



INSTYTUT INFORMATYKI
UNIwersYTETU WROCLAWSKIEGO

INSTITUTE OF COMPUTER SCIENCE
UNIVERSITY OF WROCLAW

ul. Joliot-Curie 15
50-383 Wrocław
Poland

Report 01/11

Małgorzata Biernacka, Dariusz Biernacki and Sergueï Lenglet

Simple proofs of termination of evaluation for System F with control operators

November 2011

Abstract

We present simple proofs of termination of evaluation in reduction semantics for System F with control operators. We use a modified version of Girard's proof method based on reducibility candidates, where the reducibility predicates are defined on values and on evaluation contexts. We address both abortive control operators (*callcc*) and delimited-control operators (*shift* and *reset*) and we consider both the call-by-value and call-by-name evaluation strategies. The computational content of the proofs we present are evaluators in continuation-passing style.

Simple proofs of termination of evaluation for System F with control operators

Małgorzata Biernacka, Dariusz Biernacki and Sergueï Lenglet

1 Introduction

In a previous work [4, 5], it has been shown that a context-based variant of Tait's proof method based on reducibility predicates [21, 22] allows for direct and concise proofs of termination of evaluation in reduction semantics for the simply-typed λ -calculus with control operators, be they abortive or delimited. A reduction semantics is a form of operational semantics with explicit representation of evaluation (reduction) contexts [13, 14], where the evaluation contexts represent continuations (more precisely, they are defunctionalized continuations [9, 10]). As such, reduction semantics is particularly convenient for expressing non-local control effects and has been most successfully used to express the semantics of control operators such as *callcc* [13, 14], or *shift* and *reset* [6] that otherwise requires a higher-order representation of continuations and is given either as a continuation-passing evaluation function, or through a translation to continuation-passing style [16].

One of the standard properties required of typed λ -calculi, crucial from both proof-theoretic and type-theoretic viewpoints, is termination of reductions. In the case of a λ -calculus considered as a deterministic programming language one is usually interested in termination of reductions according to an evaluation strategy that can be given by evaluation contexts, rather than in more general normalization properties. For typed languages with control operators, it is common to prove termination of evaluation (and of normalization in general) by translating, in a reduction-preserving way, terms in the source language to a target language for which the normalization property has been established before [16, 20]. The context-based proof method, on the other hand, directly takes advantage of the format of the reduction semantics, where the key role is played by evaluation contexts. So, for instance, in order to prove termination of evaluation for the simply-typed λ -calculus under call by value using the context-based method, one simply defines mutually inductively reducibility predicates on values (normal forms) as well as on evaluation contexts. The termination result then follows by induction on well-typed terms, where the reasoning is driven by the control flow of a typical evaluator in continuation-passing style [4, 5].

In this article, we show that the context-based method can be generalized from reducibility predicates to reducibility candidates, and therefore it provides simple proofs of termination of evaluation for System F with control operators. Just as for the simply-typed λ -calculi, normalization for polymorphic λ -calculi with control operators has been mainly established indirectly, via translations to strongly normalizing calculi: Harper and Lillibridge reduced termination of call-by-value evaluation for F_ω with *abort* and *callcc* to normalization in F_ω [17] by a CPS translation, Parigot reduced strong normalization of the second-order $\lambda\mu$ -calculus to strong normalization of the simply-typed λ -calculus [19], Danos et al. reduced strong normalization of the second-order classical logic to strong normalization of linear logic [8], and Asai and Kameyama reduced strong normalization of System F with *shift* and *reset* under the standard semantics to strong normalization of System F [2]. On the other hand, Parigot directly proved strong normalization of the second-order $\lambda\mu$ -calculus using another variant of Girard’s reducibility candidates [19]—a result that is our closest related work, discussed further in Section 2.3.

The calculi we consider are System F with *callcc* under call by value and call by name as well as System F with *shift* and *reset* under call by value (call by name is omitted for brevity), in each case with the standard semantics, where unlike in the ML-like semantics evaluation does not proceed under polymorphic abstraction [17]. The type system for *callcc* is inspired by that of Harper and Lillibridge [17], whereas the one for *shift* and *reset* is new and generalizes Asai and Kameyama’s type system [2] in that it allows for polymorphic abstractions over arbitrary expressions, not only pure ones. It is worth noting that, as in the simply-typed case [4, 5], the context-based proofs we present in this article have the structure of an evaluator in continuation-passing style that can be extracted from the proofs [3, 7].

The rest of this article is organized as follows. In Section 2, we present System F with abortive control operators and we prove termination of evaluation under the call-by-value and call-by-name evaluation strategies. We also relate this result to Parigot’s work [19]. In Section 3, we present System F with delimited-control operators and we prove termination of evaluation under call by value and we relate our type system to Asai and Kameyama’s [2]. In Section 4, we conclude.

2 System F with abortive control operators

In this section, we present a context-based proof of termination for call-by-value evaluation in System F extended with the control operator *callcc*. We use a variant of Girard’s method of reducibility candidates, where in particular we define reducibility predicates for reduction contexts.

2.1 Syntax and Semantics

We consider the explicitly typed System F under the call-by-value reduction strategy, which we extend with the binder version of the *callcc* operator (denoted \mathcal{K}) and with a construct to apply captured continuations \leftrightarrow , similar to the *throw* construct of SML/NJ [18]. We call this language λ_v^F . The syntax of terms, term types, and call-by-value contexts of λ_v^F is defined as follows:

$$\begin{aligned}
\text{Terms: } t & ::= x \mid \lambda x^S.t \mid tt \mid \Lambda X.t \mid t\{S\} \mid \\
& \quad \mathcal{K}k.t \mid k \leftrightarrow t \mid \ulcorner E^\urcorner \leftrightarrow t \\
\text{Term types: } S & ::= X \mid S \rightarrow S \mid \forall X.S \\
\text{CBV contexts: } E & ::= \bullet \mid (\lambda x^S.t) E \mid E t \mid E \{S\} \mid \ulcorner E^\urcorner \leftrightarrow E \\
\text{Values: } v & ::= \lambda x^S.t \mid \Lambda X.t
\end{aligned}$$

We let x range over term variables, k range over continuation variables, and X range over type variables, and we assume the three sets of variables are pairwise disjoint. We use capital letters starting from S to denote term types. The term $\Lambda X.t$ quantifies over the type variable X , while the term $t\{S\}$ instantiates such quantification with type S . The term $\mathcal{K}k.t$ denotes the *callcc* operator that binds a captured context (representing a continuation) to the variable k and makes it available in its body t . In turn, constructs of the form $k \leftrightarrow t$ and $\ulcorner E^\urcorner \leftrightarrow t$ denote the operation of *throwing* the term t to a continuation variable k and to the captured context $\ulcorner E^\urcorner$, respectively. The use of $\ulcorner \cdot \urcorner$ indicates that a context is reified as a term, as opposed to its role as the representation of the “rest of the program.”

Expressions of the form $\ulcorner E^\urcorner \leftrightarrow t$ are not allowed in source programs (because we do not let programmers handle contexts explicitly), but they may occur during evaluation. In the sequel, it will be useful to distinguish the subset of *plain terms*, i.e., terms without any subterm of the form $\ulcorner E^\urcorner \leftrightarrow t$.

An abstraction $\lambda x^S.t$ (resp., $\Lambda X.t$, $\mathcal{K}k.t$) binds x (resp., X , k) in t , and a type $\forall X.S$ binds X in S . We write $ftv(S)$ for the set of free type variables occurring in type S , defined in the usual way. The definitions of free term variables, free type variables, and free continuation variables of a term are also standard. A term is *closed* if it does not have any free variable of any kind. We identify terms and types up to α -conversion of their bound variables.

The syntax of reduction contexts encodes the reduction strategy, here—call by value. The contexts can be seen as “terms with a hole”, and are represented inside-out. Informally, \bullet denotes the empty context, $(\lambda x^S.t) E$ represents $E[(\lambda x^S.t) []]$ with the hole indicated by $[]$, $E t$ represents $E[[] t]$, $E \{S\}$ represents $E[[]\{S\}]$, and $\ulcorner E_0^\urcorner \leftrightarrow E$ represents $E[\ulcorner E_0^\urcorner \leftrightarrow []]$. A reduction context is closed iff all its components (terms, types, or contexts) are closed. We make the meaning of contexts precise by defining a function

$plug$ which maps a term and a context to the term which is obtained by putting the term in the hole of the context:

$$\begin{aligned}
plug(t, \bullet) &= t \\
plug(t_0, (\lambda x^S.t) E) &= plug((\lambda x^S.t) t_0, E) \\
plug(t_0, E t_1) &= plug(t_0 t_1, E) \\
plug(t, E \{S\}) &= plug(t\{S\}, E) \\
plug(t, \ulcorner E_0 \urcorner \leftarrow E) &= plug(\ulcorner E_0 \urcorner \leftarrow t, E)
\end{aligned}$$

We write $E[t]$ for the result of plugging t in the context E (i.e., the result of $plug(t, E)$).

A program p is a pair consisting of a term and a reduction context:

$$p ::= \langle t, E \rangle$$

The program $\langle t, E \rangle$ is an alternative representation of the term $E[t]$. We prefer to use the program notation to keep the term being decomposed in focus, separated from its surrounding context—such a representation is well suited to the context-based approach of proving termination that we take in Section 2.3. It is easy to see that different pairs of terms and contexts may represent the same plugged term; for example, the program $\langle (\lambda x^S.t_0) t_1, \bullet \rangle$ can be also represented by $\langle (\lambda x^S.t_0), \bullet t_1 \rangle$, or by $\langle t_1, (\lambda x^S.t_0) \bullet \rangle$. We equate all these representations by considering programs as equivalence classes of the following relation:

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

where $=$ denotes syntactic equality up to α -conversion.

The one-step reduction relation in the call-by-value strategy is defined on programs by the following rules:

$$\begin{aligned}
\langle (\lambda x^S.t) v, E \rangle &\rightarrow_v \langle t\{v/x\}, E \rangle & (\beta_v) \\
\langle (\Lambda X.t)\{S\}, E \rangle &\rightarrow_v \langle t\{S/X\}, E \rangle & (\beta_T) \\
\langle \mathcal{K}k.t, E \rangle &\rightarrow_v \langle t\{\ulcorner E \urcorner/k\}, E \rangle & (callcc) \\
\langle \ulcorner E_0 \urcorner \leftarrow v, E_1 \rangle &\rightarrow_v \langle v, E_0 \rangle & (throw_v)
\end{aligned}$$

where $t\{v/x\}$ (resp., $t\{S/X\}$, $t\{\ulcorner E \urcorner/k\}$) is the usual capture-avoiding substitution of value v (resp., of type S , of context $\ulcorner E \urcorner$) for variable x (resp., for X , for k) in t . The rules (β_v) and (β_T) are standard in System F; in addition, we introduce the rule $(callcc)$, where the current context E is captured by the $callcc$ operator and bound to the continuation variable k , and the rule $(throw_v)$, where a previously captured context E_0 is restored as the current context, and the context E_1 is discarded (the latter fact shows the abortive character of the $callcc$ operator). The term components of programs on the

left-hand side of the arrow in the above rules are called redexes, and are ranged over by r .

We define the call-by-value evaluation relation as the reflexive and transitive closure of the relation \rightarrow_v . The expected result of evaluation is a program value of the form $p_v := \langle v, \bullet \rangle$.

As can be expected, the reduction relation \rightarrow_v is deterministic; this property is ensured by the unique-decomposition lemma. We could state this lemma in a general version for all terms, but in order to consider only well-behaved programs and simplify the statement of the lemma, we choose to postpone it to the next section where we define well-typed programs.

2.2 Type System

We define a type system for λ_v^F that is an extension of the type system for the lambda calculus introduced by Biernacka and Biernacki [4], where types are assigned to terms as well as to contexts. The syntax of context types is defined as follows:

$$C ::= \neg S.$$

Roughly, the type $\neg S$ of a context E indicates that any well-typed term of type S can be plugged in E . The answer type of a context need not be specified, and it is often taken to be \perp to reflect the fact that continuations never return.¹ The answer type of a closed evaluation context E of type $\neg S$ can be determined by typing the expression $E[x]$ for a fresh variable x of type S .

We let Γ range over type environments for term variables (i.e., lists of pairs of the form $x : S$), and we let Δ range over type environments for continuation variables (i.e., lists of pairs of the form $k : C$). For a type environment $\Gamma = x_1 : S_1, \dots, x_n : S_n$, we define $ftv(\Gamma) := \cup_{i \in \{1, \dots, n\}} ftv(S_i)$. The typing rules for terms and contexts are shown in Figure 1.

Finally, we define the type of a program $\langle t, E \rangle$ to be the type of the plugged term $E[t]$ it represents:

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

It can be checked that the type of a program does not depend of the chosen representation.

We can now state the unique-decomposition lemma that ensures the determinism of the reduction relation \rightarrow_v , and progress of reduction:

¹This decision has more serious implications when a type system is studied from a logical perspective via the Curry-Howard isomorphism (see for example [1]). However, we do not take this viewpoint in this article.

Typing terms ($S ::= X \mid S \rightarrow T \mid \forall X.S$):

$$\begin{array}{c}
\frac{}{\Gamma, x : S; \Delta \vdash x : S} \qquad \frac{\Gamma, x : S; \Delta \vdash t : T}{\Gamma; \Delta \vdash \lambda x^S.t : S \rightarrow T} \\
\frac{\Gamma; \Delta \vdash t_0 : S \rightarrow T \quad \Gamma; \Delta \vdash t_1 : S}{\Gamma; \Delta \vdash t_0 t_1 : T} \qquad \frac{\Gamma; \Delta \vdash t : S \quad X \notin \text{ftv}(\Gamma)}{\Gamma; \Delta \vdash \Lambda X.t : \forall X.S} \\
\frac{\Gamma; \Delta \vdash t : \forall X.S}{\Gamma; \Delta \vdash t\{T\} : S\{T/X\}} \qquad \frac{\Gamma; \Delta, k : \neg S \vdash t : S}{\Gamma; \Delta \vdash \mathcal{K}k.t : S} \\
\frac{\Gamma; \Delta, k : \neg S \vdash t : S}{\Gamma; \Delta, k : \neg S \vdash k \leftrightarrow t : T} \qquad \frac{\Gamma; \Delta \vdash E : \neg S \quad \Gamma; \Delta \vdash t : S}{\Gamma; \Delta \vdash \lceil E \rceil \leftrightarrow t : T}
\end{array}$$

Typing contexts ($C ::= \neg S$):

$$\begin{array}{c}
\frac{}{\Gamma; \Delta \vdash \bullet : \neg S} \qquad \frac{\Gamma; \Delta \vdash \lambda x^S.t : S \rightarrow T \quad \Gamma; \Delta \vdash E : \neg T}{\Gamma; \Delta \vdash (\lambda x^S.t) E : \neg S} \\
\frac{\Gamma; \Delta \vdash t : S \quad \Gamma; \Delta \vdash E : \neg T}{\Gamma; \Delta \vdash E t : \neg(S \rightarrow T)} \qquad \frac{\Gamma; \Delta \vdash E : \neg(S\{T/X\})}{\Gamma; \Delta \vdash E \{T\} : \neg(\forall X.S)} \\
\frac{\Gamma; \Delta \vdash E_0 : \neg S \quad \Gamma; \Delta \vdash E_1 : \neg T}{\Gamma; \Delta \vdash \lceil E_0 \rceil \leftrightarrow E_1 : \neg S}
\end{array}$$

Figure 1: Type system for λ_v^F

Proposition 1 (Unique decomposition). *For all closed well-typed programs p , p either is a program value, or it decomposes uniquely into a context E and a redex r , i.e., $p = \langle r, E \rangle$.*

As for the subject reduction property, it is a more subtle issue. In the reduction rule ($throw_v$), the current evaluation context E_1 is replaced by another context E_0 , where the answer types of the two contexts do not have to be related in any way. Therefore, as observed before [4, 18, 23], in general subject reduction does not hold for languages with reduction and typing rules for continuation invocation similar to the ones presented in this

work. However, if we assume that all reified contexts in a given term have the same answer type as the type of the term itself—which is the case for all terms in the reduction sequence starting in a plain term—subject reduction will be recovered [4, 18, 23]. Such an assumption can be made implicit [18] or explicit in a refined type system that controls the answer types of evaluation contexts [4, 23]. It can be shown that from subject reduction of such a refined type system strong type soundness for plain terms in the original type system follows [4, 23].

2.3 Termination

We now prove termination of the call-by-value evaluation for λ_v^F , using a context variant of Girard’s method of reducibility candidates [15]. For simplicity, we only consider closed programs, but the proof can be extended to work for open terms as well.

In Girard’s proof of strong normalization for System F, a reducibility candidate is defined as a set of terms satisfying the following three properties:

- all terms belonging to the set are strongly normalizing;
- if a term belonging to the set reduces to a term t , then t is also in the set;
- if a non-value term t reduces only to terms belonging to the set, then t is also in the set.

Among all the candidates, the “true” reducibility predicate on terms is to be found; it is constructed by induction on types.

In our case, the definition of a reducibility candidate is simpler, because we are interested only in the termination of the call-by-value evaluation, not in strong normalization. First, we define the normalization predicate $\mathcal{N}(p)$ as follows:

$$\mathcal{N}(p) := \exists v. p \rightarrow_v^* \langle v, \bullet \rangle,$$

i.e., a program p normalizes ($\mathcal{N}(p)$ holds) if it reduces in several steps to a program value.

Definition 1 (Reducibility candidate). *A reducibility candidate \mathcal{R} of type S is any set of closed values of type S .*

We consider sets consisting of values only, and not of arbitrary terms; it is enough for the proof to go through. We write $\mathcal{RC}(S)$ for the set of reducibility candidates of type S . For each reducibility candidate \mathcal{R} of type S , we define the associated predicate $\mathcal{C}_{\mathcal{R}}$ on closed contexts of type $\neg S$ as follows:

$$\mathcal{C}_{\mathcal{R}}(E) := \forall v. v \in \mathcal{R} \rightarrow \mathcal{N}(\langle v, E \rangle)$$

As in the original proof, we introduce the notion of parametric reducibility candidates. Let S be a type, $\vec{X} = ftv(S)$, \vec{T} be a sequence of types of the same size as \vec{X} , and $\vec{\mathcal{R}}$ be such that $\mathcal{R}_i \in \mathcal{RC}(T_i)$. We define the parametric reducibility candidate $RED_S[\vec{\mathcal{R}}/\vec{X}]$ by induction on S :

$$\begin{aligned}
v \in RED_{X_i}[\vec{\mathcal{R}}/\vec{X}] & \text{ iff } v \in \mathcal{R}_i \\
v_0 \in RED_{S_1 \rightarrow S_2}[\vec{\mathcal{R}}/\vec{X}] & \text{ iff } \forall v_1. v_1 \in RED_{S_1}[\vec{\mathcal{R}}/\vec{X}] \rightarrow \\
& \quad \forall E. \mathcal{C}_{RED_{S_2}[\vec{\mathcal{R}}/\vec{X}]}(E) \rightarrow \mathcal{N}(\langle v_0 v_1, E \rangle) \\
v \in RED_{\forall c.S}[\vec{\mathcal{R}}/\vec{X}] & \text{ iff } \forall U. \forall \mathcal{S}. \mathcal{S} \in \mathcal{RC}(U) \rightarrow \\
& \quad \forall E. \mathcal{C}_{RED_S[\vec{\mathcal{R}}/\vec{X}, \mathcal{S}/c]}(E) \rightarrow \mathcal{N}(\langle v\{U\}, E \rangle)
\end{aligned}$$

It is easy to see that $RED_S[\vec{\mathcal{R}}/\vec{X}] \in \mathcal{RC}(S\{\vec{T}/\vec{X}\})$. To prove the main result, we need a substitution lemma.

Lemma 1. *We have $RED_{S\{T/c\}}[\vec{\mathcal{R}}/\vec{X}] = RED_S[\vec{\mathcal{R}}/\vec{X}, RED_T[\vec{\mathcal{R}}/\vec{X}]/c]$.*

Proof. By induction on S . □

We are now ready to state the main lemma:

Lemma 2. *Let t be a plain term such that $\Gamma; \Delta \vdash t : S$ with $\Gamma = x_1 : T_1, \dots, x_n : T_n$ and $\Delta = k_1 : \neg U_1, \dots, k_m : \neg U_m$. Let $\{X_1, \dots, X_p\}$ be the set of free type variables of S , T_1, \dots, T_n , and U_1, \dots, U_m . Let \vec{V} be a sequence of types of length p , and $\vec{\mathcal{R}}$ be reducibility candidates such that $\mathcal{R}_i \in \mathcal{RC}(V_i)$ for all $i = 1, \dots, p$. Let \vec{v} be closed values such that $\vdash v_i : T_i\{\vec{V}/\vec{X}\}$ and $v_i \in RED_{T_i}[\vec{\mathcal{R}}/\vec{X}]$ for all $i = 1, \dots, n$. Let \vec{E} be closed contexts such that $\mathcal{C}_{RED_{U_i}[\vec{\mathcal{R}}/\vec{X}]}(E_i)$ for all $i = 1, \dots, m$. Let E be such that $\mathcal{C}_{RED_S[\vec{\mathcal{R}}/\vec{X}]}(E)$. Then $\mathcal{N}(\langle t\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \vec{E}^\neg/\vec{k}\}, E \rangle)$ holds.*

Proof. By induction on t .

- In the case $t = x_i$, we have $t\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \vec{E}^\neg/\vec{k}\} = v_i$, as well as $T_i = S$. Because $v_i \in RED_{T_i}[\vec{\mathcal{R}}/\vec{X}]$, we have the required result by definition of $\mathcal{C}_{RED_{T_i}[\vec{\mathcal{R}}/\vec{X}]}(E)$.
- In the case $t = \lambda x^{S_1}.s$, we have $S = S_1 \rightarrow S_2$. We put $s' = t\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \vec{E}^\neg/\vec{k}\}$ and $S'_1 = S_1\{\vec{V}/\vec{X}\}$; then $t\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \vec{E}^\neg/\vec{k}\} = \lambda x^{S'_1}.s'$. We now prove that $\lambda x^{S'_1}.s' \in RED_S[\vec{\mathcal{R}}/\vec{X}]$; from that we can deduce the required result by the definition of $\mathcal{C}_{RED_S[\vec{\mathcal{R}}/\vec{X}]}(E)$. Let v be such that $v \in RED_{S_1}[\vec{\mathcal{R}}/\vec{X}]$, and let E' be such that $\mathcal{C}_{RED_{S_2}[\vec{\mathcal{R}}/\vec{X}]}(E')$. We have $\langle (\lambda x^{S'_1}.s') v, E' \rangle \rightarrow_v \langle s'\{v/x\}, E' \rangle$. By induction, we have $\mathcal{N}(\langle s'\{v/x\}, E' \rangle)$, therefore $\mathcal{N}(\langle (\lambda x^{S'_1}.s') v, E' \rangle)$ holds. Consequently, we have $\lambda x^{S'_1}.s' \in RED_S[\vec{\mathcal{R}}/\vec{X}]$ as required.

- In the case $t = t_0 t_1$, we have $\Gamma; \Delta \vdash t_0 : S' \rightarrow S$ and $\Gamma; \Delta \vdash t_1 : S'$ for some S' . Let $t'_0 = t_0\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \ulcorner \vec{E} \urcorner / \vec{k}\}$, $t'_1 = t_1\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \ulcorner \vec{E} \urcorner / \vec{k}\}$; then $t\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \ulcorner \vec{E} \urcorner / \vec{k}\} = t'_0 t'_1$. We have $\langle t'_0 t'_1, E \rangle = \langle t'_0, E t'_1 \rangle$, and to conclude, we would like to apply the induction hypothesis to t_0 . To this end, we have to prove that $\mathcal{C}_{RED_{S' \rightarrow S}[\vec{\mathcal{R}}/\vec{X}]}(E t'_1)$ holds. Let v_0 be such that $v_0 \in RED_{S' \rightarrow S}[\vec{\mathcal{R}}/\vec{X}]$. We want to prove that $\mathcal{N}(\langle v_0, E t'_1 \rangle)$ holds, which is equivalent to proving that $\mathcal{N}(\langle t'_1, v_0 E \rangle)$ holds. Again, we want to prove this fact by using the induction hypothesis on t_1 , but to do this, we first have to prove that $\mathcal{C}_{RED_{S'}[\vec{\mathcal{R}}/\vec{X}]}(v_0 E)$ holds. Let v_1 be such that $v_1 \in RED_{S'}[\vec{\mathcal{R}}/\vec{X}]$. Since we have $v_0 \in RED_{S' \rightarrow S}[\vec{\mathcal{R}}/\vec{X}]$ and $\mathcal{C}_{RED_S[\vec{\mathcal{R}}/\vec{X}]}(E)$, therefore $\mathcal{N}(\langle v_0 v_1, E \rangle)$ holds, i.e., we have $\mathcal{N}(\langle v_1, v_0 E \rangle)$. Consequently, $\mathcal{C}_{RED_{S'}[\vec{\mathcal{R}}/\vec{X}]}(v_0 E)$ holds. Therefore, we can use the induction hypothesis on t_1 to deduce that $\mathcal{N}(\langle t'_1, v_0 E \rangle)$ holds. As a result, we have $\mathcal{C}_{RED_{S' \rightarrow S}[\vec{\mathcal{R}}/\vec{X}]}(E t'_1)$; therefore we can prove the required fact by using the induction hypothesis on t_0 .
- In the case $t = \Lambda c.s$, we have $S = \forall c.S'$ for some S' . Let $s' = s\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \ulcorner \vec{E} \urcorner / \vec{k}\}$; then $t\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \ulcorner \vec{E} \urcorner / \vec{k}\} = \Lambda c.s'$. We now prove that $\Lambda c.s' \in RED_S[\vec{\mathcal{R}}/\vec{X}]$; the required result then holds by the definition of $\mathcal{C}_{RED_S[\vec{\mathcal{R}}/\vec{X}]}(E)$. Let V' be a type and let $\mathcal{R}' \in \mathcal{RC}(V')$. Let E' be such that $\mathcal{C}_{RED_{S'}[\vec{\mathcal{R}}/\vec{X}, \mathcal{R}'/c]}(E')$ holds. We have $\langle (\Lambda c.s')\{V'\}, E' \rangle \rightarrow_v \langle s'\{V'/c\}, E' \rangle$. By induction, $\mathcal{N}(\langle s'\{V'/c\}, E' \rangle)$ holds, therefore we obtain that $\mathcal{N}(\langle (\Lambda c.s')\{V'\}, E' \rangle)$ holds as required.
- In the case $t = t_0\{V'\}$, we have $\Gamma; \Delta \vdash t_0 : \forall c.S'$ with $S = S'\{V'/c\}$ for some S' . Let $t'_0 = t_0\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \ulcorner \vec{E} \urcorner / \vec{k}\}$ and $V'' = V'\{\vec{V}/\vec{X}\}$; we have $t\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \ulcorner \vec{E} \urcorner / \vec{k}\} = t'_0\{V''\}$. We have $\langle t'_0\{V''\}, E \rangle = \langle t'_0, E\{V''\} \rangle$. To conclude, we want to apply the induction hypothesis to t_0 , but first we have to prove that $\mathcal{C}_{RED_{\forall c.S'}[\vec{\mathcal{R}}/\vec{X}]}(E\{V''\})$ holds. Let v be such that $v \in RED_{\forall c.S'}[\vec{\mathcal{R}}/\vec{X}]$. By Lemma 1, we obtain that $RED_{S'\{V'/c\}}[\vec{\mathcal{R}}/\vec{X}] = RED_{S'}[\vec{\mathcal{R}}/\vec{X}, RED_{V'}[\vec{\mathcal{R}}/\vec{X}]/c]$, and from this fact we obtain that $\mathcal{C}_{RED_{S'}[\vec{\mathcal{R}}/\vec{X}, RED_{V'}[\vec{\mathcal{R}}/\vec{X}]/c]}(E)$ holds. Besides, we have that $RED_{V'}[\vec{\mathcal{R}}/\vec{X}] \in \mathcal{RC}(V'')$ holds, and hence by the definition of $RED_{\forall c.S'}[\vec{\mathcal{R}}/\vec{X}]$ we obtain $\mathcal{N}(\langle v\{V''\}, E \rangle)$, i.e., $\mathcal{N}(\langle v, E\{V''\} \rangle)$ holds. Consequently, $\mathcal{C}_{RED_{\forall c.S'}[\vec{\mathcal{R}}/\vec{X}]}(E\{V''\})$ holds, hence we have the required result by using the induction hypothesis on s .
- Suppose $t = \mathcal{K}k.s$. Let $s' = s\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \ulcorner \vec{E} \urcorner / \vec{k}\}$. We then have $t\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \ulcorner \vec{E} \urcorner / \vec{k}\} = \mathcal{K}k.s'$. We have $\langle \mathcal{K}k.s', E \rangle \rightarrow_v \langle s'\{\ulcorner \vec{E} \urcorner / \vec{k}\}, E \rangle$.

By induction, we obtain $\mathcal{N}(\langle s' \{ \ulcorner E \urcorner / k \}, E \rangle)$, therefore $\mathcal{N}(\langle \mathcal{K}k.s', E \rangle)$ holds.

- Suppose $k_i \leftrightarrow s$. Let $s' = s\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \ulcorner E \urcorner / \vec{k}\}$. We then have $t\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \ulcorner E \urcorner / \vec{k}\} = \ulcorner E_i \urcorner \leftrightarrow s'$. The program $\langle \ulcorner E_i \urcorner \leftrightarrow s', E \rangle$ is equivalent to $\langle s', \ulcorner E_i \urcorner \leftrightarrow E \rangle$; we now prove that $\mathcal{N}(\langle s', \ulcorner E_i \urcorner \leftrightarrow E \rangle)$ holds, applying the induction hypothesis to s . To this end, we first have to prove that $\mathcal{C}_{RED_{U_i}[\vec{\mathcal{R}}/\vec{X}]}(\ulcorner E_i \urcorner \leftrightarrow E)$ holds. Let v be such that $v \in RED_{U_i}[\vec{\mathcal{R}}/\vec{X}]$. The program $\langle v, \ulcorner E_i \urcorner \leftrightarrow E \rangle$ is equivalent to $\langle \ulcorner E_i \urcorner \leftrightarrow v, E \rangle$. We have $\langle \ulcorner E_i \urcorner \leftrightarrow v, E \rangle \rightarrow_v \langle v, E_i \rangle$, and since $\mathcal{C}_{RED_{U_i}[\vec{\mathcal{R}}/\vec{X}]}(E_i)$ holds, we obtain $\mathcal{N}(\langle v, E_i \rangle)$. Consequently, $\mathcal{N}(\langle v, \ulcorner E_i \urcorner \leftrightarrow E \rangle)$ holds. □

Theorem 1. *If t is a closed plain term, then $\mathcal{N}(\langle t, \bullet \rangle)$ holds.*

Proof. We have $\mathcal{C}_{\mathcal{R}}(\bullet)$ for any \mathcal{R} , therefore we can use the previous lemma. □

The proof of Theorem 1 is constructive and its computational content is a call-by-value evaluator for plain terms, written in the continuation-passing style. It is obtained in a similar way as evaluators presented in the previous work of Biernacka and Biernacki [4, 5].

In [19], Parigot gives a proof of strong normalization for the second-order (i.e., with the types of System F) $\lambda\mu$ -calculus using a variant of Girard's method of reducibility candidates. Parigot's proof and ours share some similarities, even though the results are quite different in nature (strong normalization vs. termination of a particular strategy) and Parigot's proof is more general: it can be applied to the implicitly typed as well as to the explicitly typed language. The key point in Parigot's proof is the following reducibility candidates characterization result: for all reducibility candidates \mathcal{R} , there exists a set \mathcal{S} of (possibly empty) finite sequences of strongly normalizing terms such that we have $t \in \mathcal{R}$ iff for all $\vec{s} \in \mathcal{R}^\perp$, $t \vec{s}$ is strongly normalizing. The greatest such set \mathcal{S} is denoted by \mathcal{R}^\perp . The characterization result is then used to prove a lemma similar to Lemma 2.

The finite sequences of terms can be seen as (call-by-name) contexts in our setting. Moreover, we notice that $\vec{s} \in \mathcal{R}^\perp$ iff for all $t \in \mathcal{R}$, $t \vec{s}$ is strongly normalizing; this resembles the definition of the reducibility predicates on contexts $\mathcal{C}_{\mathcal{R}}$ in our proof. However, the terms in a sequence \vec{s} have to be strongly normalizing in Parigot's proof, while we do not have a similar requirement on contexts. This fact will have consequences for program extraction; the program extracted from our proof would be an evaluator in CPS style where contexts (continuations) are passed around without being

deconstructed (as in [4]). It would be interesting to see what kind of program can be extracted from Parigot's proof.

2.4 Call by name

The proof method can be adapted to the call-by-name strategy. In this case, the syntax of reduction contexts becomes:

$$\text{CBN contexts: } E ::= \bullet \mid E t \mid E \{S\} \mid \ulcorner E \urcorner \leftarrow E$$

and the reduction rules are modified in that a lambda abstraction and a throwing operation can be applied to an arbitrary term instead of only to a value, in the rules (β_n) and $(throw_n)$ below:

$$\begin{aligned} \langle (\lambda x^S.t_0) t_1, E \rangle &\rightarrow_n \langle t_0\{t_1/x\}, E \rangle & (\beta_n) \\ \langle (\Lambda X.t)\{S\}, E \rangle &\rightarrow_n \langle t\{S/X\}, E \rangle & (\beta_T) \\ \langle \mathcal{K}k.t, E \rangle &\rightarrow_n \langle t\{\ulcorner E \urcorner/k\}, E \rangle & (\text{callcc}) \\ \langle \ulcorner E_0 \urcorner \leftarrow t, E_1 \rangle &\rightarrow_n \langle t, E_0 \rangle & (\text{throw}_n) \end{aligned}$$

The type system is as before, except there are fewer rules for typing contexts. In the remainder of this section we briefly point out the main differences in the proof of termination of evaluation between the call-by-value and the call-by-name strategies.

A reducibility candidate of type S in call by name is a set of values of type S (where values are as in call by value), and the definition of the associated predicate $\mathcal{C}_{\mathcal{R}}$ is the same as in call by value, except that the predicate $\mathcal{N}(\cdot)$ is defined using the call-by-name reduction relation \rightarrow_n .

However, parametric reducibility candidates are defined in a substantially different way, reflecting the call-by-name strategy. Let S be a type, $\vec{X} = ftv(S)$, \vec{T} be a sequence of types of the same size as \vec{X} , and $\vec{\mathcal{R}}$ be reducibility candidates such that each \mathcal{R}_i is of type T_i . We define the parametric reducibility candidate $RED_S[\vec{\mathcal{R}}/\vec{X}]$ of type $S\{\vec{T}/\vec{X}\}$ as follows:

$$\begin{aligned} v \in RED_{X_i}[\vec{\mathcal{R}}/\vec{X}] &\text{ iff } v \in \mathcal{R}_i \\ v \in RED_{S_1 \rightarrow S_2}[\vec{\mathcal{R}}/\vec{X}] &\text{ iff } \forall t. \mathcal{Q}_{RED_{S_1}[\vec{\mathcal{R}}/\vec{X}]}(t) \rightarrow \mathcal{Q}_{RED_{S_2}[\vec{\mathcal{R}}/\vec{X}]}(v t) \\ v \in RED_{\forall c.S}[\vec{\mathcal{R}}/\vec{X}] &\text{ iff } \forall U. \forall S. S \in \mathcal{RC}(U) \rightarrow \\ &\quad \forall E. \mathcal{C}_{RED_S[\vec{\mathcal{R}}/\vec{X}, S/c]}(E) \rightarrow \mathcal{N}(\langle v\{U\}, E \rangle) \end{aligned}$$

with

$$\mathcal{Q}_{\mathcal{R}}(t) = \forall E. \mathcal{C}_{\mathcal{R}}(E) \rightarrow \mathcal{N}(\langle t, E \rangle)$$

The predicate \mathcal{Q} used in the second clause of this definition reflects the fact that a term given as argument to a function is not yet a value (whose reducibility is immediate), but it can be seen as a delayed computation that may be forced later, by putting it in a context.

The main lemma is now formulated as follows:

Lemma 3. *Let t be a plain term such that $\Gamma; \Delta \vdash t : S$ with $\Gamma = x_1 : T_1, \dots, x_n : T_n$ and $\Delta = k_1 : \neg U_1, \dots, k_m : \neg U_m$. Let $\{X_1, \dots, X_p\}$ be the free type variables of S , T_1, \dots, T_n , and U_1, \dots, U_m . Let \vec{V} be a sequence of types of length p , and let $\vec{\mathcal{R}}$ be reducibility candidates such that $\mathcal{R}_i \in \mathcal{RC}(V_i)$ for all $i = 1, \dots, p$. Let \vec{t} be closed terms such that $\vdash t_i : T_i\{\vec{V}/\vec{X}\}$ and $\mathcal{Q}_{RED_{T_i}[\vec{\mathcal{R}}/\vec{X}]}(t_i)$ for all $i = 1, \dots, n$. Next, let \vec{E} be closed contexts such that $\mathcal{C}_{RED_{U_i}[\vec{\mathcal{R}}/\vec{X}]}(E_i)$ for all $i = 1, \dots, m$, and let E be such that $\mathcal{C}_{RED_S[\vec{\mathcal{R}}/\vec{X}]}(E)$. Then $\mathcal{N}(\langle t\{\vec{V}/\vec{X}, \vec{t}/\vec{x}, \vec{E}/\vec{k}\}, E \rangle)$ holds.*

The termination of call-by-name evaluation for closed well-typed programs follows from Lemma 3. As before, the computational content can be extracted from the proof: it is a call-by-name evaluator in CPS.

3 System F with delimited control operators

In this section, we prove termination of the call-by-value evaluation in an extension of the explicitly typed System F with the delimited control operators *shift* and *reset* of Danvy and Filinski [12]. While the abortive control operators such as *callcc* model jumps, *shift* and *reset* allow for delimited-control capture and continuation composition.

3.1 Syntax and semantics

We extend the explicitly typed System F with the operators *shift* S , *reset* $\langle \cdot \rangle$, and *throw* \leftrightarrow . We call the language $\lambda_{S,v}^F$. The syntax of terms, term types, contexts, and metacontexts of $\lambda_{S,v}^F$ is given as follows:

$$\begin{aligned}
\text{Terms: } t & ::= x \mid \lambda x^S.t \mid tt \mid \Lambda X.t \mid t\{S\} \mid \\
& \quad Sk.t \mid \langle t \rangle \mid k \leftrightarrow t \mid \lceil E \rceil \leftrightarrow t \\
\text{Term types: } S & ::= X \mid S_S \rightarrow_S S \mid \forall X.S^{S,S} \\
\text{CBV contexts: } E & ::= \bullet \mid (\lambda x^S.t) E \mid E t \mid E \{S\} \mid \lceil E \rceil \leftrightarrow E \\
\text{Metacontexts: } F & ::= \square \mid E \cdot F
\end{aligned}$$

The new term constructs are the *shift* operator $Sk.t$ binding the continuation variable k in t , and a term delimited by *reset*, denoted $\langle t \rangle$. The remaining term constructs are as before. The (non-standard) syntax of types is discussed in Section 3.2. The syntax of reduction contexts is the same as in Section 2.1, and terms are plugged in contexts using a function *plug*, defined in a similar manner as before.

The new syntactic category is that of metacontexts. A metacontext can be seen as a stack of contexts: \square is the empty metacontext and the metacontext $E \cdot F$ is obtained by pushing the context E on top of F . Each context in the stack is separated from the rest by a delimiter; therefore \square

represents the term with a hole $[\]$, and $E \cdot F$ represents the term $F[\langle E[\] \rangle]$. The meaning of metacontexts is formalized through a function $plug_m$, defined below:

$$\begin{aligned} plug_m(t, \square) &= t \\ plug_m(t, E \cdot F) &= plug_m(\langle plug(t, E) \rangle, F) \end{aligned}$$

The result of $plug_m(t, F)$ is denoted by $F[t]$. Programs are represented as triples consisting of a term, a context, and a metacontext:

$$\text{Programs: } p ::= \langle t, E, F \rangle$$

The program $\langle t, E, F \rangle$ represents the term obtained by plugging t into E and F , i.e., the term $plug_m(plug(t, E), F)$, or $F[\langle E[t] \rangle]$. Such representation is useful to exhibit the decomposition of a given program, where the first component is the term currently in focus, the second component (a context) represents the current continuation up to dynamically nearest enclosing delimiter $\langle \cdot \rangle$, and the metacontext F represents the rest of the computation beyond this delimiter. Before a reduction rule can be applied to a term, the term must be *decomposed* (if possible) into a redex, a context, and a metacontext. As before, we equate programs which represent the same term after plugging.

The call-by-value reduction relation for $\lambda_{S,v}^F$ is defined by the following rules:

$$\begin{aligned} \langle (\lambda x^S.t) v, E, F \rangle &\rightarrow_v \langle t\{v/x\}, E, F \rangle && (\beta_v) \\ \langle (\Lambda X.t)\{S\}, E, F \rangle &\rightarrow_v \langle t\{S/X\}, E, F \rangle && (\beta_T) \\ \langle Sk.t, E, F \rangle &\rightarrow_v \langle t\{\ulcorner E \urcorner/k\}, \bullet, F \rangle && (shift) \\ \langle \ulcorner E' \urcorner \leftrightarrow v, E, F \rangle &\rightarrow_v \langle v, E', E \cdot F \rangle && (throw_v) \\ \langle \langle v \rangle, E, F \rangle &\rightarrow_v \langle v, E, F \rangle && (reset) \end{aligned}$$

where values are defined as before. The first two reduction rules are standard (and insensitive to the surrounding context and metacontext). The rule (*shift*) states that reducing $Sk.t$ consists in capturing the context E and substituting it for the continuation variable k in the body t (the current context is then set to be empty). When a captured context E' is applied to a value (in the rule (*throw_v*)), it is reinstated as the current context, and the then-current context E is pushed on the metacontext F . Finally, the last rule states that when a value is enclosed in a *reset*, it means that the *reset* can be discarded since no further captures can occur inside it.

The evaluation relation \rightarrow_v^* is defined as the reflexive, transitive closure of the reduction relation \rightarrow_v . The expected result of evaluating a program is a program value $p_v := \langle v, \bullet, \square \rangle$.

Typing terms ($S ::= X \mid S_U \rightarrow_V T \mid \forall X.S^{T,U}$):

$$\begin{array}{c}
\frac{}{\Gamma, x : S; \Delta \mid T \vdash x : S \mid T} \quad \frac{\Gamma, x : S; \Delta \mid U \vdash t : T \mid V}{\Gamma; \Delta \mid W \vdash \lambda x^S.t : S_U \rightarrow_V T \mid W} \\
\\
\frac{\Gamma; \Delta \mid X \vdash t_0 : S_U \rightarrow_W T \mid V \quad \Gamma; \Delta \mid W \vdash t_1 : S \mid X}{\Gamma; \Delta \mid U \vdash t_0 t_1 : T \mid V} \\
\\
\frac{\Gamma; \Delta \mid T \vdash t : S \mid U \quad X \notin \text{ftv}(\Gamma) \cup \text{ftv}(\Delta)}{\Gamma; \Delta \mid V \vdash \Lambda X.t : \forall X.S^{T,U} \mid V} \\
\\
\frac{\Gamma; \Delta \mid U\{V/X\} \vdash t : \forall X.S^{T,U} \mid W}{\Gamma; \Delta \mid T\{V/X\} \vdash t\{V\} : S\{V/X\} \mid W} \quad \frac{\Gamma; \Delta \mid U \vdash t : U \mid S}{\Gamma; \Delta \mid T \vdash \langle t \rangle : S \mid T} \\
\\
\frac{\Gamma; \Delta, k : S \triangleright T \mid V \vdash t : V \mid U}{\Gamma; \Delta \mid T \vdash \mathcal{S}k.t : S \mid U} \quad \frac{\Gamma; \Delta, k : S \triangleright T \mid U \vdash t : S \mid V}{\Gamma; \Delta, k : S \triangleright T \mid U \vdash k \leftrightarrow t : T \mid V} \\
\\
\frac{\Gamma; \Delta \vdash E : S \triangleright T \quad \Gamma; \Delta \mid U \vdash t : S \mid V}{\Gamma; \Delta \mid U \vdash \lceil E \rceil \leftrightarrow t : T \mid V}
\end{array}$$

Figure 2: Type system for System F with delimited control operators

3.2 Type system

We add System F types to the type system of Biernacka and Biernacki [5], which is a slight modification of the classical Danvy and Filinski’s type system for *shift* and *reset* [11]. The type system is presented in Figures 2 and 3. In a type $\forall X.S^{T,U}$, the quantifier binds the occurrences of X in S, T , and U . We define the set of free type variables $\text{ftv}(S)$ of a term type S accordingly, and we define $\text{ftv}(S \triangleright T) := \text{ftv}(S) \cup \text{ftv}(T)$.

In this system, contexts are assigned types of the form $S \triangleright T$, where S is the type of the hole and T is the answer type, and metacontexts are assigned types of the form $\neg S$, where S is the type of the hole. A typing judgement $\Gamma; \Delta \mid T \vdash t : S \mid U$ roughly means that under the assumptions Γ and Δ , the term t can be plugged into a context of type $S \triangleright T$ and a metacontext of type $\neg U$ (in general, the evaluation of t may use the surrounding context of type $S \triangleright T$ to produce a value of type U , with $T \neq U$). Because both abstractions $\lambda x^S.t$ and $\Lambda X.t$ denote “frozen” computations—waiting for a term and a type, resp., to activate them—the arrow type and the \forall -type contain additional type annotations. Roughly, the type $S_U \rightarrow_V T$ is assigned to a function that can be applied to an argument of type S within a context of type $T \triangleright U$ and a metacontext of type $\neg V$. Similarly, the

Typing contexts ($C ::= S \triangleright T$):

$$\begin{array}{c}
\frac{}{\Gamma; \Delta \vdash \bullet : S \triangleright S} \quad \frac{\Gamma; \Delta \vdash E : T \triangleright U \quad \Gamma; \Delta \mid V \vdash t : S \mid W}{\Gamma; \Delta \vdash E t : (S_U \rightarrow_V T) \triangleright W} \\
\\
\frac{\Gamma; \Delta \mid W \vdash \lambda x^S.t : S_U \rightarrow_V T \mid W \quad \Gamma; \Delta \vdash E : T \triangleright U}{\Gamma; \Delta \vdash (\lambda x^S.t) E : S \triangleright V} \\
\\
\frac{\Gamma; \Delta \vdash E : S\{V/X\} \triangleright T\{V/X\}}{\Gamma; \Delta \vdash E \{V\} : \forall X.S^{T,U} \triangleright U\{V/X\}} \\
\\
\frac{\Gamma; \Delta \vdash E' : S \triangleright T \quad \Gamma; \Delta \vdash E : T \triangleright U}{\Gamma; \Delta \vdash \lceil E' \rceil \leftrightarrow E : S \triangleright U}
\end{array}$$

Typing metacontexts ($D ::= \neg S$):

$$\frac{}{\Gamma; \Delta \vdash \square : \neg S} \quad \frac{\Gamma; \Delta \vdash E : S \triangleright T \quad \Gamma; \Delta \vdash F : \neg T}{\Gamma; \Delta \vdash E \cdot F : \neg S}$$

Typing 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 for System F with delimited control operators, ctd.

type $\forall X.S^{T,U}$ is assigned to a term that can be applied to a type V within a context of type $S\{V/X\} \triangleright T\{V/X\}$ and a metacontext of type $\neg U\{V/X\}$. It can be shown that closed well-typed terms either are values or decompose uniquely into a redex, a context and a metacontext, and that the reduction rules preserve types.

The type system of Figures 2 and 3 is more liberal than the one defined for $\lambda_2^{s/r, Std}$, a language defined by Asai and Kameyama in [2], which is similar to $\lambda_{S,v}^F$. In [2], polymorphic abstraction types do not contain any additional type annotations, and can only be assigned to abstractions $\Lambda X.t$ where t is a *pure* term, i.e., a term such that $\Gamma; \Delta \mid T \vdash t : S \mid T$ is derivable for any T . Pure terms are terms free from control effects, such as x , $\lambda x^S.t$, $\langle t \rangle$, or $\Lambda X.t$. In contrast, we allow arbitrary abstractions of the form $\Lambda X.t$, at the cost of additional type annotations in the polymorphic abstraction types. As pointed out by Asai and Kameyama, restricting \forall -introduction to pure terms is not mandatory in a calculus with standard call-by-value evaluation, such

as $\lambda_2^{s/r, Std}$ and our calculus. However, such restriction becomes necessary for the calculus $\lambda_2^{s/r, ML}$ of [2] with ML-like call-by-value evaluation (where reduction is allowed under Λ) for subject reduction to hold.

3.3 Termination

The proof of termination is very similar to that of Section 2.3, and here we only briefly point out the main differences. We define the normalization predicate $\mathcal{N}(p)$ as follows:

$$\mathcal{N}(p) := \exists v. p \rightarrow_v^* \langle v, \bullet, \square \rangle$$

A *reducibility candidate* \mathcal{R} of type S is a set of closed values of type S . We write $\mathcal{RC}(S)$ for the set of reducibility candidates of type S . Let \mathcal{R}, \mathcal{S} be reducibility candidates of types S and T , respectively. We define the predicate $\mathcal{C}_{\mathcal{R}, \mathcal{S}}(E)$ on closed contexts of type $S \triangleright T$ and the predicate $\mathcal{M}_{\mathcal{R}}(F)$ on closed metacontexts of type $\neg S$ as follows:

$$\begin{aligned} \mathcal{C}_{\mathcal{R}, \mathcal{S}}(E) &:= \forall v. v \in \mathcal{R} \rightarrow \forall F. \mathcal{M}_{\mathcal{S}}(F) \rightarrow \mathcal{N}(\langle v, E, F \rangle) \\ \mathcal{M}_{\mathcal{R}}(F) &:= \forall v. v \in \mathcal{R} \rightarrow \mathcal{N}(\langle v, \bullet, F \rangle) \end{aligned}$$

Let S be a type, $\vec{X} = ftv(S)$, \vec{T} be a sequence of types of the same size as \vec{X} , and $\vec{\mathcal{R}}$ be reducibility candidates such that each \mathcal{R}_i is of type T_i . We now define the parametric reducibility candidate $RED_S[\vec{\mathcal{R}}/\vec{X}]$ of type $S\{\vec{T}/\vec{X}\}$ as follows:

$$\begin{aligned} v \in RED_{X_i}[\vec{\mathcal{R}}/\vec{X}] &\text{ iff } v \in \mathcal{R}_i \\ v_0 \in RED_{S_{U \rightarrow VT}}[\vec{\mathcal{R}}/\vec{X}] &\text{ iff } \forall v_1. v_1 \in RED_S[\vec{\mathcal{R}}/\vec{X}] \rightarrow \\ &\quad \forall E. \mathcal{C}_{RED_T[\vec{\mathcal{R}}/\vec{X}], RED_U[\vec{\mathcal{R}}/\vec{X}]}(E) \rightarrow \\ &\quad \forall F. \mathcal{M}_{RED_V[\vec{\mathcal{R}}/\vec{X}]}(F) \rightarrow \\ &\quad \mathcal{N}(\langle v_0 v_1, E, F \rangle) \\ v \in RED_{\forall c. S_{T,U}}[\vec{\mathcal{R}}/\vec{X}] &\text{ iff } \forall V. \forall \mathcal{S}. \mathcal{S} \in \mathcal{RC}(V) \rightarrow \\ &\quad \forall E. \mathcal{C}_{RED_S[\vec{\mathcal{R}}/\vec{X}, \mathcal{S}/c], RED_T[\vec{\mathcal{R}}/\vec{X}, \mathcal{S}/c]}(E) \rightarrow \\ &\quad \forall F. \mathcal{M}_{RED_U[\vec{\mathcal{R}}/\vec{X}, \mathcal{S}/c]}(F) \rightarrow \\ &\quad \mathcal{N}(\langle v\{V\}, E, F \rangle) \end{aligned}$$

Using a substitution lemma similar to Lemma 1, we can prove the following result, from which the termination theorem follows for closed plain terms:

Lemma 4. *Let t be a plain term such that $\Gamma; \Delta \mid T \vdash t : S \mid U$, with $\Gamma = x_1 : T_1, \dots, x_n : T_n$, and $\Delta = k_1 : C_1, \dots, k_m : C_m$. Let $\{X_1, \dots, X_p\}$ be the free variables of S, T_1, \dots, T_n , and C_1, \dots, C_m . Let \vec{V} be a sequence of types of the same length as $\{X_1, \dots, X_p\}$, and let $\vec{\mathcal{R}}$ be reducibility candidates such that $\mathcal{R}_i \in \mathcal{RC}(V_i)$. Let \vec{v} be closed values such that $\cdot \mid W \vdash v_i : T_i\{\vec{V}/\vec{X}\} \mid W$ and $v_i \in RED_{T_i}[\vec{\mathcal{R}}/\vec{X}]$ for each i . Let*

\vec{E} be closed contexts such that $;\cdot \vdash E_i : C_i$, $C_i = W_i^1 \triangleright W_i^2$, and $\mathcal{C}_{RED_{W_i^1}[\vec{\mathcal{R}}/\vec{X}], RED_{W_i^2}[\vec{\mathcal{R}}/\vec{X}]}(E_i)$ for each i . Let E be such that $;\cdot \vdash E : S \triangleright T$ and $\mathcal{C}_{RED_S[\vec{\mathcal{R}}/\vec{X}], RED_T[\vec{\mathcal{R}}/\vec{X}]}(E)$. Let F be such that $;\cdot \vdash F : \neg U$ and $\mathcal{M}_{RED_U[\vec{\mathcal{R}}/\vec{X}]}(F)$. Then $\mathcal{N}(\langle t\{\vec{V}/\vec{X}, \vec{v}/\vec{x}, \vec{E}/\vec{k}\}, E, F \rangle)$ holds.

Again, the computational content of this proof takes the form of an evaluator in CPS, this time with two layers of continuations, as shown in our previous work [5]. Also, one could quite easily repeat the developments of this section for the call-by-name *shift* and *reset*, by extending the type system defined elsewhere [5] to System F.

4 Conclusion

We have shown that the context-based proof method developed by the first two authors for simply-typed calculi with control operators, be they abortive or delimited, scales rather seamlessly to much more expressive type systems based on System F. The presented proofs are quite simple and elegant, and they do not require a journey through an optimized CPS translation in order to show termination of evaluation for such calculi. Furthermore, they give rise to certified evaluators in CPS by program extraction from proofs [3, 7], and can lead to executable specifications of programming languages with control operators and polymorphism, for example, if formalized in a logical framework equipped with program extraction mechanism, such as Coq. We leave such a formalization as a future work.

Acknowledgements

This work has been supported by the MNiSW grant number N N206 357436, 2009-2011. The third author has been supported by the Alain Bensoussan Fellowship Program.

References

- [1] Z. M. Ariola, H. Herbelin, and A. Sabry. A proof-theoretic foundation of abortive continuations. *Higher-Order and Symbolic Computation*, 20(4):403–429, 2007.
- [2] K. Asai and Y. 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, Dec. 2007.

- [3] U. Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, Mar. 1993. Springer-Verlag.
- [4] M. Biernacka and D. 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, Apr. 2009.
- [5] M. Biernacka and D. Biernacki. Context-based proofs of termination for typed delimited-control operators. In F. J. López-Fraguas, editor, *Proceedings of the 11th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, Coimbra, Portugal, Sept. 2009. ACM Press.
- [6] M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, Nov. 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).
- [7] M. Biernacka, O. Danvy, and K. Støvring. Program extraction from proofs of weak head normalization. In M. Escardó, A. Jung, and M. 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.
- [8] V. Danos, J.-B. Joinet, and H. Schellinx. A new deconstructive logic: Linear logic. *Journal of Symbolic Logic*, 62(3):755–807, 1997.
- [9] O. Danvy. On evaluation contexts, continuations, and the rest of the computation. In H. 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, pages 13–23, Venice, Italy, Jan. 2004. Invited talk.
- [10] O. Danvy. Defunctionalized interpreters for programming languages. In P. Thiemann, editor, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, SIGPLAN Notices, Vol. 43, No. 9, Victoria, British Columbia, Sept. 2008. ACM Press. Invited talk.

- [11] O. Danvy and A. Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.
- [12] O. Danvy and A. Filinski. Abstracting control. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.
- [13] M. Felleisen and D. P. Friedman. Control operators, the SECD machine, and the λ -calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.
- [14] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
- [15] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
- [16] T. G. Griffin. A formulae-as-types notion of control. In P. Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, Jan. 1990. ACM Press.
- [17] B. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In S. L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, South Carolina, Jan. 1993. ACM Press.
- [18] R. Harper, B. F. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, Oct. 1993. A preliminary version was presented at the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1991).
- [19] M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *Journal of Symbolic Logic*, 62(4):1461–1479, 1997.
- [20] H. Schwichtenberg. Proofs, lambda terms and control operators. In H. Schwichtenberg, editor, *Logic of Computation, volume 157 of Series F: Computer and Systems Sciences, Proceedings of the NATO Advanced Study Institute on Logic of Computation, Marktoberdorf, Germany, July 25 - August 6, 1995*, pages 309–347. Springer Verlag, 1997.
- [21] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32:198–212, 1967.

- [22] A. S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, 1973.
- [23] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.