma; \Delta \mid T\ \vdash\ t\ :\ S \mid U$ *and* $\Gamma; \Delta\ \vdash\ E\ :\ S \rhd T$, *then* $\Gamma; \Delta \mid T\ \vdash\ E[t]\ :\ T \mid U$.
2. *If* $\Gamma; \Delta \mid S\ \vdash\ t\ :\ S \mid T$ *and* $\Gamma; \Delta\ \vdash\ F\ :\ \neg T$, *then* $\Gamma; \Delta \mid V\ \vdash\ F[\langle t\rangle]\ :\ U \mid V$ *for some type* U *and any type* V.
3. *If* $\Gamma; \Delta \mid U\ \vdash\ t\ :\ T \mid V$, $\Gamma; \Delta\ \vdash\ E\ :\ T \rhd U$ *and* $\Gamma; \Delta\ \vdash\ F\ :\ \neg V$, *then* $\Gamma; \Delta\ \vdash\ \langle t, E, F\rangle\ :\ S$ *for some type* S.

LEMMA 3 (Type preservation through plugging). *Let* $\Gamma; \Delta \mid S\ \vdash\ E[t]\ :\ S \mid T$, $\Gamma; \Delta \mid S\ \vdash\ E'[t']\ :\ S \mid T$ *and* $\Gamma; \Delta\ \vdash\ F\ :\ \neg T$. *If* $\Gamma; \Delta | V \vdash F[\langle E[t]\rangle]\ :\ U | V$, *then* $\Gamma; \Delta | V \vdash F[\langle E'[t']\rangle]\ :\ U | V$.

Substituting a well-typed value (context) for a free variable (continuation variable) of the right type preserves types:

LEMMA 4 (Substitution lemma). *The following hold:*

1. *If* $\Gamma,\ x\ :\ V; \Delta \mid T\ \vdash\ t\ :\ S \mid U$ *and* $\Gamma; \Delta \mid W\ \vdash\ v\ :\ V \mid W$, *then* $\Gamma; \Delta \mid T\ \vdash\ t\{v/x\}\ :\ S \mid U$.
2. *If* $\Gamma; \Delta,\ k\ :\ C \mid T\ \vdash\ t\ :\ S \mid U$ *and* $\Gamma; \Delta\ \vdash\ E\ :\ C$, *then* $\Gamma; \Delta \mid T\ \vdash\ t\{E/k\}\ :\ S \mid U$.

Finally, using the above lemmas we prove the subject reduction property of our type system:

THEOREM 1 (Subject Reduction). *If* $\Gamma; \Delta\ \vdash\ p\ :\ S$ *and* $p \rightarrow_{\mathrm{v}} p'$, *then* $\Gamma; \Delta\ \vdash\ p'\ :\ S$.

As a corollary, we obtain strong type soundness of evaluation of closed well-typed programs:

[3] Biernacka et al. [10] presented a version of this program with the expression $k\ nil$ replaced by $\langle k\ nil\rangle$. Such a modification solves the typing problem discussed in this section, but the added delimiter is operationally redundant—k is a static delimited continuation and as such it already contains a control delimiter.

COROLLARY 1. *If* $\cdot;\cdot \vdash p : S$ *and,* $p \to_{\mathrm{v}}^* p_v$*, then* $\cdot;\cdot \vdash p_v : S$.

Let us end this section by presenting a CPS translation for the language under consideration and a theorem that justifies the proposed typing rules with respect to the image of the translation—the simply typed lambda calculus. We show a CPS translation with one layer of continuations that extends Plotkin's call-by-value CPS translation [34], leading to terms in continuation-composing style [18] (the evaluator of Section 2.3.2, on the other hand, embodies the two-layer CPS that, thanks to the presence of the meta-continuation, eliminates non-tail calls):

$$\begin{array}{rcl}
\overline{x} & = & \lambda k.k\ x \\
\overline{\lambda x.t} & = & \lambda k.k\ (\lambda x.\overline{t}) \\
\overline{t_0\ t_1} & = & \lambda k.\overline{t_0}\ (\lambda v_0.\overline{t_1}\ (\lambda v_1.v_0\ v_1\ k)) \\
\overline{\langle t \rangle} & = & \lambda k.k\ (\overline{t}\ (\lambda v.v)) \\
\overline{\mathcal{S}k'.t} & = & \lambda k.\overline{t}\{k/k'\}\ (\lambda v.v) \\
\overline{k' \hookleftarrow t} & = & \lambda k.\overline{t}\ (\lambda v.k\ (k'\ v)) \\
\overline{E \hookleftarrow t} & = & \lambda k.\overline{t}\ (\lambda v.k\ ([\![E]\!]\ v))
\end{array}$$

where $[\![\cdot]\!]$ refunctionalizes contexts into continuations they represent:

$$\begin{array}{rcl}
[\![\bullet]\!] & = & \lambda v.v \\
[\![E\ t_1]\!] & = & \lambda v_0.\overline{t_1}\ (\lambda v_1.v_0\ v_1\ [\![E]\!]) \\
[\![v_0\ E]\!] & = & \lambda v_1.v_0^*\ v_1\ [\![E]\!] \\
[\![E' \hookleftarrow E]\!] & = & \lambda v.[\![E]\!]\ ([\![E']\!]\ v)
\end{array}$$

and $(\lambda x.t)^* = \lambda x.\overline{t}$.

The translation on types is defined as follows:

$$\begin{array}{rcl}
\overline{b} & = & b \\
\overline{S\ _U \to_V T} & = & \overline{S} \to (\overline{T} \to \overline{U}) \to \overline{V}
\end{array}$$

and is extended to typing environments pointwise, using the following translation rule for Δ:

$$\overline{S \rhd T} \quad = \quad \overline{S} \to \overline{T}$$

PROPOSITION 1. *The CBV type system of Figure 1 is correct with respect to the call-by-value CPS translation:*

1. *If* $\Gamma;\Delta \mid T \vdash t : S \mid U$*, then* $\overline{\Gamma},\overline{\Delta} \vdash \overline{t} : (\overline{S} \to \overline{T}) \to \overline{U}$.
2. *If* $\Gamma;\Delta \vdash E : S \rhd T$*, then* $\overline{\Gamma},\overline{\Delta} \vdash [\![E]\!] : \overline{S} \to \overline{T}$.

2.3 Termination of Evaluation

We are now in a position to state the main result—a direct proof of termination for call-by-value evaluation using a context-based variant of Tait's reducibility predicates. [4] For simplicity, we will only consider closed terms but the theorem and the proof extend to open terms. In the absence of values of base type, the only values obtained by evaluation of closed terms are lambda abstractions.

2.3.1 Logical Predicates and the Proof of Termination

We define three families of mutually inductive predicates. Each of these families is defined by induction on types: $\mathcal{R}_S$ is defined on term values and is indexed by term types, $\mathcal{C}_C$ is defined on contexts and is indexed by context types, and finally $\mathcal{M}_D$ is defined on metacontexts and is indexed by metacontext types.

[4] One of the reviewers pointed out that an extended version of Asai's article on call-by-value logical relations for *shift* and *reset* [4] contains a definition of reducibility predicates that are intended for proving normalization of *shift* and *reset*. Whereas Asai's predicates are defined on (arbitrary) terms and lambda abstractions representing continuations, ours reflect the structure of the definitional (derived from the definitional interpreter [10]) call-by-value reduction semantics with two layers of contexts and are defined on values, contexts and metacontexts.

$$\begin{array}{rcl}
\mathcal{R}_b(v) & := & \mathit{True} \\
\mathcal{R}_{S\ U \to_V T}(v_0) & := & \forall v_1.\mathcal{R}_S(v_1) \to \forall E.\mathcal{C}_{T \rhd U}(E) \to \\
 & & \quad \forall F.\mathcal{M}_{\neg V}(F) \to \mathcal{N}(\langle v_0\ v_1, E, F\rangle) \\
\mathcal{C}_{S \rhd T}(E) & := & \forall v.\mathcal{R}_S(v) \to \\
 & & \quad \forall F.\mathcal{M}_{\neg T}(F) \to \mathcal{N}(\langle v, E, F\rangle) \\
\mathcal{M}_{\neg S}(F) & := & \forall v.\mathcal{R}_S(v) \to \mathcal{N}(\langle v, \bullet, F\rangle) \\
\textit{where}\quad \mathcal{N}(p) & := & \exists v.\, p \to_{\mathrm{v}}^* \langle v, \bullet, \Box\rangle
\end{array}$$

These predicates can be seen as a contextualized (or, double-CPS-translated) version of standard, direct-style Tait reducibility predicates.[5] A reducible value of function type is such that whenever applied to another reducible value, it normalizes if plugged in any reducible context and metacontext. A reducible context in turn is such that plugged with a reducible value and put in a reducible metacontext, it normalizes. Finally, a reducible metacontext is such that plugged with a reducible value in the empty context, it normalizes. We do not need to define reducibility predicates on all terms because under call by value both functions and continuations accept only values as arguments.

Our goal is then to prove $\mathcal{N}(p)$ for each closed, well-typed program p, i.e., that each such program evaluates to a value program using the call-by-value evaluation strategy.

The predicates are only defined for well-typed terms, contexts and metacontexts of suitable types, and—by construction—all programs constructed in these definitions are well typed as well. Note that the normalization predicate $\mathcal{N}$ does not have a type annotation—we do not need to know the type of the entire program in order to prove the normalization theorem; we only need to know that it is well typed.

In the following, whenever we say that a value v is of type S, we mean that v is well typed, i.e., the judgment $\cdot;\cdot \mid T \vdash v : S \mid T$ is derivable for a type T. We do not care for type T, because if such a judgment is derivable for one type T, it is derivable for any type (cf. the discussion in Section 2.2).[6]

In the proof of the main result, we will need the following property:

LEMMA 5. *For all types* S*,* $\mathcal{C}_{S \rhd S}(\bullet)$ *holds.*

PROOF. Assume v is a value of type S and such that $\mathcal{R}_S(v)$ holds. We need to show that for all metacontexts F satisfying $\mathcal{M}_{\neg S}$, $\mathcal{N}(\langle v, \bullet, F\rangle)$ holds. But this fact holds by the assumption $\mathcal{M}_{\neg S}(F)$ applied to v. □

Now we are ready to state the main lemma.

LEMMA 6. *Let* $\Gamma;\Delta \mid T \vdash t : S \mid U$*, where* $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ *and* $\Delta = k_1 : C_1, \ldots, k_m : C_m$*, and let* t *be a plain term. Next, let* $\vec{v}$ *be a sequence of closed well-typed value terms such that* $\cdot;\cdot \mid V \vdash v_i : T_i \mid V$ *and* $\mathcal{R}_{T_i}(v_i)$ *for* $1 \le i \le n$*, and let* $\vec{E}$ *be a sequence of closed well-typed contexts such that* $\cdot;\cdot \vdash E_i : C_i$ *and* $\mathcal{C}_{C_i}(E_i)$ *for* $1 \le i \le m$*. Then for all closed well-typed contexts* E *such that* $\cdot;\cdot \vdash E : S \rhd T$ *and* $\mathcal{C}_{S \rhd T}(E)$ *and for all closed well-typed metacontexts* F *such that* $\cdot;\cdot \vdash F : \neg U$ *and* $\mathcal{M}_{\neg U}(F)$*, the program* $\langle t\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\}, E, F\rangle$ *normalizes, i.e.,* $\mathcal{N}(\langle t\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\}, E, F\rangle)$ *holds.*

[5] Tait's reducibility predicates define a reducible term of an arrow type as such that, when applied to another reducible term, produces an application that again is reducible.

[6] In a similar vein, if we add a constant value c of base type to the language, its typing rule would be $\Gamma;\Delta \mid T \vdash c : b \mid T$.

PROOF. The proof is done by induction on the structure of t.

Case x**.** By assumption x is one of the variables x_i and we have $t\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\} = v_i$. Hence, by assumption $\mathcal{R}_S(v_i)$ and for any E such that $\mathcal{C}_{S\triangleright T}(E)$ and for any F such that $\mathcal{M}_{\neg U}(F)$ holds, unfolding the definition of $\mathcal{C}_{S\triangleright T}$ entails that $\mathcal{N}(\langle v_i, E, F\rangle)$ holds.

Case $\lambda x.r$**.** Because $\lambda x.r$ is well typed, its type S must be an arrow type; let $S = S'\ {}_V \to_W\ S''$. Moreover, $T = U$. Taking $r' = r\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\}$, we have $(\lambda x.r)\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\} = \lambda x.r'$. We will show that $\mathcal{R}_S(\lambda x.r')$ holds, and from this fact it follows that the required $\mathcal{N}(\langle(\lambda x.r)\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\}, E, F\rangle)$ holds as in the previous case. In order to prove $\mathcal{R}_S(\lambda x.r')$, let us assume that v is a value of type S' and such that $\mathcal{R}_{S'}(v)$ holds. Next, let E be a well-typed context of type $S'' \triangleright V$ and such that $\mathcal{C}_{S''\triangleright V}(E)$ holds. Next let F be a well-typed metacontext of type $\neg W$ such that $\mathcal{M}_{\neg W}(F)$ holds. We have to prove that $\mathcal{N}(\langle(\ \lambda x.r')v, E, F\rangle)$ holds. By the reduction rule $(beta_v)$, $\langle(\lambda x.r')\ v, E, F\rangle$ reduces in one step to the program $\langle r\{\vec{v}/\vec{x}, v/x\}\{\vec{E}/\vec{k}\}, E, F\rangle$. By induction hypothesis, $\mathcal{N}(\langle r\{\vec{v}/\vec{x}, v/x\}\{\vec{E}/\vec{k}\}, E, F\rangle)$ holds and therefore also $\mathcal{N}(\langle(\lambda x.r')\ v, E, F\rangle)$ also holds.

Case $t_0\ t_1$**.** Since $t_0\ t_1$ is well typed, then $\Gamma; \Delta \mid V \vdash t_0 : W\ {}_T \to_Y S \mid U$ and $\Gamma; \Delta \mid Y \vdash t_1 : W \mid V$ for some types V, W, Y. Taking $t_0' = t_0\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\}$ and $t_1' = t_1\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\}$, we have $(t_0\,t_1)\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\} = t_0'\,t_1'$. By definition, the program $\langle t_0'\ t_1', E, F\rangle$ is the same as the program represented by $\langle t_0', E\ t_1', F\rangle$. Since t_0 is a subterm of $t_0\ t_1$, we can apply the induction hypothesis to deduce $\mathcal{N}(\langle t_0', E\ t_1', F\rangle)$ provided that $E\ t_1'$ is well typed and that $\mathcal{C}_{(W\ {}_T\to_Y S)\triangleright V}(E\ t_1')$ holds. The former is easy to see, and for the latter let us unfold the definition of $\mathcal{C}_{(W\ {}_T\to_Y S)\triangleright V}$. Let v be a value of type $W\ {}_T \to_Y S$ and such that $\mathcal{R}_{W\ {}_T\to_Y S}(v)$ holds. We need to show that $\mathcal{N}(\langle v, E\ t_1', F'\rangle)$ holds for any F' satisfying $\mathcal{M}_{\neg V}(F')$. Here again we can use another representative of the class of programs equal to $\langle v, E\ t_1', F'\rangle$, such as $\langle t_1', v\ E, F'\rangle$. Now we can apply the induction hypothesis again, this time for t_1, provided that $v\ \ E$ is well typed and $\mathcal{C}_{W\triangleright Y}(v\ \ E)$ holds. And again, the former property is easy to see, and for the latter we again unfold the definition of $\mathcal{C}_{W\triangleright Y}$: let v' be a value of type W and such that $\mathcal{R}_W(v')$ holds. We now need to show that $\mathcal{N}(\langle v', v\ E, F''\rangle)$ holds for any typable F'' such that $\mathcal{M}_{\neg Y}(F'')$ holds. But this is equivalent to showing that $\mathcal{N}(\langle v\ v', E, F''\rangle)$ holds, and this property follows from the fact that $\mathcal{R}_{W\ {}_T\to_Y S}(v)$ holds by an earlier assumption.

Case $\mathcal{S}k.r$**.** In this case, we have $\Gamma; \Delta, k : S \triangleright T \,|\, V \vdash r : V \,|\, U$ for some type V. Let $r' = r\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\}$. We need to show $\mathcal{N}(\langle \mathcal{S}k.r', E, F\rangle)$ for any E, F of suitable types and such that $\mathcal{C}_{S\triangleright T}(E), \mathcal{M}_{\neg U}(F)$ hold. According to the $(shift)$ reduction rule, $\langle \mathcal{S}k.r', E, F\rangle$ reduces in one step to $\langle r'\{E/k\}, \bullet, F\rangle$. It is thus enough to prove that $\mathcal{N}(\langle r'\{E/k\}, \bullet, F\rangle)$ holds. This fact we can prove by applying the induction hypothesis, because E satisfies the required conditions, and $\mathcal{C}_{V\triangleright V}(\bullet)$ holds by Lemma 5.

Case $\langle r\rangle$. Here, $\Gamma; \Delta \mid V \vdash r : V \mid S$ holds for some type V and $T = U$. We have to prove that $\mathcal{N}(\langle\langle r'\rangle, E, F\rangle)$ holds for any E and F of suitable types such that $\mathcal{C}_{S\triangleright T}(E), \mathcal{M}_{\neg U}(F)$ hold (and where $r' = r\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\}$). We can decompose the program $\langle\langle r'\rangle, E, F\rangle$ alternatively as $\langle r', \bullet, E \cdot F\rangle$. But we can prove $\mathcal{N}(\langle r', \bullet, E \cdot F\rangle)$ by induction hypothesis, using two facts: $\mathcal{C}_{V\triangleright V}(\bullet)$ holds by Lemma 5 and $\mathcal{M}_{\neg S}(E \cdot F)$ holds. To see the latter fact, let us take a value v such that $\mathcal{R}_S(v)$ holds. We need to show $\mathcal{N}(\langle v, \bullet, E \cdot F\rangle)$. The program $\langle v, \bullet, E \cdot F\rangle$ can be alternatively represented by program $\langle\langle v\rangle, E, F\rangle$ and by the $(reset)$ reduction rule, $\langle\langle v\rangle, E, F\rangle$ reduces in one step to program $\langle v, E, F\rangle$ and $\mathcal{N}(\langle v, E, F\rangle)$ holds by the assumption $\mathcal{C}_{S\triangleright T}(E)$ applied to v and F.

Case $\kappa_i \hookleftarrow r$. Here, $\Gamma; \Delta \mid T \vdash r : V \mid U$ holds for some type V such that $C_i = V \triangleright S$. We have to prove that $\mathcal{N}(\langle E_i \hookleftarrow r', E, F\rangle)$ with r', E, F given as in the previous cases. Choosing another representative of the same program, we can prove $\mathcal{N}(\langle r', E_i \hookleftarrow E, F\rangle)$ instead. This last fact can be obtained by applying the induction hypothesis to r if only we can prove that $E_i \hookleftarrow E$ is well typed and $\mathcal{C}_{V\triangleright T}(E_i \hookleftarrow E)$. To see the latter, let v be a value of type V satisfying $\mathcal{R}_V$ and let F' be a metacontext of type $\neg T$ satisfying $\mathcal{M}_{\neg T}$. We now need to show that $\mathcal{N}(\langle v, E_i \hookleftarrow E, F'\rangle)$ holds. Again, we can use an alternative representation of this program and show that $\mathcal{N}(\langle E_i \hookleftarrow v, E, F'\rangle)$. We can now see that this program reduces in one step to $\langle v, E_i, E \cdot F'\rangle$. Next, we observe that $\mathcal{N}(\langle v, E_i, E \cdot F'\rangle)$ holds by applying $\mathcal{C}_{V\triangleright S}(E_i)$ to v and $E \cdot F'$. The final fact to prove is $\mathcal{M}_{\neg S}(E \cdot F')$. To this end, let v' be value of type S such that $\mathcal{R}_S(v')$ holds. To prove $\mathcal{N}(\langle v', \bullet, E \cdot F'\rangle)$ we can use another representative, $\langle v', E, F'\rangle$, and we obtain $\mathcal{N}(\langle v', E, F'\rangle)$ by applying the assumption $\mathcal{C}_{S\triangleright T}(E)$.

□

The main result of this section is the following theorem.

THEOREM 2 (Termination of CBV evaluation). *If t is a closed well-typed plain term such that $\cdot; \cdot \mid S \vdash t : S \mid S$ is derivable, then it evaluates to a value, i.e., $\mathcal{N}(\langle t, \bullet, \Box\rangle)$ holds.*

PROOF. By Lemma 5, $\mathcal{C}_{S\triangleright S}(\bullet)$ holds. Furthermore, $\mathcal{M}_{\neg S}(\Box)$ holds because $\langle v, \bullet, \Box\rangle$ is already a value program for any value v. Therefore, we can apply Lemma 6 with the empty sequence of values and contexts and with the empty context and metacontext to obtain $\mathcal{N}(\langle t, \bullet, \Box\rangle)$. □

2.3.2 Extracted Evaluator

The normalization problem and Theorem 2 can be formalized in type theory in various ways and the computational content can be extracted from the proof. In order to keep things simple, we present here an informal description of the extraction method and the resulting program—an evaluator for the object language, obtained by hand based on the modified realizability interpretation. (In contrast, a fully formalized development in a proof assistant usually contains many distracting details; furthermore, the program obtained by mechanical extraction may be unreadable and require further simplifications done by hand.)

The idea of program extraction relies on the Curry-Howard correspondence between proofs and terms (programs). A proof of Theorem 2 treated as a lambda term contains both logical and computational information. The computation in the proof is used to construct terms—witnesses for the existential quantifier.[7] Program extraction then consists in erasing the logical parts from the proof term. The extraction method ensures that the resulting program is provably correct with respect to the specification stated in the theorem. In our case, the extracted program is a call-by-value evaluator for plain terms in the lambda calculus with *shift* and *reset* that uses two layers of continuations (more precisely, it is a normalizer, i.e., a function that returns object-level weak head normal forms). Leaving the technicalities of such formalization and extraction aside (there exist various formalizations of related

[7] Theorem 2 has the form $\forall t P(t) \to \exists v Q(t, v)$, where P, Q are logical formulas. Hence the computational content of its proof is a function from terms to values.

$$
\begin{array}{lcl}
\mathbf{eval}^{\vec{x},\vec{k}}\ x_i & = & \lambda\vec{v}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.\kappa\ v_i\ u_i\ \gamma \\
\mathbf{eval}^{\vec{x},\vec{k}}\ \lambda x.t & = & \lambda\vec{v}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.\kappa\ (\lambda x.t')\ (\lambda vuE\kappa\gamma.\mathbf{eval}^{\vec{x}x,\vec{k}}\ t\ (\vec{v}v)(\vec{u}u)\vec{E}\vec{\kappa}\ E\kappa\ \gamma)\ \gamma \\
 & & \quad \textit{where } \lambda x.t' = (\lambda x.t)\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\} \\
\mathbf{eval}^{\vec{x},\vec{k}}\ t_0\ t_1 & = & \lambda\vec{v}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.\mathbf{eval}^{\vec{x},\vec{k}}\ t_0\ \vec{v}\vec{u}\vec{E}\vec{\kappa}(E\ t_1') \\
 & & \qquad (\lambda v_0u_0\gamma.\mathbf{eval}^{\vec{x},\vec{k}}\ t_1\ \vec{v}\vec{u}\vec{E}\vec{\kappa}(v_0\ E) \\
 & & \qquad\quad (\lambda v_1u_1\gamma.u_0v_1u_1\ E\kappa\gamma)\ \gamma)\gamma \\
 & & \quad \textit{where } t_1' = t_1\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\} \\
\mathbf{eval}^{\vec{x},\vec{k}}\ \mathcal{S}k.t & = & \lambda\vec{v}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.\mathbf{eval}^{\vec{x},(\vec{k}k)}\ t\ \vec{v}\vec{u}(\vec{E}E)(\vec{\kappa}\kappa)\ \bullet\ \kappa_0\ \gamma \\
\mathbf{eval}^{\vec{x},\vec{k}}\ \langle t\rangle & = & \lambda\vec{v}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.\mathbf{eval}^{\vec{x},\vec{k}}\ t\ \vec{v}\vec{u}\vec{E}\vec{\kappa}\ \bullet\ \kappa_0\ (\lambda vu.\kappa\ v\ u\ \gamma) \\
\mathbf{eval}^{\vec{x},\vec{k}}\ k_i \hookleftarrow t & = & \lambda\vec{v}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.\mathbf{eval}^{\vec{x},\vec{k}}\ t\ \vec{v}\vec{u}\vec{E}\vec{\kappa}\ (E_i \hookleftarrow E)(\lambda vu\gamma.\kappa_i\ v\ u\ (\lambda vu.k\ v\ u\ \gamma))\ \gamma \\
\mathbf{norm}\ t & = & \mathbf{eval}^{\epsilon,\epsilon}\ t\epsilon\epsilon\epsilon\epsilon\ \bullet\ \kappa_0\ \gamma_0 \\
 & & \quad \textit{where} \quad \kappa_0 = \lambda vu\gamma.\gamma\ v\ u \\
 & & \qquad\qquad \gamma_0 = \lambda vu.v
\end{array}
$$

Figure 2. The call-by-value evaluator extracted from proof of Theorem 2

problems the reader may consult for illustration [6, 7, 11]), we only present the resulting evaluator. We also ignore the question of typability of the extracted code.

The evaluator is presented in Figure 2. The function $\mathbf{eval}$ is the computational content of Lemma 6 and it reflects object-level terms at the metalevel. The function $\mathbf{norm}$ reifies these metafunctions back to the syntax by applying $\mathbf{eval}$ to the empty sequence of values and contexts (the empty sequence is denoted ϵ), and to the initial continuation and metacontinuation.[8] The initial continuation κ_0 is the program extracted from the proof of Lemma 5 and the initial metacontinuation γ_0 is the program extracted from the proof of $\mathcal{M}_{\neg D}(\Box)$ for any type D.

The function $\mathbf{eval}$ is parameterized by the vectors of free object variables ($\vec{x}$) and free continuation variables ($\vec{k}$). For each kind of variables, the evaluator threads two environments: one with the syntactic objects to be substituted in the final value, and one with their functional representations. Specifically, $\vec{v}$ contains values, $\vec{u}$ contains functions representing values, $\vec{E}$ contains contexts, and $\vec{\kappa}$ contains continuations (i.e., functional representations of contexts).

The current continuation is denoted κ and it is obtained by extracting the computational content of the predicate $\mathcal{C}_C$. The current metacontinuation is denoted γ and it is obtained by extraction from the predicate $\mathcal{M}_D$. Apart from these functional representations of the current context and the current metacontext, from the proof of Lemma 6 we also extract the syntactic representations of the current context and metacontext. The former is threaded so that it may be captured by a *shift* operator; the latter is unused and therefore it has been omitted in Figure 2. This optimization in extraction corresponds to dead-code elimination and it has been proven correct in the setting of modified realizability for first-order logic by Berger [6].

[8] The evaluator is thus an instance of normalization by evaluation, i.e., a technique where normalization of object-level terms is obtained by evaluation at metalevel [8, 14]. In particular, β-reduction is performed by an application of a metalevel function to a value, and throwing a value to a captured context is performed by an application of a metalevel continuation to this value.

The evaluator of Figure 2 differs from the definitional interpreter for *shift* and *reset* with two layers of continuations because of the use of normalization by evaluation. In particular, here we compute syntactic normal forms and therefore two environments must be maintained for both object and continuation variables.

A careful reader may notice that the evaluator in Figure 2 works for any term, typable or not. In particular, it may diverge for some input terms. This fact is due to the formalization of the termination theorem, where the typability information for a term is considered computationally irrelevant and is dropped in extraction. The formalization can be adjusted to ensure that only well-typed terms are evaluated by making typing information computationally relevant. One of possible ways in this case is to prove decidability of type-checking in a constructive logic. The computational content of such a proof is a type-checker that can be called by the evaluator before trying to evaluate a given term (the evaluator will be given a term and its type as input).

3. Call-by-Name Delimited Continuations

In this section, we consider a call-by-name version of typed lambda calculus with *shift* and *reset*. The reduction semantics we present coincides with that attributed to Danvy in a recent work of Herbelin and Ghilezan who proposed a different calculus for call-by-name delimited continuations [27].

Since Danvy and Filinski's type system has been derived from the call-by-value definitional evaluator in CPS, it is sensitive to reduction strategy and does not account for call by name. Therefore, we give a novel type system for call-by-name delimited continuations, derived from call-by-name CPS. Next, we define reducibility predicates and prove termination of call-by-name evaluation. We only show the highlights of the development which is otherwise carried out along the same lines as for call by value in Section 2.

3.1 Reduction Semantics

The grammar of terms and metacontexts is the same for call by name and call by value; only the syntax of contexts differs:

$$(CBN\ contexts) \quad E ::= \bullet \mid E\ t$$

Terms:

$$S \quad ::= \quad b \mid {S^{T,U}}_W \to_X V$$

$$\frac{}{\Gamma,\, x : S^{T,U}; \Delta \mid T \vdash x : S \mid U} \qquad \frac{\Gamma,\, x : S^{T,U}; \Delta \mid W \vdash t : V \mid X}{\Gamma; \Delta \mid Y \vdash \lambda x.t : {S^{T,U}}_W \to_X V \mid Y}$$

$$\frac{\Gamma; \Delta \mid X \vdash t_0 : {S^{T,U}}_W \to_X V \mid Y \qquad \Gamma; \Delta \mid T \vdash t_1 : S \mid U}{\Gamma; \Delta \mid W \vdash t_0\, t_1 : V \mid Y}$$

$$\frac{\Gamma; \Delta \mid S \vdash t : S \mid T}{\Gamma; \Delta \mid U \vdash \langle t \rangle : T \mid U} \qquad \frac{\Gamma; \Delta,\, k : S \rhd T \mid W \vdash t : W \mid U}{\Gamma; \Delta \mid T \vdash \mathcal{S}k.t : S \mid U}$$

$$\frac{\Gamma; \Delta,\, k : S \rhd T \mid T \vdash t : S \mid U}{\Gamma; \Delta,\, k : S \rhd T \mid V \vdash k \hookleftarrow t : U \mid V} \qquad \frac{\Gamma; \Delta \vdash E : S \rhd T \qquad \Gamma; \Delta \mid T \vdash t : S \mid U}{\Gamma; \Delta \mid V \vdash E \hookleftarrow t : U \mid V}$$

Contexts:

$$C \quad ::= \quad S \rhd T$$

$$\frac{}{\Gamma; \Delta \vdash \bullet : S \rhd S} \qquad \frac{\Gamma; \Delta \vdash E : S \rhd T \qquad \Gamma; \Delta \mid W \vdash t : V \mid X}{\Gamma; \Delta \vdash E\, t : (V^{W,X}{}_T \to_Y S) \rhd Y}$$

Metacontexts:

$$D \quad ::= \quad \neg S$$

$$\frac{}{\Gamma; \Delta \vdash \Box : \neg S} \qquad \frac{\Gamma; \Delta \vdash E : S \rhd T \qquad \Gamma; \Delta \vdash F : \neg T}{\Gamma; \Delta \vdash E \cdot F : \neg S}$$

Programs:

$$\frac{\Gamma; \Delta \mid T \vdash F[\langle E[t] \rangle] : S \mid T}{\Gamma; \Delta \vdash \langle t, E, F \rangle : S}$$

Figure 3. Type system à la Danvy and Filinski for call by name

Programs are defined as triples as in call by value.

The one-step reduction relation for call by name contains the same reduction rules as the call-by-value case except for β-reduction and the rule for applying a captured context:

$$\begin{array}{lrcl} (\beta) & \langle (\lambda x.r)\, s, E, F \rangle & \to_{\mathrm{n}} & \langle r\{s/x\}, E, F \rangle \\ (\mathit{throw}) & \langle E' \hookleftarrow t, E, F \rangle & \to_{\mathrm{n}} & \langle t, E', E \cdot F \rangle \end{array}$$

where a function or a captured context is applied to an arbitrary term rather than to a value.

3.2 Type System à la Danvy and Filinski

We present a call-by-name typing system à la Danvy and Filinski (Figure 3), derived from the call-by-name CPS for *shift* and *reset*, be it in the form of an evaluator or a translation to the simply typed lambda calculus. The judgments of the type system are of the form $\Gamma; \Delta \mid T \vdash t : S \mid U$ and they are interpreted exactly as Danvy and Filinski's judgments described in Section 2.2. In the call-by-name CPS, however, functions do not accept values but thunks, i.e., delayed computations expecting a continuation. This fact is manifested in the form of the arrow type ${S^{T,U}}_W \to_X V$ and in the type of variables $S^{T,U}$ occurring in typing environments Γ. The intuition behind this notation is as follows: a function argument of type $S^{T,U}$ can be evaluated in a context of type $S \rhd T$ and metacontext of type $\neg U$. Programs are typed as in call by value. For example, the type of the program listing list prefixes from Section 2.2 is ${S\ \mathit{list}}^{S\ \mathit{list}\ \mathit{list}, S\ \mathit{list}\ \mathit{list}}{}_T \to_T S\ \mathit{list}\ \mathit{list}$ (for any types S and T) under call by name.

Following the same steps as in Section 2.2, we can show that the CBN reduction semantics preserves types assigned by the CBN type system.

THEOREM 3 (Subject Reduction). *If* $\Gamma; \Delta \vdash p : S$ *and* $p \to_{\mathrm{n}} p'$, *then* $\Gamma; \Delta \vdash p' : S$.

COROLLARY 2. *If* $\cdot; \cdot \vdash p : S$ *and* $p \to_{\mathrm{n}}^* p_v$, *then* $\cdot; \cdot \vdash p_v : S$.

We also define the call-by-name CPS translation on terms and types in order to show correctness of the type system with respect to CPS. The call-by-name CPS translation to the simply typed lambda calculus is defined as follows:

$$\begin{array}{rcl} \overline{x} & = & \lambda k.x\, k \\ \overline{\lambda x.t} & = & \lambda k.k\, (\lambda x.\overline{t}) \\ \overline{t_0\, t_1} & = & \lambda k.\overline{t_0}\, (\lambda v.v\, \overline{t_1}\, k) \\ \overline{\langle t \rangle} & = & \lambda k.k\, (\overline{t}\, (\lambda v.v)) \\ \overline{\mathcal{S}k'.t} & = & \lambda k.\overline{t}\{k/k'\}\, (\lambda v.v) \\ \overline{k' \hookleftarrow t} & = & \lambda k.k\, (\overline{t}\, k') \\ \overline{E \hookleftarrow t} & = & \lambda k.k\, (\overline{t}\, [\![E]\!]) \end{array}$$

where $[\![\cdot]\!]$ refunctionalizes contexts into continuations they represent:

$$\begin{array}{rcl} [\![\bullet]\!] & = & \lambda v.v \\ [\![E\ t_1]\!] & = & \lambda v_0.v_0\, \overline{t_1}\, [\![E]\!] \end{array}$$

$$\begin{array}{rcl} \overline{b} & = & b \\ \overline{{S^{T,U}}_W \to_X V} & = & ((\overline{S} \to \overline{T}) \to \overline{U}) \to (\overline{V} \to \overline{W}) \to \overline{X} \end{array}$$

$$
\begin{array}{lcl}
\mathbf{eval}^{\vec{x},\vec{k}}\ x_i & = & \lambda\vec{r}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.u_i\ E\ \kappa\ \gamma \\
\mathbf{eval}^{\vec{x},\vec{k}}\ \lambda x.t & = & \lambda\vec{r}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.\kappa\ (\lambda x.t')\ (\lambda ruE\kappa\gamma.\mathbf{eval}^{\vec{x}x,\vec{k}}\ t\ (\vec{r}r)(\vec{u}u)\vec{E}\vec{\kappa}\ E\kappa\ \gamma)\ \gamma \\
 & & \quad \textit{where } \lambda x.t' = (\lambda x.t)\{\vec{r}/\vec{x}\}\{\vec{E}/\vec{k}\} \\
\mathbf{eval}^{\vec{x},\vec{k}}\ t_0\ t_1 & = & \lambda\vec{r}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.\mathbf{eval}^{\vec{x},\vec{k}}\ t_0\ \vec{r}\vec{u}\vec{E}\vec{\kappa}(E\ t_1') \\
 & & \qquad (\lambda v_0u_0\gamma.u_0t_1'\ (\lambda E\kappa\gamma.\mathbf{eval}^{\vec{x},\vec{k}}\ t_1\ \vec{r}\vec{u}\vec{E}\vec{\kappa}\ E\kappa\ \gamma)\ E\kappa\ \gamma)\ \gamma \\
 & & \quad \textit{where } t_1' = t_1\{\vec{r}/\vec{x}\}\{\vec{E}/\vec{k}\} \\
\mathbf{eval}^{\vec{x},\vec{k}}\ \mathcal{S}k.t & = & \lambda\vec{r}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.\mathbf{eval}^{\vec{x},(\vec{k}k)}\ t\ \vec{r}\vec{u}(\vec{E}E)(\vec{\kappa}\kappa)\ \bullet\ \kappa_0\ \gamma \\
\mathbf{eval}^{\vec{x},\vec{k}}\ \langle t\rangle & = & \lambda\vec{r}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.\mathbf{eval}^{\vec{x},\vec{k}}\ t\ \vec{r}\vec{u}\vec{E}\vec{\kappa}\ \bullet\ \kappa_0\ (\lambda vu.\kappa\ v\ u\ \gamma) \\
\mathbf{eval}^{\vec{x},\vec{k}}\ k_i \hookleftarrow t & = & \lambda\vec{r}\vec{u}\vec{E}\vec{\kappa}E\kappa\gamma.\mathbf{eval}^{\vec{x},\vec{k}}\ t\ \vec{r}\vec{u}\vec{E}\vec{\kappa}\ E_i\kappa_i\ (\lambda vu.\kappa\ v\ u\ \gamma) \\
\mathbf{norm}\ t & = & \mathbf{eval}^{\epsilon,\epsilon}\ t\epsilon\epsilon\epsilon\epsilon\ \bullet\ \kappa_0\ \gamma_0 \\
 & & \quad \textit{where}\quad \kappa_0 = \lambda vu\gamma.\gamma\ v\ u \\
 & & \qquad\qquad \gamma_0 = \lambda vu.v
\end{array}
$$

Figure 4. The call-by-name evaluator extracted from proof of Theorem 4

Typing environments are translated pointwise using the following translation rules:

$$
\begin{array}{lcl}
\overline{S^{T,U}} & = & (\overline{S} \to \overline{T}) \to \overline{U} \\
\overline{S \rhd T} & = & \overline{S} \to \overline{T}
\end{array}
$$

PROPOSITION 2. *The CBN type system of Figure 3 is correct with respect to the call-by-name CPS translation:*

1. *If* $\Gamma;\Delta \mid T \vdash t : S \mid U$, *then* $\overline{\Gamma},\overline{\Delta} \vdash \overline{t} : (\overline{S} \to \overline{T}) \to \overline{U}$.
2. *If* $\Gamma;\Delta \vdash E : S \rhd T$, *then* $\overline{\Gamma},\overline{\Delta} \vdash [\![E]\!] : \overline{S} \to \overline{T}$.

3.3 Termination of Evaluation

We now give the definitions of reducibility predicates and the statement of the termination theorem for call by name.

3.3.1 Logical Predicates and the Proof of Termination

We introduce three families of logical predicates defined on well-typed values ($\mathcal{R}_S$), contexts ($\mathcal{C}_C$) and metacontexts ($\mathcal{M}_D$) in a similar way to the call-by-value case:

$$
\begin{array}{rcl}
\mathcal{R}_b(v) & := & \mathit{True} \\
\mathcal{R}_{S^{T,U}\ W \to X\ V}(v) & := & \forall t.\, \mathcal{Q}_S^{T,U}(t) \to \mathcal{Q}_V^{W,X}(v\ t) \\
\mathcal{Q}_S^{T,U}(t) & := & \forall E.\, \mathcal{C}_{S \rhd T}(E) \to \\
 & & \quad \forall F.\, \mathcal{M}_{\neg U}(F) \to \mathcal{N}(\langle t, E, F\rangle) \\
\mathcal{C}_{S \rhd T}(E) & := & \forall v.\, \mathcal{R}_S(v) \to \\
 & & \quad \forall F.\, \mathcal{M}_{\neg T}(F) \to \mathcal{N}(\langle v, E, F\rangle) \\
\mathcal{M}_{\neg S}(F) & := & \forall v.\, \mathcal{R}_S(v) \to \mathcal{N}(\langle v, \bullet, F\rangle) \\
\mathcal{N}(p) & := & \exists v.\, p \to_{\mathrm{n}}^* \langle v, \bullet, \Box\rangle
\end{array}
$$

Unlike in call by value, we need an auxiliary predicate $\mathcal{Q}_S^{T,U}$ defined on well-typed terms t such that $\cdot;\cdot \mid T \vdash t : S \mid U$. This predicate expresses the property that a term plugged in any reducible context and metacontext normalizes as a program.

LEMMA 7. *Let* $\Gamma;\Delta \mid T \vdash t : S \mid U$, *where* $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ *and* $\Delta = k_1 : C_1, \ldots, k_m : C_m$, *and let* t *be a plain term. Next, let* $\vec{r}$ *be a sequence of closed well-typed terms such that* $\cdot;\cdot \mid T \vdash r_i : T_i \mid U$ *and* $\mathcal{Q}_{T_i}^{T,U}(r_i)$ *for* $1 \le i \le n$, *and let* $\vec{E}$ *be a sequence of closed well-typed contexts such that* $\cdot;\cdot \vdash E_i : C_i$ *and* $\mathcal{C}_{C_i}(E_i)$ *for* $1 \le i \le m$. *Then for all closed well-typed contexts* E *such that* $\cdot;\cdot \vdash E : S \rhd T$ *and* $\mathcal{C}_{S \rhd T}(E)$ *and for all closed well-typed metacontexts* F *such that* $\cdot;\cdot \vdash F : \neg U$ *and* $\mathcal{M}_{\neg U}(F)$, *the program* $\langle t\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\}, E, F\rangle$ *normalizes, i.e.,* $\mathcal{N}(\langle t\{\vec{v}/\vec{x}\}\{\vec{E}/\vec{k}\}, E, F\rangle)$ *holds.*

THEOREM 4 (Termination of CBN evaluation). *If* t *is a closed well-typed plain term such that* $\cdot;\cdot \mid S \vdash t : S \mid S$ *is derivable, then it evaluates to a value, i.e.,* $\mathcal{N}(\langle t, \bullet, \Box\rangle)$ *holds.*

The proofs of Lemma 7 and Theorem 4 are carried out analogously to the call-by-value case.

3.3.2 Extracted Evaluator

The program extracted from Theorem 4 is presented in Figure 4. The evaluator for call by name differs from that in call by value in that the use of continuations imposes the call-by-name evaluation order. Moreover, the two environments $\vec{t}$ and $\vec{u}$ contain terms and *thunks*, respectively, to be substituted for free object variables (rather than values and their functional representations—functions, as in call by value). Thunks represent delayed computations that can be activated by an application—here, to a context, a continuation and a metacontext. They arise as the computational content of proofs of the predicate $\mathcal{Q}_S$ for any type S.

4. Type System with a Fixed Answer Type

Another type system for *shift* and *reset* considered in the literature is the type system with a fixed answer type, induced by Filinski's implementation of *shift* and *reset* in ML [24]. This system is considerably simpler but also more restrictive than that of Danvy and Filinski. The idea is that the answer type of a context and the type of the surrounding metacontext must agree and all contexts inside the program are required to have the same answer type—the (top-level) type of the program (programs always are closed by the top-level reset).[9] This type system is insensitive to reduction strategy. We

[9] Relaxing the latter restriction leads to Murthy's type system [33] that is a special case of Danvy and Filinski's type system.

Terms:

$$S \quad ::= \quad b \mid S \to T$$

$$\frac{}{\Gamma,\, x : S; \Delta \vdash_U x : S} \qquad \frac{\Gamma,\, x : S; \Delta \vdash_U t : T}{\Gamma; \Delta \vdash_U \lambda x.t : S \to T} \qquad \frac{\Gamma; \Delta \vdash_U t_0 : S \to T \qquad \Gamma; \Delta \vdash_U t_1 : S}{\Gamma; \Delta \vdash_U t_0\, t_1 : T}$$

$$\frac{\Gamma; \Delta \vdash_U t : U}{\Gamma; \Delta \vdash_U \langle t \rangle : U} \qquad \frac{\Gamma; \Delta, k : S \rhd U \vdash_U t : U}{\Gamma; \Delta \vdash_U \mathcal{S}k.t : S}$$

$$\frac{\Gamma; \Delta, k : S \rhd U \vdash_U t : S}{\Gamma; \Delta, k : S \rhd U \vdash_U k \hookleftarrow t : U} \qquad \frac{\Gamma; \Delta \vdash_U E : S \rhd U \qquad \Gamma; \Delta \vdash_U t : S}{\Gamma; \Delta \vdash_U E \hookleftarrow t : U}$$

Contexts:

$$C \quad ::= \quad S \rhd T$$

$$\frac{}{\Gamma; \Delta \vdash_U \bullet : U \rhd U} \qquad \frac{\Gamma; \Delta \vdash_U E : T \rhd U \qquad \Gamma; \Delta \vdash_U t : S}{\Gamma; \Delta \vdash_U E\, t : (S \to T) \rhd U}$$

$$\frac{\Gamma; \Delta \vdash_U v : S \to T \qquad \Gamma; \Delta \vdash_U E : T \rhd U}{\Gamma; \Delta \vdash_U v\, E : S \rhd U} \qquad \frac{\Gamma; \Delta \vdash_U E' : S \rhd U \qquad \Gamma; \Delta \vdash_U E : V \rhd U}{\Gamma; \Delta \vdash_U E' \hookleftarrow E : S \rhd U}$$

Metacontexts:

$$D \quad ::= \quad \neg S$$

$$\frac{}{\Gamma; \Delta \vdash_U \Box : \neg U} \qquad \frac{\Gamma; \Delta \vdash_U E : U \rhd U \qquad \Gamma; \Delta \vdash_U F : \neg U}{\Gamma; \Delta \vdash_U E \cdot F : \neg U}$$

Programs:

$$\frac{\Gamma; \Delta \vdash_S F[\langle E[t] \rangle] : S}{\Gamma; \Delta \vdash \langle t, E, F \rangle : S}$$

Figure 5. Type system with a fixed answer type

adapt it to the language with explicit contexts (where the chosen reduction strategy becomes essential) and show it for the call-by-value language in Figure 5 (for the call-by-name language, we only need to drop the lower two typing rules for contexts). Because of the type restriction, each typing judgment is annotated with the top-level type and all the continuation variables are required to have this type as answer type (cf. the last three rules). Furthermore, the type of a delimited term must be equal to the top-level type.

While the type system with a fixed answer type proves useful in many theoretical and practical applications, it is too restrictive to type-check, e.g., the function *prefix* of Section 2.2: in *prefix* contexts of type $S\ list \rhd S\ list$ are used to produce values of type $S\ list\ list$ (the answer type of the context does not agree with the type of the metacontext).

It has been shown by Ariola et al. that the type system with a fixed answer type has the normalization property provided the fixed type is a base type (it is a sufficient condition) and that in general the property may not hold [1]. Assuming that the fixed type is a base type, we can state the theorem and prove it along exactly the same lines as shown in Section 2.3 for call by value and in Section 3.3 for call by name. The definition of reducibility predicates in each case differs only in type annotations. In particular, the logical relations for call by value are defined as follows:

$$\begin{aligned}
\mathcal{R}^U_b(v) &:= \mathit{True} \\
\mathcal{R}^U_{S\to T}(v_0) &:= \forall v_1.\, \mathcal{R}^U_S(v_1) \to \forall E.\, \mathcal{C}_{T\rhd U}(E) \to \\
&\qquad \forall F.\, \mathcal{M}_{\neg U}(F) \to \mathcal{N}(\langle v_0\, v_1, E, F\rangle) \\
\mathcal{C}_{S\rhd U}(E) &:= \forall v.\, \mathcal{R}^U_S(v) \to \\
&\qquad \forall F.\, \mathcal{M}_{\neg U}(F) \to \mathcal{N}(\langle v, E, F\rangle) \\
\mathcal{M}_{\neg U}(F) &:= \forall v.\, \mathcal{R}^U_U(v) \to \mathcal{N}(\langle v, \bullet, F\rangle) \\
\mathcal{N}(p) &:= \exists v.\, p \to^*_{\mathrm{v}} \langle v, \bullet, \Box\rangle
\end{aligned}$$

Let us remark that when U is a base type the above definition is correct and otherwise it is not—the predicates are no longer defined inductively on the type structure. For example, if $U = S \to T$, then in order to see whether $\mathcal{R}^U_U(v)$ holds, one needs to know whether $\mathcal{R}^U_U(v)$ holds.

In the call-by-name case, reducibility predicates are defined as follows:

$$\begin{aligned}
\mathcal{R}^U_b(v) &:= \mathit{True} \\
\mathcal{R}^U_{S\to T}(v) &:= \forall t.\, \mathcal{Q}^U_S(t) \to \mathcal{Q}^U_T(v\, t) \\
\mathcal{Q}^U_S(t) &:= \forall E.\, \mathcal{C}_{S\rhd U}(E) \to \\
&\qquad \forall F.\, \mathcal{M}_{\neg U}(F) \to \mathcal{N}(\langle t, E, F\rangle) \\
\mathcal{C}_{S\rhd U}(E) &:= \forall v.\, \mathcal{R}^U_S(v) \to \\
&\qquad \forall F.\, \mathcal{M}_{\neg U}(F) \to \mathcal{N}(\langle v, E, F\rangle) \\
\mathcal{M}_{\neg U}(F) &:= \forall v.\, \mathcal{R}^U_U(v) \to \mathcal{N}(\langle v, \bullet, F\rangle) \\
\mathcal{N}(p) &:= \exists v.\, p \to^*_{\mathrm{n}} \langle v, \bullet, \Box\rangle
\end{aligned}$$

5. Conclusion and Future Work

Proofs of termination of evaluation or—more generally—of normalization for typed languages with control operators are usually done by a CPS translation to a strongly normalizing language, e.g., the simply typed lambda calculus. In this work we showed that context-based reduction semantics for static delimited-control operators with type structure put on terms, contexts and metacontexts lend themselves to direct proofs of termination of evaluation. Furthermore, the computational content of each of the proofs is materialized as a continuation-passing evaluator whose continuations and metacontinuations are extracted from the proofs of reducibility of contexts and metacontexts, respectively.

It is worth noting that the type-and-effect systems we present in this article arise naturally from continuation-passing semantics of the language and by treating captured continuations as separate entities that require an explicit *throw* construct. In particular, the latter choice led us to a refinement of Danvy and Filinski's original type system.

While the type systems we considered are monomorphic, the proof method we presented should be adaptable to Asai and Kameyama's polymorphic calculi for delimited continuations [5, 29]. It remains to investigate whether the context-based approach to proving weak head normalization of calculi with control operators could be adjusted to proofs of weak and strong normalization where reductions under binders can take place.

Acknowledgments

We would like to thank Olivier Danvy and the anonymous reviewers for their insightful and helpful comments on the present article. This work has been partially supported by the MNiSW grant number N N206 357436, 2009-2011.

References

[1] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A type-theoretic foundation of delimited continuations. *Higher-Order and Symbolic Computation*, 20(4):403–429, 2007.

[2] Kenichi Asai. Online partial evaluation for shift and reset. In Peter Thiemann, editor, *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2002)*, SIGPLAN Notices, Vol. 37, No 3, pages 19–30, Portland, Oregon, March 2002. ACM Press.

[3] Kenichi Asai. Offline partial evaluation for shift and reset. In Nevin Heintze and Peter Sestoft, editors, *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2004)*, pages 3–14, Verona, Italy, August 2003. ACM Press.

[4] Kenichi Asai. Logical relations for call-by-value delimited continuations. In Marko van Eekelen, editor, *Proceedings of the Sixth Symposium on Trends in Functional Programming (TFP 2005)*, pages 413–428, Tallinn, Estonia, September 2005. Institute of Cybernetics at Tallinn Technical University. Extended version available as Technical Report of Department of Information Science, Ochanomizu University, OCHA-IS 06-1.

[5] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *Proceedings of the Fifth Asian symposium on Programming Languages and Systems, APLAS'07*, number 4807 in Lecture Notes in Computer Science, pages 239–254, Singapore, December 2007.

[6] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[7] Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.

[8] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[9] Małgorzata Biernacka and Dariusz Biernacki. A context-based approach to proving termination of evaluation. In *Proceedings of the 25th Annual Conference on Mathematical Foundations of Programming Semantics(MFPS XXV)*, Oxford, UK, April 2009.

[10] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).

[11] Małgorzata Biernacka, Olivier Danvy, and Kristian Støvring. Program extraction from proofs of weak head normalization. In Martin Escardó, Achim Jung, and Michael Mislove, editors, *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics(MFPS XXI)*, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 169–189, Birmingham, UK, May 2005. Elsevier Science Publishers. Extended version available as the research report BRICS RS-05-12.

[12] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.

[13] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006.

[14] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[15] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[16] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Thielecke [38], pages 13–23. Invited talk.

[17] Olivier Danvy. Defunctionalized interpreters for programming languages. In Peter Thiemann, editor, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, SIGPLAN Notices, Vol. 43, No. 9, Victoria, British Columbia, September 2008. ACM Press. Invited talk.

[18] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[19] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[20] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[21] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.

[22] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Program-*

ming Languages. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987.

[23] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[24] Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994. ACM Press.

[25] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[26] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993. A preliminary version was presented at the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1991).

[27] Hugo Herbelin and Silvia Ghilezan. An approach to call-by-name delimited continuations. In Philip Wadler, editor, *Proceedings of the Thirty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 383–394. ACM Press, January 2008.

[28] Yukiyoshi Kameyama. Axioms for delimited continuations in the CPS hierarchy. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 442–457, Karpacz, Poland, September 2004. Springer.

[29] Yukiyoshi Kameyama and Kenichi Asai. Strong normalization of polymorphic calculus for delimited continuations. In *Proceedings of the Austrian-Japanese Workshop on Symbolic Computation in Software Science (SCSS 2008), RISC-Linz Report Series No. 08-08*, pages 96–108, Hagenberg, Austria, July 2008.

[30] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In Olin Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, SIGPLAN Notices, Vol. 38, No. 9, pages 177–188, Uppsala, Sweden, August 2003. ACM Press.

[31] Oleg Kiselyov. Call-by-name linguistic side effects. In *Proceedings of the 2008 Workshop on Symmetric calculi and Ludics for the semantic interpretation*, Hamburg, Germany, August 2008.

[32] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers. In Benjamin Pierce, editor, *Proceedings of the 2005 ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, SIGPLAN Notices, Vol. 40, No. 9, pages 192–203, Tallinn, Estonia, September 2005. ACM Press.

[33] Chethan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Olivier Danvy and Carolyn L. Talcott, editors, *Proceedings of the First ACM SIGPLAN Workshop on Continuations (CW'92)*, Technical report STAN-CS-92-1426, Stanford University, pages 49–72, San Francisco, California, June 1992.

[34] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[35] Chung-chieh Shan. Delimited continuations in natural language: quantification and polarity sensitivity. In Thielecke [38], pages 55–64.

[36] Eijiro Sumii. An implementation of transparent migration on standard Scheme. In Matthias Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, Technical Report 00-368, Rice University, pages 61–64, Montréal, Canada, September 2000.

[37] William W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32:198–212, 1967.

[38] Hayo Thielecke, editor. *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, Venice, Italy, January 2004.

[39] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[40] Anne S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, 1973.