Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 249 (2009) 169–192
www.elsevier.com/locate/entcs

A Context-based Approach to Proving Termination of Evaluation

Małgorzata Biernacka [1] Dariusz Biernacki [2]

Institute of Computer Science
University of Wrocław
Wrocław, Poland

Abstract

We present a context-based approach to proving termination of evaluation in reduction semantics (i.e., a form of operational semantics with explicit representation of reduction contexts), using Tait-style reducibility predicates defined on both terms and contexts. We consider the simply typed lambda calculus as well as its extension with abortive control operators for first-class continuations under the call-by-value and the call-by-name evaluation strategies. For each of the proofs we present its computational content that takes the form of an evaluator in continuation-passing style and is an instance of normalization by evaluation.

Keywords: reduction semantics, evaluation context, weak head normalization, control operators, normalization by evaluation

1 Introduction

In the term-rewriting setting, a typical presentation of the lambda calculus as a prototypical programming language relies on the grammar of terms and a reduction relation defined on these terms. Felleisen et al. have introduced the notion of reduction/evaluation contexts [15–17] that proved useful in expressing various reduction strategies concisely, building on the notion of context as a term with a hole [2]. Felleisen's contexts represent "the surrounding term" of the current subterm, or "the rest of the computation", and they directly correspond to continuations: the latter can be seen as functional representations of contexts. More precisely, Danvy observed that reduction contexts arise as defunctionalized continuations of a one-step reduction function whereas evaluation contexts arise as defunctionalized continuations of an evaluation function (i.e., big-step) [11, 12]. Since these defunctionalized representations of continuations are in both cases the same, the terms "evaluation

[1] Email: mabi@ii.uni.wroc.pl
[2] Email: dabi@ii.uni.wroc.pl

doi:10.1016/j.entcs.2009.07.090

context" and "reduction context" are usually used interchangeably, and we will adhere to this practice in the remainder of this article.

Because of their close relation to continuations, the benefits of using contexts can be seen perhaps most prominently in languages with control operators, i.e., syntactic constructs that manipulate the current continuation/context [15]. Moreover, as shown by Wright and Felleisen [27], context-based reduction semantics of a programming language provide a convenient formalism for expressing and proving type soundness properties.

In this article we present yet another application of contexts: we give novel proofs of termination of evaluation in the simply typed lambda calculus under the call-by-value and call-by-name reduction strategies where reduction contexts play a major role. Subsequently we extend the simply typed lambda calculus with common abortive control operators: *callcc*, *abort* and Felleisen's $\mathcal{C}$ and we use the same approach as for the pure lambda calculus to prove termination for the extended language, using its standard context-based reduction semantics.

The method of proof we apply in this work—using context-based variant of Tait-style reducibility predicates [25]—is a modification of the method considered in a previous work of Biernacka et al. that used "direct-style" reducibility predicates [9]. In effect, we obtain direct, simple proofs of termination that take advantage of the context-based formulation of the reduction semantics. In contrast, many of the existing proofs of normalization properties for typed lambda calculi with control operators are indirect and they use a translation to another language already known to be normalizable [1, 18, 24]. This line of work on proof-theoretic properties of typed control operators was originated by Griffin who gave a type assignment to Felleisen's $\mathcal{C}$ operator, *abort* and *callcc*, and who proved termination of evaluation for his language using a translation to the simply typed lambda calculus akin to Plotkin's colon translation [18].

On the other hand, the method of proving normalization using Tait-style reducibility predicates has been applied to the pure lambda calculus, both for weak and strong normalization [5,25,26] as well as for weak head normalization under call by name (essentially due to Martin-Löf) and call by value (due to Hoffmann) [9]. An extension to control operators has been considered by Parigot who modified Girard's reducibility candidates to prove strong normalization for his second-order $\lambda\mu$-calculus corresponding to classical natural deduction [21]. Berger and Schwichtenberg identified the computational content of their constructive proof of strong normalization that uses the reducibility method to be an instance of normalization by evaluation, and subsequently this observation has been applied to proofs of weak head normalization by Coquand and Dybjer for combinatory logic [10] and by Biernacka et al. for the lambda calculus [9]. Some of the proofs have been formalized in proof assistants and normalizers have been extracted from them in the form of functional programs [4,6]. Not surprisingly, the computational content of our proofs are instances of normalization by evaluation; the extracted programs are evaluators in continuation-passing style, whose continuations arise by extraction from a context reducibility predicate. Thus the present article provides a logical confirmation of

the connection between continuations and contexts, previously observed and investigated by Danvy [11,12].

2 The simply typed lambda calculus

In this section we present two proofs of weak head normalization for the simply typed lambda calculus using context-based reducibility predicates à la Tait. We consider closed terms and two strategies: call by value (i.e., applicative order) and call by name (i.e., normal order). Contrary to previous work, we use a different formulation of logical predicates: instead of a type-indexed family of reducibility predicates on terms, we define two such families: one for terms and one for evaluation contexts. This formulation relies on the fact that we define programs as pairs consisting of a term and an evaluation context, and evaluation contexts are part of the syntax of the language. The specificity of this approach is that the definition of reducibility predicates differs for each evaluation strategy. The proofs themselves seem to be even easier to carry out than the proofs using the standard reducibility predicates. Finally, an—expected—consequence of this approach is that the computational content of the proofs (i.e., the extracted program) are evaluators in continuation-passing style. These CPS evaluators can be otherwise obtained by CPS-translating the evaluators extracted from the standard proofs (in both the call-by-value and call-by-name strategies).

2.1 Terms: syntax and typing

We introduce terms and reduction contexts as two syntactic categories, where the syntax of terms is standard:

$$(\textit{terms}) \quad t ::= x \mid \lambda x.t \mid t\,t$$

and the syntax of reduction contexts depends on the strategy we choose for reduction (in fact, the grammar of reduction contexts reflects the reduction strategy). Because of that, we postpone the actual definitions of reduction contexts for call by value and call by name to Section 2.2 and Section 2.3, respectively.

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

Next, we define a typing relation for terms, again in the standard way. Types are either base types, or arrow types:

$$(\textit{types}) \quad A ::= b \mid A \to A$$

and the typing relation on terms is given by the following inference system, where Γ is the usual typing environment associating free variables with their types:

$$\frac{}{\Gamma,\, x : A \vdash x : A} \qquad \frac{\Gamma,\, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\frac{\Gamma \vdash t_0 : A \to B \quad \Gamma \vdash t_1 : A}{\Gamma \vdash t_0\, t_1 : B}$$

2.2 The call-by-value reduction strategy

2.2.1 Contexts

Given the grammar and the typing of terms from Section 2.1, we now define call-by-value reduction contexts as follows:

$$(\mathit{CBV\ contexts})\ \ \mathrm{E} ::= \bullet \mid v\,\mathrm{E} \mid \mathrm{E}\,t$$

$$(\mathit{values})\ \ \ v ::= \lambda x.t$$

where values form a subcategory of terms and are used to denote normal forms.

Contexts are part of the syntax and not just a metarepresentation of "terms with a hole". They are represented inside-out, i.e.: $\bullet$ represents the empty context, $v\ \mathrm{E}$ represents the "term with a hole" $\mathrm{E}[v\,[\,]]$ (in an informal notation), and $\mathrm{E}\ t$ represents the "term with a hole" $\mathrm{E}[[\,]\,t]$. We say a reduction context is closed, if its constituent terms are all closed.

In order to formalize the meaning of contexts, we define the function *plug* mapping a term and a context to the term such a pair represents:

$$\begin{aligned} \mathit{plug}\ (t, \bullet) &= t \\ \mathit{plug}\ (t, v\ \mathrm{E}) &= \mathit{plug}\ (v\,t, \mathrm{E}) \\ \mathit{plug}\ (t_0, \mathrm{E}\ t_1) &= \mathit{plug}\ (t_0\,t_1, \mathrm{E}) \end{aligned}$$

We write the result of plugging the term t in the context E in the usual way: $\mathrm{E}[t]$.

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

$$(\mathit{programs})\ \ p ::= \langle t, \mathrm{E}\rangle$$

The program $\langle t, \mathrm{E}\rangle$ represents the term obtained by plugging the term t into the context E, i.e., the term $\mathrm{E}[t]$. This representation allows us to represent all lambda terms (and only lambda terms) in such a way that we explicitly state the "boundary" of a program (or, top level); note that we do not have a way to compose programs, so we cannot obtain a bigger program by plugging one program into another reduction context—which is possible if we treat terms as programs in the usual way. While this choice of representation does not matter for the pure lambda calculus, it will play a significant role later on, when we extend the language with abortive control operators (cf. Section 3).

Of course, according to the definition of program, various pairs of a term and a context can represent the same "plugged term", i.e., the application of the function *plug* to different pairs may give the same lambda term as a result. From the point of view of computation, all such pairs will be regarded as various representations of the same program. Therefore, from now on, we will consider programs as abstraction classes of the equivalence relation between well-typed pairs defined as follows:

$$\langle t_0, \mathrm{E}_0\rangle \sim \langle t_1, \mathrm{E}_1\rangle := \mathrm{E}_0[t_0] = \mathrm{E}_1[t_1]$$

where the equality on the right-hand side denotes syntactic equality modulo alpha renaming. For example, the program $\langle (\lambda x.r)\ s, \bullet \rangle$ can be otherwise represented by another program $\langle \lambda x.r, (\bullet\ s) \rangle$ or by $\langle s, ((\lambda x.r)\ \bullet) \rangle$. All these representations correspond to different *decompositions* of the same term.

Next, we introduce a typing relation on reduction contexts in a way consistent with the standard typing of lambda terms.
Types of contexts are defined using the following syntax:

$$(\textit{context types})\quad T ::= \mathsf{cont}\ A$$

and the typing relation on contexts is defined by the following inference system:

$$\frac{}{\Gamma \vdash \bullet : \mathsf{cont}\ A} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash \mathrm{E} : \mathsf{cont}\ B}{\Gamma \vdash \mathrm{E}\ t : \mathsf{cont}\ (A \to B)}$$

$$\frac{\Gamma \vdash v : A \to B \quad \Gamma \vdash \mathrm{E} : B}{\Gamma \vdash v\ \mathrm{E} : \mathsf{cont}\ A}$$

It is not difficult to see that function *plug* ensures and preserves well-typedness of terms in the way formalized by the following lemma.

Lemma 2.1 *The following hold:*

(i) *If* $\Gamma \vdash t : A$ *and* $\Gamma \vdash \mathrm{E} : \mathsf{cont}\ A$, *then there exists a type* B *such that* $\Gamma \vdash \mathrm{E}[t] : B$.

(ii) *If* $\Gamma \vdash \mathrm{E}[t] : B$, *then* $\Gamma \vdash t : A$ *and* $\Gamma \vdash \mathrm{E} : \mathsf{cont}\ A$ *for some type* A.

Proof. The proof is done by induction on the structure of E. □

The type of the program $\langle t, \mathrm{E} \rangle$ can naturally be defined to be the type of the term $\mathrm{E}[t]$; the following rule for typing programs is well defined (it does not depend on the choice of a particular pair from an abstraction class):

$$\frac{\Gamma \vdash \mathrm{E}[t] : A}{\Gamma \vdash \langle t, \mathrm{E} \rangle : A}$$

The type $\mathsf{cont}\ A$ could be interpreted as in Griffin's work [18], i.e., as $\neg A$ $(A \to \bot)$, if we included $\bot$ in the grammar of types (interpreted as formulas through the Curry-Howard isomorphism [20]). However, according to the above rule, $\bot$ would play no role in typing programs.

Finally, we observe that the class of well-typed programs defines exactly the set of simply typed lambda terms.

2.2.2 Reduction

The grammar of contexts defined in the previous subsection determines the call-by-value reduction strategy for evaluation. We define a one-step reduction relation on programs as follows:

$$\langle (\lambda x.r)\ v, \mathrm{E} \rangle \to_v \langle r\{v/x\}, \mathrm{E} \rangle$$

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. Terms of the form $(\lambda x.r)\, v$ are the familiar call-by-value *β-redexes*.

Thanks to the unique-decomposition property of the lambda calculus under call by value, the relation $\rightarrow_v$ is deterministic and it is a function on abstraction classes.

Property 1 (Unique decomposition (CBV)) *For all terms t, t either is a value, or it decomposes* uniquely *into a CBV reduction context* E *and a redex*[3] *r, i.e., $t = \mathrm{E}[r]$.*

Next, we define the evaluation relation as the reflexive-transitive closure of one-step reduction ($\rightarrow_v^*$). The result of the evaluation is a (program) value of the form $p_v := \langle v, \bullet \rangle$.

It is easy to see that there is an exact correspondence between reductions of programs in this sense and reductions of terms in the usual sense, according to the following lemma.

Lemma 2.2 *For each program $p := \langle t, \mathrm{E} \rangle$, p reduces to another program $p' := \langle t', \mathrm{E}' \rangle$ if and only if the simply typed lambda term $\mathrm{E}[t]$ reduces to the term $\mathrm{E}'[t']$ under the standard CBV reduction strategy.*

The reduction relation preserves types of programs, because of the subject reduction property for simply typed lambda terms: the type of a β-redex is preserved after the reduction.

Corollary 2.3 (Progress and Preservation) *For each program p, p either is a value or it reduces uniquely to another program p' such that if $\Gamma \vdash p : A$, then $\Gamma \vdash p' : A$.*

2.2.3 Termination

We now give a proof of termination for call-by-value evaluation that uses logical predicates in the style of Tait but based on contexts as well as on terms rather than on terms only. From now on, for simplicity, we only consider closed programs, although the method generalizes to open well-typed terms.

We first introduce two mutually inductive logical predicates: $\mathcal{R}_A$ is defined on closed values of type A, and $\mathcal{C}_{\mathsf{cont}\, A}$ is defined on closed contexts of type $\mathsf{cont}\, A$ as follows:

$$\begin{aligned} \mathcal{R}_b(v) &:= True \\ \mathcal{R}_{A\to B}(v_0) &:= \forall v_1.\, \mathcal{R}_A(v_1) \to \forall \mathrm{E}.\, \mathcal{C}_{\mathsf{cont}\, B}(\mathrm{E}) \to \mathcal{N}(\langle v_0\, v_1, \mathrm{E} \rangle) \end{aligned}$$

$$\mathcal{C}_{\mathsf{cont}\, A}(\mathrm{E}) := \forall v.\, \mathcal{R}_A(v) \to \mathcal{N}(\langle v, \mathrm{E} \rangle)$$

where

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

[3] More precisely, a decomposition is in general a context and a potential redex, i.e., a proper redex that can be contracted, or a "stuck" term. We ignore this issue here, since the languages considered in this article do not contain stuck terms.

In the standard approach, the reducibility predicate on well-typed terms expresses the property that whenever a reducible term is applied to another reducible term of the right type, the resulting term has also this property. Moreover, if a term is reducible, then it normalizes. The proof of termination consists in showing that all well-typed terms are reducible, from which it follows that all well-typed terms normalize.

Here, we prove normalization using a modified version of the reducibility predicate, noted $\mathcal{R}_A$. First of all, we only need to define this property on well-typed values (we could extend it to all well-typed terms, but it is not necessary for the proof). A reducible value is such that, when applied to another reducible value, and paired with a reducible context, normalizes as a program. Simultaneously, we define a reducibility predicate on well-typed reduction contexts, $\mathcal{C}_{\mathsf{cont}\,A}$, saying that any reducible value in a reducible context normalizes as a program. The typing properties ensure that the programs occurring in the definitions of the predicates are all well typed, but we do not need to know their type in order to prove the normalization theorem.

Lemma 2.4 *Let t be a well-typed term such that $x_1 : B_1, \ldots, x_n : B_n \vdash t : A$. Next, let $\vec{v}$ be a sequence of closed well-typed value terms such that $\vdash v_i : B_i$ and $\mathcal{R}_{B_i}(v_i)$ for $1 \leq i \leq n$. Then for all closed well-typed reduction contexts* E *such that $\vdash$ E : $\mathsf{cont}\,A$ and $\mathcal{C}_{\mathsf{cont}\,A}(\mathrm{E})$, the program $\langle t\{\vec{v}/\vec{x}\}, \mathrm{E}\rangle$ normalizes, i.e., $\mathcal{N}(\langle t\{\vec{v}/\vec{x}\}, \mathrm{E}\rangle)$ holds. (Notation $t\{\vec{v}/\vec{x}\}$ stands for the simultaneous substitution of each value term v_i for the free variable x_i in 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 $t\{\vec{v}/\vec{x}\} = v_i$. Hence, by assumption $\mathcal{R}_A(v_i)$ and for any E such that $\mathcal{C}_{\mathsf{cont}\,A}(\mathrm{E})$ holds, unfolding the definition of $\mathcal{C}_{\mathsf{cont}\,A}$ entails that $\mathcal{N}(\langle v_i, \mathrm{E}\rangle)$ holds.

Case $\lambda x.r$**.** Because $\lambda x.r$ is well typed, its type A must be an arrow type; let $A = A' \to A''$. Taking $r' = r\{\vec{v}/\vec{x}\}$, we have $(\lambda x.r)\{\vec{v}/\vec{x}\} = \lambda x.r'$. We will show that $\mathcal{R}_A(\lambda x.r')$ holds, and from this fact it follows that the required $\mathcal{N}(\langle(\lambda x.r)\{\vec{v}/\vec{x}\}, \mathrm{E}\rangle)$ holds as in the previous case. In order to prove $\mathcal{R}_A(\lambda x.r')$, let us assume that v is a value of type A' and such that $\mathcal{R}_{A'}(v)$ holds. Next, let E be a well-typed context of type A'' and such that $\mathcal{C}_{\mathsf{cont}\,A''}(\mathrm{E})$ holds. We have to prove that $\mathcal{N}(\langle(\,\lambda x.r')v, \mathrm{E}\rangle)$ holds. By the reduction rule, $\langle(\lambda x.r')\,v, \mathrm{E}\rangle$ reduces in one step to program $\langle r\{\vec{v}/\vec{x}, v/x\}, \mathrm{E}\rangle$. By induction hypothesis, $\mathcal{N}(\langle r\{\vec{v}/\vec{x}, v/x\}, \mathrm{E}\rangle)$ holds and hence also $\mathcal{N}(\langle(\lambda x.r')\,v, \mathrm{E}\rangle)$ holds.

Case $t_0\,t_1$**.** Since $t_0\,t_1$ is well typed, then $x_1 : B_1, \ldots, x_n : B_n \vdash t_0 : C \to A$ and $x_1 : B_1, \ldots, x_n : B_n \vdash t_1 : C$ for some type C. Taking $t_0' = t_0\{\vec{v}/\vec{x}\}$ and $t_1' = t_1\{\vec{v}/\vec{x}\}$, we have $(t_0\,t_1)\{\vec{v}/\vec{x}\} = t_0'\,t_1'$. By definition, the program $\langle t_0'\,t_1', \mathrm{E}\rangle$ is the same as the program represented by $\langle t_0', \mathrm{E}\;t_1'\rangle$. Since t_0 is a subterm of $t_0\,t_1$, we can apply the induction hypothesis to deduce $\mathcal{N}(\langle t_0', \mathrm{E}\;t_1'\rangle)$ provided that $\mathrm{E}\;t_1'$ is well typed and that $\mathcal{C}_{\mathsf{cont}\,(C\to A)}(\mathrm{E}\;t_1')$ holds. The former is easy to see, and for the latter let us unfold the definition of $\mathcal{C}_{\mathsf{cont}\,(C\to A)}$. Let v be a value of type $C \to A$ and such that $\mathcal{R}_{C\to A}(v)$ holds. We need to show that $\mathcal{N}(\langle v, \mathrm{E}\;t_1'\rangle)$ holds.

Here again we can use another representative of the class of programs equal to $\langle v, \mathrm{E}\ t_1' \rangle$, such as $\langle t_1', v\ \mathrm{E} \rangle$. Now we can apply the induction hypothesis again, this time for t_1, provided that $v\ \mathrm{E}$ is well typed and $\mathcal{C}_{\mathsf{cont}\,C}(v\ \mathrm{E})$ holds. And again, the former property is easy to see, and for the latter we again unfold the definition of $\mathcal{C}_{\mathsf{cont}\,C}$: let v' be a value of type C and such that $\mathcal{R}_C(v')$ holds. We now need to show that $\mathcal{N}(\langle v', v\ \mathrm{E} \rangle)$ holds. But this is equivalent to showing that $\mathcal{N}(\langle v\ v', \mathrm{E} \rangle)$ holds, and this property follows from the fact that $\mathcal{R}_{C \to A}(v)$ holds by an earlier assumption.

□

Theorem 2.5 (Termination of CBV evaluation) *If t is a closed well-typed term, then $\mathcal{N}(\langle t, \bullet \rangle)$ holds.*

Proof. It is straightforward to see that the empty context satisfies $\mathcal{C}_{\mathsf{cont}\,A}$ for any type A. From Lemma 2.4 it follows that if we take a closed well-typed term t and put it in the empty context, then the resulting program evaluates to a value program. □

It follows that all closed well-typed terms evaluate to a value in the standard sense.

2.2.4 Extracted evaluator

The specification of the normalization problem and the proof of Theorem 2.5 can be formalized in a number of ways and its computational content can be extracted in the form of a lambda term that can be interpreted as an evaluator for the object language [3–6, 9]. In this work, our interest lies not in completely formalizing the problem—it can easily be done, e.g., along the lines of the work cited above—but in showing another way of proving normalization using a context-based approach. Therefore we conduct the development on an informal level and we only outline the program that can be extracted from the proof of Theorem 2.5. The basic idea of program extraction relies on the Curry-Howard correspondence between proofs and programs: roughly, we can view the proof of Theorem 2.5 as a lambda term (the proof is constructive). In this proof term, some parts represent logical inferences and some parts can be seen as computations (here, these computations serve to build the normal form of a given term). Erasing the logical parts, we obtain a lambda term that only contains computationally relevant parts of the original proof, and it is this term that we call the "extracted" program—in our case, an evaluator, i.e., a program computing weak head normal forms of lambda terms. This is essentially what the modified realizability interpretation does to a proof term to extract its computational content [3, 9].

If we apply this method of extraction to the proof term for Theorem 2.5, we obtain a program that normalizes simply-typed lambda terms into values according to the call-by-value strategy. The program extracted from the proof of Lemma 2.4

is in continuation-passing style and its structure is the following:

$$\begin{aligned}
\mathbf{eval}^{\vec{x}}\ x_i &= \lambda\vec{v}\vec{u}\kappa.\kappa\ v_i\ u_i \\
\mathbf{eval}^{\vec{x}}\ \lambda x.t &= \lambda\vec{v}\vec{u}\kappa.\kappa\ ((\lambda x.t)\{\vec{v}/\vec{x}\})\ (\lambda v u\kappa.\mathbf{eval}^{\vec{x}x}\ t\ (\vec{v}v)\ (\vec{u}u)\ \kappa) \\
\mathbf{eval}^{\vec{x}}\ t_0\ t_1 &= \lambda\vec{v}\vec{u}\kappa.\mathbf{eval}^{\vec{x}}\ t_0\ \vec{v}\vec{u}\ (\lambda v_0 u_0.\mathbf{eval}^{\vec{x}}\ t_1\ \vec{v}\vec{u}\ (\lambda v_1 u_1.u_0\ v_1 u_1\ \kappa))
\end{aligned}$$

The evaluator is parameterized by the vector of free variables occurring in a term ($\vec{x}$) and it uses two environments: one ($\vec{v}$) containing values to be substituted for the free variables $\vec{x}$ in the evaluated term, and one ($\vec{u}$) containing functions (these functions arise as the computational content of the relations $\mathcal{R}_A$ for appropriate A). The substitutions are needed in the final step of computation when we have to return a value as a closed term—this is the only place where the first environment plays a role. But whenever a lambda abstraction in the object language is applied to a value, instead of substitution, we apply the suitable function from the second environment and evaluate the body of the lambda abstraction with the given argument. (Note that apart from supplying the argument as a syntactic value, the function evaluating the body of an abstraction expects another function that knows how to evaluate the body of the supplied argument—a value—in case it becomes applied in the future.) Therefore, this evaluator is an instance of normalization by evaluation—normalization (reduction) in the source language is done by evaluation on the metalevel.

Continuations (κ) in the evaluator arise as the computational content of the relations $\mathcal{C}_{\mathsf{cont}\,A}$ for appropriate A. The syntactic representations of contexts we used in the proof can be optimized away (i.e., simply erased) since they do not play any role in the evaluator. This optimization is not arbitrary—it is provably correct and it corresponds to Berger's optimization to eliminate unused object variables, based on distinguishing between computationally relevant and irrelevant variables [3]. Without this optimization, the extracted evaluator would thread an additional argument—the context—which is never used. [4]

The function **eval** is the computational content of the proof of Lemma 2.4. In the proof of Theorem 2.5, we apply Lemma 2.4 with the empty sequence of values and with the empty context to obtain the proof of $\mathcal{N}(\langle t, \bullet\rangle)$. Thus the program extracted from the proof of the fact $\mathcal{C}_{\mathsf{cont}\,A}(\bullet)$ is the initial continuation with which we activate the **eval** function. It is easy to observe that this initial continuation is the function $\lambda v u.v$; as expected, this initial continuation immediately returns the value it is passed as argument (here, it is also passed a function associated with the value which is ignored).

The complete evaluator therefore can be written as follows:

$$\mathbf{norm}\ t = \mathbf{eval}^{\epsilon}\ t\ \epsilon\epsilon\kappa_{init}$$

where $\kappa_{init} = \lambda v u.v$ and ϵ denotes the empty sequence.

According to the normalization-by-evaluation nomenclature, the **eval** function "reflects" object-level terms at the metalevel (as functions accepting two environ-

[4] In contrast, the representations of contexts are essential in the evaluators for control operators presented in Section 3.

ments and a continuation) and the application to the initial (empty) environments and the initial continuation is the "reification" of metaobjects at the object level.

The evaluator extracted from the proof is in continuation-passing style, i.e., all computations are sequentialized and their intermediate results are named. In this case, the order of evaluation imposed by using continuations is call by value.

2.3 The call-by-name reduction strategy

The development for the call-by-name reduction strategy is done along the same lines as the one for call by value, modulo necessary adjustments. In this subsection, we only give a brief account of call by name, pinpointing the main differences with the previous subsection.

2.3.1 Syntax and typing

The terms are the same as in call by value, but reduction contexts have to be defined differently:

$$(\textit{CBN contexts})\quad \mathrm{E} ::= \bullet \mid \mathrm{E}\ t$$

In call by name, we do not have the context $v\ \mathrm{E}$ and so the plug function has fewer cases. The typing relation for CBN contexts is a subset of the inference rules for CBV contexts.

The notion of program and its typing are defined as in the CBV case, using the equivalence relation on pairs of terms and CBN contexts. All the typing properties stated in Section 2.2.1 hold for call by name as well.

2.3.2 Reduction and termination

The one-step reduction relation for the call-by-name strategy differs in that a lambda abstraction can be applied to an arbitrary term instead of to a value:

$$\langle (\lambda x.r)\ t, \mathrm{E}\rangle \to_n \langle r\{t/x\}, \mathrm{E}\rangle$$

All the above adjustments are standard and the properties analogous to those of Section 2.2.2 hold for call-by-name as well. Next we need to define the logical relations needed for the proof of termination for the call-by-name case:

$$\begin{aligned}
\mathcal{R}_b(v) &:= \mathit{True}\\
\mathcal{R}_{A\to B}(v) &:= \forall t.\, \mathcal{Q}_A(t) \to \mathcal{Q}_B(v\ t)\\
\mathcal{Q}_A(t) &:= \forall \mathrm{E}.\, \mathcal{C}_{\mathsf{cont}\ A}(\mathrm{E}) \to \mathcal{N}(\langle t, \mathrm{E}\rangle)\\
\mathcal{C}_{\mathsf{cont}\ A}(\mathrm{E}) &:= \forall v.\, \mathcal{R}_A(v) \to \mathcal{N}(\langle v, \mathrm{E}\rangle)
\end{aligned}$$

where

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

Here, we also define two main logical predicates: $\mathcal{R}_A$ on closed values of type A and $\mathcal{C}_{\mathsf{cont}\ A}$ on closed contexts of type $\mathsf{cont}\ A$. The auxiliary predicate $\mathcal{Q}_A$ is defined

on closed terms of type A and it expresses the property that a term in any context satisfying $\mathcal{C}_{\mathsf{cont}\,A}$ normalizes (as a program).

We are now ready to state the main result of this section.

Lemma 2.6 *Let t be a well-typed term such that $x_1 : B_1, \ldots, x_n : B_n \vdash t : A$. Next, let $\vec{t}$ be a sequence of closed well-typed terms such that $\vdash t_i : B_i$ and $\mathcal{Q}_{B_i}(t_i)$ for $1 \leq i \leq n$. Then for all closed well-typed reduction contexts E such that $\vdash \mathrm{E} : \mathsf{cont}\,A$ and $\mathcal{C}_{\mathsf{cont}\,A}(\mathrm{E})$, the program $\langle t\{\vec{t}/\vec{x}\}, \mathrm{E}\rangle$ normalizes, i.e., $\mathcal{N}(\langle t\{\vec{t}/\vec{x}\}, \mathrm{E}\rangle)$ holds.*

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 $t\{\vec{t}/\vec{x}\} = t_i$. Hence, by assumption $\mathcal{Q}_A(t_i)$ and for any E such that $\mathcal{C}_{\mathsf{cont}\,A}(\mathrm{E})$ holds, unfolding the definition of $\mathcal{Q}_A(t_i)$ entails that $\mathcal{N}(\langle t_i, \mathrm{E}\rangle)$ holds.

Case $\lambda x.r$**.** Because $\lambda x.r$ is well typed, its type A must be an arrow type; let $A = A' \to A''$. Taking $r' = r\{\vec{t}/\vec{x}\}$, we have $(\lambda x.r)\{\vec{t}/\vec{x}\} = \lambda x.r'$. We will show that $\mathcal{R}_A(\lambda x.r')$ holds, and from this fact, by unfolding the definition of $\mathcal{C}_{\mathsf{cont}\,A}(\mathrm{E})$, it follows that the required $\mathcal{N}(\langle(\lambda x.r)\{\vec{t}/\vec{x}\}, \mathrm{E}\rangle)$ holds. In order to prove $\mathcal{R}_A(\lambda x.r')$, let us assume that s is a well-typed term of type A' and such that $\mathcal{Q}_{A'}(s)$ holds. Next, let E be a well-typed context of type A'' and such that $\mathcal{C}_{\mathsf{cont}\,A''}(\mathrm{E})$ holds. We have to prove that $\mathcal{N}(\langle(\lambda x.r')\, s, \mathrm{E}\rangle)$. By the reduction rule, $\langle(\lambda x.r')\, s, \mathrm{E}\rangle$ reduces in one step to program $\langle r\{\vec{t}/\vec{x}, s/x\}, \mathrm{E}\rangle$. By induction hypothesis, $\mathcal{N}(\langle r\{\vec{t}/\vec{x}, s/x\}, \mathrm{E}\rangle)$ holds and hence also $\mathcal{N}(\langle(\lambda x.r')\, s, \mathrm{E}\rangle)$ holds.

Case $t_0\, t_1$**.** Since $t_0\, t_1$ is well typed, then $x_1 : B_1, \ldots, x_n : B_n \vdash t_0 : C \to A$ and $x_1 : B_1, \ldots, x_n : B_n \vdash t_1 : C$ for some type C. Taking $t_0' = t_0\{\vec{t}/\vec{x}\}$ and $t_1' = t_1\{\vec{t}/\vec{x}\}$, we have $(t_0\, t_1)\{\vec{t}/\vec{x}\} = t_0'\, t_1'$. By definition, the program $\langle t_0'\, t_1', \mathrm{E}\rangle$ is the same as the program represented by $\langle t_0', \mathrm{E}\, t_1'\rangle$. Since t_0 is a subterm of $t_0\, t_1$, we can apply the induction hypothesis to deduce $\mathcal{N}(\langle t_0', \mathrm{E}\, t_1'\rangle)$ provided that $\mathrm{E}\, t_1'$ is well typed and that $\mathcal{C}_{\mathsf{cont}\,(C\to A)}(\mathrm{E}\, t_1')$ holds. The former is easy to see, and for the latter let us unfold the definition of $\mathcal{C}_{\mathsf{cont}\,(C\to A)}$. Let v be a value of type $C \to A$ and such that $\mathcal{R}_{C\to A}(v)$ holds. We need to show that $\mathcal{N}(\langle v, \mathrm{E}\, t_1'\rangle)$. Here again we can use another representative of the class of programs equal to $\langle v, \mathrm{E}\, t_1'\rangle$, such as $\langle v\, t_1', \mathrm{E}\rangle$. From the definition of $\mathcal{R}_{C\to A}(v)$, it is sufficient to show that $\mathcal{Q}_C(t_1')$. By induction hypothesis on t_1, we obtain that $\mathcal{N}(\langle t_1', \mathrm{E}'\rangle)$ for any context E' such that $\mathcal{C}_{\mathsf{cont}\,C}(\mathrm{E}')$, which proves that $\mathcal{Q}_C(t_1')$.

□

Theorem 2.7 (Termination of CBN evaluation) *If t is a closed well-typed term, then $\mathcal{N}(\langle t, \bullet\rangle)$ holds.*

Proof. Since the empty context satisfies $\mathcal{C}_{\mathsf{cont}\,A}$ for any type A, the theorem follows from Lemma 2.6. □

2.3.3 Extracted evaluator

The program we obtain by extraction from the proof of Lemma 2.6 is as follows:

$$\begin{aligned}
\mathbf{eval}^{\vec{x}}\ x_i &= \lambda \vec{t}\vec{u}\kappa.u_i\ \kappa \\
\mathbf{eval}^{\vec{x}}\ \lambda x.t &= \lambda \vec{t}\vec{u}\kappa.\kappa\ ((\lambda x.t)\{\vec{t}/\vec{x}\})\ (\lambda su\kappa.\mathbf{eval}^{\vec{x}x}\ t\ (\vec{t}s)\ (\vec{u}u)\ \kappa) \\
\mathbf{eval}^{\vec{x}}\ t_0\ t_1 &= \lambda \vec{t}\vec{u}\kappa.\mathbf{eval}^{\vec{x}}\ t_0\ \vec{t}\vec{u}\ (\lambda vu.u\ (t_1\{\vec{t}/\vec{x}\})\ (\lambda\kappa.\mathbf{eval}^{\vec{x}}\ t_1\ \vec{t}\vec{u}\ \kappa))
\end{aligned}$$

As in call by value, the evaluator is in continuation-passing style (but here, the use of continuations imposes the call-by-name evaluation order) and it threads two environments: $\vec{t}$ with unevaluated closed terms to be substituted in the final value, and $\vec{u}$ with delayed computations, i.e., thunks, waiting to be activated with a continuation (κ).

The complete evaluator for call by name, extracted from the proof of Theorem 2.7, can be written as follows:

$$\mathbf{norm}\ t = \mathbf{eval}^{\epsilon}\ t\ \epsilon\epsilon\kappa_{init}$$

where $\kappa_{init} = \lambda vu.v$.

2.4 Comparison with the standard approach

In a previous work by Biernacka et al. the authors formalized the problem of weak head normalization for the simply typed lambda calculus using standard, "direct-style" logical predicates à la Tait [9]. By extraction using modified realizability, they obtained two evaluators for the two reduction strategies. Not surprisingly, the evaluators obtained in the present work are closely related to those "direct-style" evaluators. In the call-by-name case, the evaluator we obtained here is a CPS-translated counterpart of the call-by-name evaluator from the cited work (using the standard call-by-name CPS transformation [22]). In the call-by-value case, the evaluator is also in CPS but obtained by the standard call-by-value translation from the call-by-value direct-style evaluator. Both evaluators obtained here differ slightly from literal CPS translations because here we used slightly optimized logical predicates: we defined them on values only and therefore we did not need to include the condition that they normalize in the empty context in the definition of the predicate $\mathcal{R}_A$, because it is trivially satisfied for values. If we had defined the logical predicates on terms, in each case we would have obtained an evaluator that would be exactly the CPS-translated version of the respective direct-style evaluator (but it would contain redundancies.)

3 Abortive control operators

In this section, we extend the simply typed lambda calculus with abortive control operators for first-class continuations and we prove termination of evaluation in the extended language under the call-by-value and call-by-name reduction strategies.

3.1 The call-by-value reduction strategy

3.1.1 Terms and contexts: syntax and typing

The language we consider here is the simply typed lambda calculus extended with the binder version of the operator *callcc* ($\mathcal{K}k.t$), introduced by Reynolds [23], and with a construct to apply a captured continuation ($k \hookleftarrow t$) akin to the operator *throw* known from Standard ML of New Jersey [19]. When evaluated, the expression $\mathcal{K}k.t$ captures the current continuation (in some representation, e.g., as a reduction context), binds it to k and evaluates t with the current continuation. If at some point a value is thrown to k, the then-current continuation is discarded and the continuation bound to k becomes the current continuation. Hence, abortive control operators model jumps.

In the reduction semantics for *callcc* that we consider, captured continuations will be represented syntactically by reduction contexts. Therefore, we extend the syntax with applications of a captured context to a term ($\mathrm{E} \hookleftarrow t$), an expression that may arise in the process of evaluation of programs containing *callcc*. The extended grammar of terms reads as follows:

$$(\mathit{terms}) \quad t ::= x \mid \lambda x.t \mid t\,t \mid \mathcal{K}k.t \mid k \hookleftarrow t \mid \mathrm{E} \hookleftarrow t$$

where context variables (or, continuation variables) k are drawn from a separate set than object variables x, i.e., a continuation variable can only be used in the binder $\mathcal{K}k.t$ or in a context application expression $k \hookleftarrow t$.

In accordance with the description above, the construct $\mathrm{E} \hookleftarrow t$ is never used in writing actual programs in languages with *callcc* and *throw*. Therefore, we distinguish terms that do not contain any subterm of the form $\mathrm{E} \hookleftarrow t$ and we call such terms *plain terms*.

In addition to the standard call-by-value reduction contexts, the language contains contexts of the form $\mathrm{E}'\,\mathrm{E}$ representing "the term with the hole" $\mathrm{E}[\mathrm{E}' \hookleftarrow [\,]]$, whereas functions remain the only values:

$$(\mathit{CBV\ contexts}) \quad \mathrm{E} ::= \bullet \mid v\,\mathrm{E} \mid \mathrm{E}\,t \mid \mathrm{E}' \hookleftarrow \mathrm{E}$$

$$(\mathit{values}) \qquad\qquad v ::= \lambda x.t$$

The plugging function is defined as before, with the new context handled as follows:

$$\mathit{plug}\ (t, \mathrm{E}'\ \mathrm{E}) = \mathit{plug}\ (\mathrm{E}' \hookleftarrow t, \mathrm{E})$$

As for the simply typed lambda calculus, we define programs as pairs consisting of a term and a reduction context and we equate such pairs if they represent the same plugged term. We say a term, a context or a program is closed if none of its object variables or continuation variables occur free.

Besides plain terms, we also distinguish plain contexts and plain programs. In the sequel, we will show that plain programs have the strong type soundness property (not guaranteed if we consider arbitrary terms) and we will prove termination of evaluation for plain programs.

The grammar of types of terms and contexts remains unchanged. However, in the presence of continuation variables (k) the typing judgments use an additional typing context Δ that associates continuation variables with their types. Terms are assigned types according to the following inference rules:

$$\frac{}{\Gamma, x : A; \Delta \vdash x : A} \qquad \frac{\Gamma, x : A; \Delta \vdash t : B}{\Gamma; \Delta \vdash \lambda x.t : A \to B}$$

$$\frac{\Gamma; \Delta \vdash t_0 : A \to B \quad \Gamma; \Delta \vdash t_1 : A}{\Gamma; \Delta \vdash t_0\, t_1 : B} \qquad \frac{\Gamma; \Delta, k : \mathsf{cont}\, A \vdash t : A}{\Gamma; \Delta \vdash \mathcal{K}k.t : A}$$

$$\frac{\Gamma; \Delta, k : \mathsf{cont}\, A \vdash t : A}{\Gamma; \Delta, k : \mathsf{cont}\, A \vdash k \hookleftarrow t : B} \qquad \frac{\Gamma; \Delta \vdash \mathrm{E} : \mathsf{cont}\, A \quad \Gamma; \Delta \vdash t : A}{\Gamma; \Delta \vdash \mathrm{E} \hookleftarrow t : B}$$

We can see that these rules agree with the standard typing for first-class continuations both from the semantics and logic viewpoints [1, 19, 27]. In particular, if we interpret them through the Curry-Howard correspondence, we obtain a natural deduction system for minimal classical logic, i.e., minimal logic + Peirce's law. Indeed, we have $\lambda y.\mathcal{K}k.y\,(\lambda x.k \hookleftarrow x) : ((A \to B) \to A) \to A$.

We also need to define a set of rules for typing contexts:

$$\frac{}{\Gamma; \Delta \vdash \bullet : \mathsf{cont}\, A} \qquad \frac{\Gamma; \Delta \vdash v : A \to B \quad \Gamma; \Delta \vdash \mathrm{E} : \mathsf{cont}\, B}{\Gamma; \Delta \vdash v\,\mathrm{E} : \mathsf{cont}\, A}$$

$$\frac{\Gamma; \Delta \vdash t : A \quad \Gamma; \Delta \vdash \mathrm{E} : \mathsf{cont}\, B}{\Gamma; \Delta \vdash \mathrm{E}\,t : \mathsf{cont}\,(A \to B)} \qquad \frac{\Gamma; \Delta \vdash \mathrm{E}' : \mathsf{cont}\, A \quad \Gamma; \Delta \vdash \mathrm{E} : \mathsf{cont}\, B}{\Gamma; \Delta \vdash \mathrm{E}' \hookleftarrow \mathrm{E} : \mathsf{cont}\, A}$$

Finally, the rule for typing a complete program refers to the type of the term represented by that program:

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

3.1.2 Reduction

The one-step reduction relation of our language is given by the following rules:

$$\begin{aligned} \langle (\lambda x.t)\, v, \mathrm{E}\rangle &\to_v \langle t\{v/x\}, \mathrm{E}\rangle \\ \langle \mathcal{K}k.t, \mathrm{E}\rangle &\to_v \langle t\{\mathrm{E}/k\}, \mathrm{E}\rangle \\ \langle \mathrm{E}' \hookleftarrow v, \mathrm{E}\rangle &\to_v \langle v, \mathrm{E}'\rangle \end{aligned}$$

Besides the usual β_v rule modeling function applications, we have the rule for capturing the current continuation (represented as a reduction context) and the rule for applying a previously captured context. Terms of the form $(\lambda x.t)\, v$, $\mathcal{K}k.t$ and $\mathrm{E}' \hookleftarrow v$ are *redexes*. Note, however, that the two new reductions are context sensitive, because—unlike in β-reduction—the reduction step alters not only redexes themselves, but also the surrounding context [8]. This is the reason why we need to be able to clearly state the boundary of the entire program.

Before proceeding to the proof of termination of evaluation of well-typed plain programs, let us discuss some of the typing properties of the presented type system. We base our presentation on Wright and Felleisen's work who considered type soundness of a polymorphic functional language with *callcc* and *abort* [27].

Because of the typing and reduction rules for context application, if we allow for non-plain programs, our language enjoys only weak type soundness, i.e., well-typed programs reduce to well-typed programs, but the type may not be preserved. The reason for the violation of the subject reduction property is the abortive character of the expression $\mathrm{E}' \hookleftarrow v$ in the reduction rule $\langle \mathrm{E}' \hookleftarrow v, \mathrm{E}\rangle \rightarrow_v \langle v, \mathrm{E}'\rangle$. In general, the answer types of E and E′ do not have to be the same. [5] Nevertheless, since the language satisfies the unique-decomposition property and weak type soundness (the proofs of both properties are routine), we can state the following proposition:

Proposition 3.1 (Progress) *For each program p, p either is a value or it reduces uniquely to another program p' such that if $\Gamma;\Delta \vdash p : A$, then $\Gamma;\Delta \vdash p' : B$ for some type B.*

Though it is impossible to prove a stronger type soundness property in the general case, we can obtain such a property if we consider only plain programs. As we will see, plain programs can be shown to satisfy the strong type soundness property stating that the type of a plain program and of its final value are the same, which in general is sufficiently strong and together with the termination theorem of Section 3.1.3 ensures that any well-typed plain program evaluates to a unique value of the same type. However, even in the case of plain programs, we cannot hope for a standard subject reduction property of our type system, since, in the course of computation, contexts get captured and are substituted for continuation variables, which leads to non-plain programs.

We shall prove strong type soundness for the above type system by relating it to a more restrictive one, namely an annotated type system that allows for applications of contexts of one fixed answer type. In the annotated type system the annotation on the turnstyle specifies the type of the entire program, of which the given phrase can be a part. Only contexts of that answer type are allowed to be captured and applied later on.

$$\frac{}{\Gamma,\, x : A;\Delta \vdash_B x : A} \qquad \frac{\Gamma,\, x : A;\Delta \vdash_C t : B}{\Gamma;\Delta \vdash_C \lambda x.t : A \to B}$$

$$\frac{\Gamma;\Delta \vdash_C t_0 : A \to B \quad \Gamma;\Delta \vdash_C t_1 : A}{\Gamma;\Delta \vdash_C t_0\, t_1 : B} \qquad \frac{\Gamma;\Delta,\, k : \mathsf{cont}\, A \vdash_B t : A}{\Gamma;\Delta \vdash_B \mathcal{K}k.t : A}$$

$$\frac{\Gamma;\Delta \vdash_C \mathrm{E} : \mathsf{cont}\, A \quad \Gamma;\Delta \vdash_C t : A}{\Gamma;\Delta \vdash_C \mathrm{E} \hookleftarrow t : B} \qquad \frac{\Gamma;\Delta,\, k : \mathsf{cont}\, A \vdash_C t : A}{\Gamma;\Delta,\, k : \mathsf{cont}\, A \vdash_C k \hookleftarrow t : B}$$

The contexts are typed as follows:

[5] The answer type of a context is the top-level type of the program obtained by pairing the context with any term of the correct type.

$$\frac{}{\Gamma;\Delta \vdash_A \bullet : \mathsf{cont}\, A} \qquad \frac{\Gamma;\Delta \vdash_C v : A \to B \quad \Gamma;\Delta \vdash_C \mathrm{E} : \mathsf{cont}\, B}{\Gamma;\Delta \vdash_C v\, \mathrm{E} : \mathsf{cont}\, A}$$

$$\frac{\Gamma;\Delta \vdash_C t : A \quad \Gamma;\Delta \vdash_C \mathrm{E} : \mathsf{cont}\, B}{\Gamma;\Delta \vdash_C \mathrm{E}\, t : \mathsf{cont}\, (A \to B)} \qquad \frac{\Gamma;\Delta \vdash_C \mathrm{E}' : \mathsf{cont}\, A \quad \Gamma;\Delta \vdash_C \mathrm{E} : \mathsf{cont}\, B}{\Gamma;\Delta \vdash_C \mathrm{E}' \hookleftarrow \mathrm{E} : \mathsf{cont}\, A}$$

The type annotation is introduced by the rule for typing programs:

$$\frac{\Gamma;\Delta \vdash_A \mathrm{E}[t] : A}{\Gamma;\Delta \vdash_A \langle t, \mathrm{E}\rangle}$$

Since all the contexts occurring in a program as terms must have the same answer type (given by the annotation), the subject reduction property for the annotated type systems can be proved in the standard way [27]:

Proposition 3.2 *If* $\Gamma;\Delta \vdash_A p$ *and* $p \to_v p'$, *then* $\Gamma;\Delta \vdash_A p'$.

Next, we state a few lemmas that establish the relationship between the unannotated and annotated type systems. First, proved by rule induction is the following lemma:

Lemma 3.3 (i) *If* t *is plain and* $\Gamma;\Delta \vdash t : A$, *then* $\Gamma;\Delta \vdash_C t : A$ *for any type* C.

(ii) *If* E *is plain and* $\Gamma;\Delta \vdash \mathrm{E} : \mathsf{cont}\, A$, *then* $\Gamma;\Delta \vdash_C \mathrm{E} : \mathsf{cont}\, A$ *for some type* C.

As a direct corollary from Lemma 3.3 we obtain:

Lemma 3.4 *If* p *is plain and* $\Gamma;\Delta \vdash p : A$, *then* $\Gamma;\Delta \vdash_A p$.

From Lemma 3.4 and Proposition 3.2, we can see that plain programs capture and subsequently apply contexts only of one fixed answer type.

Conversely, by rule induction, we obtain that we can erase type annotations from typing judgments for terms and contexts:

Lemma 3.5 (i) *If* $\Gamma;\Delta \vdash_C t : A$ *then* $\Gamma;\Delta \vdash t : A$.

(ii) *If* $\Gamma;\Delta \vdash_C \mathrm{E} : \mathsf{cont}\, A$, *then* $\Gamma;\Delta \vdash \mathrm{E} : \mathsf{cont}\, A$.

As a corollary, we can remove the type annotations from typing judgments for programs:

Lemma 3.6 *If* $\Gamma;\Delta \vdash_A p$, *then* $\Gamma;\Delta \vdash p : A$.

Combining Lemmas 3.4 and 3.6 and Proposition 3.2, we obtain strong type soundness for the unannotated type system [27]:

Proposition 3.7 (Preservation) *If* p *is plain,* $\Gamma;\Delta \vdash p : A$ *and* $p \to_v^* p_v$, *then* $\Gamma;\Delta \vdash p_v : A$.

3.1.3 Termination

Our goal in this section is to prove termination of call-by-value evaluation of well-typed plain programs (hence, of well-typed plain terms). The logical predicates for the language with *callcc* are exactly the same as for the simply typed lambda calculus and we state a termination theorem analogous to that of Section 2.2.3.

In the statement of the theorem we have to keep track not only of the terms that are to be substituted for free object variables, but also of the contexts to be substituted for free continuation variables.

Lemma 3.8 *Let $x_1 : B_1, \ldots, x_n : B_n; k_1 : \mathsf{cont}\, C_1, \ldots, k_m : \mathsf{cont}\, C_m \vdash t : A$ and t be a plain term. Next, let $\vec{v}$ be a sequence of closed well-typed value terms such that $\vdash v_i : B_i$ and $\mathcal{R}_{B_i}(v_i)$ for $1 \leq i \leq n$, and let $\vec{\mathrm{E}}$ be a sequence of closed well-typed contexts such that $\vdash \mathrm{E}_i : \mathsf{cont}\, C_i$ and $\mathcal{C}_{\mathsf{cont}\, C_i}(\mathrm{E}_i)$ for $1 \leq i \leq m$. Then for all closed well-typed reduction contexts E such that $\vdash \mathrm{E} : \mathsf{cont}\, A$ and $\mathcal{C}_{\mathsf{cont}\, A}(\mathrm{E})$, the program $\langle t\{\vec{v}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\}, \mathrm{E}\rangle$ normalizes, i.e., $\mathcal{N}(\langle t\{\vec{v}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\}, \mathrm{E}\rangle)$ holds.*

Proof. The proof proceeds exactly as in Section 2.2.3, by induction on the structure of terms. We will show only the two cases for the two new syntactic constructs.

Case $\mathcal{K}k.t$. Because $\mathcal{K}k.t$ is well typed, k is of type $\mathsf{cont}\, A$ and t is of type A. Taking $t' = t\{\vec{v}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\}$, we have $(\mathcal{K}k.t)\{\vec{v}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\} = \mathcal{K}k.t'$. We have to show that $\mathcal{N}(\langle \mathcal{K}k.t', \mathrm{E}\rangle)$ holds. But this program reduces in one step to program $\langle t'\{\mathrm{E}/k\}, \mathrm{E}\rangle$. In turn, this program normalizes by induction hypothesis, because t is a subterm of $\mathcal{K}k.t$ and we know by assumption that E is well typed and that $\mathcal{C}_{\mathsf{cont}\, A}(\mathrm{E})$, so we can use it for substitution in t in the induction step.

Case $k_i \hookleftarrow t$. By assumption, k_i is a continuation variable of type $\mathsf{cont}\, C_i$ and $(k_i \hookleftarrow t)\{\vec{v}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\} = \mathrm{E}_i \hookleftarrow t\{\vec{v}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\}$. Let $t' = t\{\vec{v}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\}$. We have to show that $\mathcal{N}(\langle \mathrm{E}_i \hookleftarrow t', \mathrm{E}\rangle)$ holds. But the program $\langle \mathrm{E}_i \hookleftarrow t', \mathrm{E}\rangle$ can be represented also as $\langle t', \mathrm{E}_i \hookleftarrow \mathrm{E}\rangle$. We can now apply the induction hypothesis for t provided that the context $\mathrm{E}_i \hookleftarrow \mathrm{E}$ is well typed and that $\mathcal{C}_{\mathsf{cont}\,(C_i \to A)}(\mathrm{E}_i \hookleftarrow \mathrm{E})$ holds. The former is easy to see, and for the latter we unfold the definition of $\mathcal{C}_{\mathsf{cont}\,(C_i \to A)}$. Let v be a value of type $C_i \to A$ and such that $\mathcal{R}_{C_i \to A}(v)$ holds. We need to show that $\mathcal{N}(\langle v, \mathrm{E}_i \hookleftarrow \mathrm{E}\rangle)$ holds. The program $\langle v, \mathrm{E}_i \hookleftarrow \mathrm{E}\rangle$ can be represented by $\langle v \hookleftarrow \mathrm{E}_i, \mathrm{E}\rangle$ and this program reduces in one step to program $\langle v, \mathrm{E}_i\rangle$. But we know that $\mathcal{N}(\langle v, \mathrm{E}_i\rangle)$ by the assumption that $\mathcal{C}_{\mathsf{cont}\, C_i}(\mathrm{E}_i)$ which concludes the proof in this case.

□

Theorem 3.9 (Termination of CBV evaluation) *If t is a plain, closed, well-typed term, then $\mathcal{N}(\langle t, \bullet\rangle)$ holds.*

Proof. Since the empty context satisfies $\mathcal{C}_{\mathsf{cont}\, A}$ for any type A, the theorem follows from Lemma 3.8. □

3.1.4 Extracted evaluator

The computational content of the proof of Lemma 3.8 can be written as follows:

$$
\begin{aligned}
\mathbf{eval}^{\vec{x},\vec{k}}\ x_i &= \lambda\vec{v}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.\kappa\ v_i\ u_i\\
\mathbf{eval}^{\vec{x},\vec{k}}\ \lambda x.t &= \lambda\vec{v}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.\kappa\ (\lambda x.t')\ (\lambda vu\mathrm{E}\kappa.\mathbf{eval}^{\vec{x}x,\vec{k}}\ t\ (\vec{v}v)\ (\vec{u}u)\vec{\mathrm{E}}\vec{\kappa}\ \mathrm{E}\kappa)\\
\mathbf{eval}^{\vec{x},\vec{k}}\ t_0\ t_1 &= \lambda\vec{v}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.\mathbf{eval}^{\vec{x},\vec{k}}\ t_0\ \vec{v}\vec{u}\vec{\mathrm{E}}\vec{\kappa}(\mathrm{E}\ t_1')\\
&\qquad\qquad (\lambda v_0u_0.\mathbf{eval}^{\vec{x},\vec{k}}\ t_1\ \vec{v}\vec{u}\vec{\mathrm{E}}\vec{\kappa}(v_0\ \mathrm{E})\ (\lambda v_1u_1.u_0\ v_1\ u_1\ \mathrm{E}\ \kappa))\\
\mathbf{eval}^{\vec{x},\vec{k}}\ \mathcal{K}k.t &= \lambda\vec{v}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.\mathbf{eval}^{\vec{x},\vec{k}k}\ t\ \vec{v}\vec{u}(\vec{\mathrm{E}}\mathrm{E})(\vec{\kappa}\kappa)\mathrm{E}\kappa\\
\mathbf{eval}^{\vec{x},\vec{k}}\ k_i \hookleftarrow t &= \lambda\vec{v}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.\mathbf{eval}^{\vec{x},\vec{k}}\ t\ \vec{v}\vec{u}\vec{\mathrm{E}}\vec{\kappa}(\mathrm{E}_i \hookleftarrow \mathrm{E})(\lambda vu.\kappa_i\ vu)
\end{aligned}
$$

where $\lambda x.t' = (\lambda x.t)\{\vec{v}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\}$ and $t_1' = t_1\{\vec{v}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\}$.

The extracted function **eval** is parameterized by a vector of free object variables ($\vec{x}$) and by a vector of free continuation variables ($\vec{k}$). It uses two additional environments: one for keeping track of contexts to be substituted in the final value ($\vec{\mathrm{E}}$) and one for storing continuations associated with these contexts ($\vec{\kappa}$)—these continuations are waiting to be activated by a *throw* construct.

The first three clauses of the function **eval** are similar to those of the call-by-value evaluator for the plain lambda calculus (cf. Section 2.2) except that they thread the two new context environments and the current context (E). Evaluation of $\mathcal{K}k.t$ consists in capturing the current context E and its functional representation κ in the appropriate environments and then evaluating t in the current context, using the modified environments. Whenever a captured context is thrown a value using the *throw* construct $\kappa_i \hookleftarrow t$, the right context E_i is fetched from the environment $\vec{\mathrm{E}}$ and its functional representation κ_i becomes the continuation with which evaluation of t is invoked. Syntactic representations of contexts E are only used in substitutions for free continuation variables in the final value.

The complete evaluator, extracted from the proof of Theorem 3.9, can be written as follows:

$$\mathbf{norm}\ t = \mathbf{eval}^{\epsilon,\epsilon}\ t\ \epsilon\epsilon\epsilon\epsilon \bullet \kappa_{init}$$

where $\kappa_{init} = \lambda vu.v$.

3.2 The call-by-name reduction strategy

In the call-by-name reduction strategy, reduction contexts and values coincide with those in the call-by-name language without control operators considered in Section 2.3. The types of terms and contexts as well as the typing rules for terms are identical with the call-by-value case of Section 3.1, whereas the typing rules for contexts take into account the environment Δ, but are otherwise the same as those for the standard call-by-name contexts. The reduction rules ensure that arguments to functions and continuations are not evaluated:

$$\begin{aligned}\langle(\lambda x.r)\,t, \mathrm{E}\rangle &\to_n \langle r\{t/x\}, \mathrm{E}\rangle\\ \langle \mathcal{K}k.t, \mathrm{E}\rangle &\to_n \langle t\{\mathrm{E}/k\}, \mathrm{E}\rangle\\ \langle \mathrm{E}' \hookleftarrow t, \mathrm{E}\rangle &\to_n \langle t, \mathrm{E}'\rangle\end{aligned}$$

Analogously to the call-by-value case, it can be shown that the plain language with the call-by-name reduction strategy satisfies both the weak and strong type soundness properties. Moreover, using the logical predicates defined for the simply typed call-by-name lambda calculus in Section 2.3.2, we prove termination of call-by-name evaluation for the language augmented with *callcc*.

Lemma 3.10 *Let $x_1 : B_1, \ldots, x_n : B_n; k_1 : \mathsf{cont}\, C_1, \ldots, k_m : \mathsf{cont}\, C_m \vdash t : A$ and t be a plain term. Next, let $\vec{t}$ be a sequence of closed well-typed value terms such that $\vdash v_i : B_i$ and $\mathcal{Q}_{B_i}(t_i)$ for $1 \le i \le n$, and let $\vec{\mathrm{E}}$ be a sequence of closed well-typed contexts such that $\vdash \mathrm{E}_i : \mathsf{cont}\, C_i$ and $\mathcal{C}_{\mathsf{cont}\, C_i}(\mathrm{E}_i)$ for $1 \le i \le m$. Then for all closed well-typed reduction contexts E such that $\vdash \mathrm{E} : \mathsf{cont}\, A$ and $\mathcal{C}_{\mathsf{cont}\, A}(\mathrm{E})$, the program $\langle t\{\vec{t}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\}, \mathrm{E}\rangle$ normalizes, i.e., $\mathcal{N}(\langle t\{\vec{t}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\}, \mathrm{E}\rangle)$ holds.*

The proof proceeds in the expected way, and the evaluator we extract from it is analogous of that in Section 3.1.4, except it uses the call-by-name strategy:

$$\begin{aligned}\mathbf{eval}^{\vec{x},\vec{k}}\ x_i &= \lambda\vec{t}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.u_i\ \mathrm{E}\ \kappa\\ \mathbf{eval}^{\vec{x},\vec{k}}\ \lambda x.t &= \lambda\vec{t}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.\kappa\ (\lambda x.t')\ (\lambda s u\mathrm{E}\kappa.\mathbf{eval}^{\vec{x}x,\vec{k}}\ t\ (\vec{t}s)\ (\vec{u}u)\vec{\mathrm{E}}\vec{\kappa}\ \mathrm{E}\kappa)\\ \mathbf{eval}^{\vec{x},\vec{k}}\ t_0\ t_1 &= \lambda\vec{t}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.\mathbf{eval}^{\vec{x},\vec{k}}\ t_0\ \vec{t}\vec{u}\vec{\mathrm{E}}\vec{\kappa}(\mathrm{E}\ t_1')\\ &\qquad (\lambda v u.u\ t_1'\ (\lambda\mathrm{E}\kappa.\mathbf{eval}^{\vec{x},\vec{k}}\ t_1\ \vec{t}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\ \mathrm{E}\kappa)\ \mathrm{E}\ \kappa)\\ \mathbf{eval}^{\vec{x},\vec{k}}\ \mathcal{K}k.t &= \lambda\vec{t}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.\mathbf{eval}^{\vec{x},\vec{k}k}\ t\ \vec{t}\vec{u}(\vec{\mathrm{E}}\mathrm{E})(\vec{\kappa}\kappa)\mathrm{E}\kappa\\ \mathbf{eval}^{\vec{x},\vec{k}}\ k_i \hookleftarrow t &= \lambda\vec{t}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.\mathbf{eval}^{\vec{x},\vec{k}}\ t\ \vec{t}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}_i\kappa_i\end{aligned}$$

where $\lambda x.t' = (\lambda x.t)\{\vec{v}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\}$ and $t_1' = t_1\{\vec{v}/\vec{x}\}\{\vec{\mathrm{E}}/\vec{k}\}$.

Operationally, *callcc* and *throw* are handled in the same way under call by name and call by value (cf. Section 3.1.4). The only significant difference is that under call by value a new form of context ($\mathrm{E}_i \hookleftarrow \mathrm{E}$) is created and passed when evaluating the *throw* construct—this context representation is then used only in substitutions for the appropriate continuation variable in the final value of the program.

From Lemma 3.10, we have:

Theorem 3.11 (Termination of CBN evaluation) *If t is a plain, closed, well-typed term, then $\mathcal{N}(\langle t, \bullet\rangle)$ holds.*

From the proof of Theorem 3.11, we obtain a complete call-by-name evaluator:

$$\mathbf{norm}\ t = \mathbf{eval}^{\epsilon,\epsilon}\ t\ \epsilon\epsilon\epsilon\epsilon\ \bullet\ \kappa_{init}$$

where $\kappa_{init} = \lambda v u.v$.

3.3 Other control operators

Besides the well known abortive control operator *callcc*, several others have been considered in the literature on continuations. One of them is *abort* ($\mathcal{A}$) [27], which discards the current continuation and can be defined in our setting by the following reduction (here, in call by value) and typing rules:

$$\langle \mathcal{A}\, t, \mathrm{E}\rangle \to_v \langle t, \bullet\rangle \qquad \frac{\Gamma;\Delta \vdash t : B}{\Gamma;\Delta \vdash \mathcal{A}\, t : A}$$

Another control operator widely studied in the literature is Felleisen's variant of *callcc*—the control operator $\mathcal{C}$ [15] (contrary to *callcc*, it captures and discards the current continuation), for the uniformity of the presentation accompanied here by the *throw* construct, whose dynamic and static semantics are as in Section 3.1.2. The reduction semantics of $\mathcal{C}$ and its type assignment are defined by the rules:

$$\langle \mathcal{C}k.t, \mathrm{E}\rangle \to_v \langle t\{\mathrm{E}/k\}, \bullet\rangle \qquad \frac{\Gamma;\Delta, k : \mathsf{cont}\, A \vdash t : B}{\Gamma;\Delta \vdash \mathcal{C}k.t : A}$$

It is a matter of some minor adjustments in the proofs of termination for the language with *callcc* under call by value or call by name, in order to obtain the same result for *abort* and $\mathcal{C}$. For example, in the call-by-value setting the extracted evaluator contains the following clauses defining normalization of the $\mathcal{A}$ and $\mathcal{C}$ expressions:

$$\begin{aligned}\mathbf{eval}^{\vec{x},\vec{k}}\ \mathcal{A}t &= \lambda\vec{v}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.\mathbf{eval}^{\vec{x},\vec{k}}\ t\ \vec{v}\vec{u}\vec{\mathrm{E}}\vec{\kappa} \bullet k_{init}\\ \mathbf{eval}^{\vec{x},\vec{k}}\ \mathcal{C}k.t &= \lambda\vec{v}\vec{u}\vec{\mathrm{E}}\vec{\kappa}\mathrm{E}\kappa.\mathbf{eval}^{\vec{x},\vec{k}k}\ t\ \vec{v}\vec{u}(\vec{\mathrm{E}}\mathrm{E})(\vec{\kappa}\kappa) \bullet k_{init}\end{aligned}$$

It is easy to see that the presented typing rules for $\mathcal{A}$ and $\mathcal{C}$ are too liberal to ensure type preservation by reduction (because of the completely unconstrained type B in the premises). So even though the evaluation in the simply typed language with $\mathcal{A}$ and/or $\mathcal{C}$ always terminates, the type of the program may change in the course of computation. If we wanted to ensure type preservation under the given reduction rules (which are standard), we could use a more restrictive type system that is an extension of the annotated type system of Section 3.1.2 with the rules:

$$\frac{\Gamma;\Delta \vdash_B t : B}{\Gamma;\Delta \vdash_B \mathcal{A}\, t : A} \qquad \frac{\Gamma;\Delta, k : \mathsf{cont}\, A \vdash_B t : B}{\Gamma;\Delta \vdash_B \mathcal{C}k.t : A}$$

4 Conclusion and future work

We have shown an approach to proving termination of evaluation in reduction semantics using context-based reducibility predicates à la Tait. In particular, we have presented short and direct proofs of termination of evaluation for the simply typed lambda calculus extended with control operators *callcc*, *abort* and Felleisen's $\mathcal{C}$ for

the call-by-value and the call-by-name reduction strategies. We have also presented evaluators extracted from each of the proofs. These evaluators are instances of normalization by evaluation. Moreover, they are in continuation-passing style and the continuations arise as the computational content of the reducibility predicates for reduction contexts. This latter fact shows a logical connection between continuations and contexts; the correspondence between them has previously been observed and investigated by Danvy in the setting of program transformations [11,12].

The method of proof developed in this paper relies on the formalism of reduction semantics and is therefore fitted for languages with context-sensitive notion of reduction. In such languages a single computation step takes into account the redex as well as its surrounding context; this context may be captured and discarded or otherwise changed by the reduction step. Hence, the way of representing and reducing programs proposed in this article seems to be particularly useful in the context-sensitive world. We have shown one example of such a language: the simply typed lambda calculus with abortive control operators. Another prominent example are delimited-control operators where—unlike for abortive control operators—captured contexts are delimited and can be composed [7,13,14]. For example, the delimited-control operators *shift* and *reset* admit a context-sensitive reduction semantics with two layers of reduction contexts. It is possible to adapt the method of reducibility predicates to this more general reduction semantics and the authors are currently working on this problem. Furthermore, it is interesting to investigate how the proposed approach can be adapted to other context-sensitive languages [8] as well as to proofs of strong normalization and to languages with a form of polymorphism.

Acknowledgement

We are grateful to Olivier Danvy for his willingness to proof-read the article at a short notice, and for his helpful, comprehensive comments. Thanks are also due to the anonymous reviewers of MFPS XXV for their comments.

References

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

[2] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.

[3] 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.

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

[5] 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.

[6] Małgorzata Biernacka. Formalization of the proof of weak head normalization for System T and its extracted evaluator (an instance of normalization by evaluation), 2007. Available online at `http://www.ii.uni.wroc.pl/~mabi/nbe/cbn-system-T-church`.

[7] 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).

[8] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007. Extended version available as the research report BRICS RS-06-18.

[9] 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.

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

[11] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In 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, pages 13–23, Venice, Italy, January 2004. Invited talk.

[12] 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.

[13] 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.

[14] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988. ACM Press.

[15] 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.

[16] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[17] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[18] Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

[19] 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).

[20] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 470–490. Academic Press, 1980.

[21] Michel Parigot. Proofs of strong normalisation for second order classical natural deduction. *Journal of Symbolic Logic*, 62(4):1461–1479, 1997.

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

[23] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972.

[24] Helmut Schwichtenberg. Proofs, lambda terms and control operators. In Helmut 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.

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

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

[27] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

A An implementation of the evaluators

Below we present example OCaml implementations of the evaluators from Section 3.1.4 and Section 3.2. These implementations can be seen as desired effects of a program extraction tool but they have not been obtained in that way (in fact, automatic tools usually produce much less readable code).

In the code, we use a function `lookup` that fetches an item from an environment and a function `subst_all` that performs substitutions of values for object variables and substitutions of contexts for continuation variables. (This function may produce non-plain terms with the constructor `Throw'`.) We state only signatures of these two functions and omit their implementations:

```
lookup : 'a -> ('a * 'b) list -> 'b
subst_all : term -> (ide_v * term) list -> (ide_k * ctxt) list -> term
```

A.1 Call by value

```
type term =
  | Var of ide_v
  | Lam of ide_k * term
  | App of term * term
  | Callcc of ide_k * term
  | Throw of ide_k * term
  | Throw' of ctxt * term

and ctxt =
  | Mt_c
  | App_c of ctxt * term
  | Val_c of term * ctxt
  | Ctx_c of ctxt * ctxt

type func = F of (term -> func -> ctxt -> cont -> term)
 and cont = term -> func -> term

(*
  eval : term ->
         (ide_v * term) list -> (ide_v * func) list ->
         (ide_k * ctxt) list -> (ide_k * cont) list ->
         ctxt -> cont -> term
*)

let rec eval t vs us cs ks c k =
  match t with
    | Var x ->
        k (lookup x vs) (lookup x us)
    | Lam(x, t) ->
        k (subst_all (Lam(x,t)) vs cs)
          (F (fun v u c k -> eval t ((x,v)::vs) ((x,u)::us) cs ks c k))
    | App(t0, t1) ->
        eval t0 vs us cs ks (App_c(c, subst_all t1 vs cs))
             (fun v0 u0 -> eval t1 vs us cs ks (Val_c(v0, c))
                               (fun v1 u1 -> match u0 with F f -> f v1 u1 c k))
    | Callcc(xk, t) ->
        eval t vs us ((xk,c)::cs) ((xk,k)::ks) c k
    | Throw(xk, t) ->
        eval t vs us cs ks (Ctx_c(lookup xk cs, c)) (lookup xk ks)

(*
  norm : term -> term
*)

let norm t = eval t [] [] [] [] Mt_c (fun v u -> v)
```

A.2 Call by name

```
type term =
  | Var of ide_v
  | Lam of ide_v * term
  | App of term * term
  | Callcc of ide_k * term
```

```
    | Throw of ide_k * term
    | Throw' of ctxt * term

  and ctxt =
    | Mt_c
    | App_c of ctxt * term

  type func = F of (term -> thunk -> ctxt -> cont -> term)
  and thunk = ctxt -> cont -> term
  and cont = term -> func -> term

  (*
    eval : term ->
           (ide_v * term) list -> (ide_v * thunk) list ->
           (ide_k * ctxt) list -> (ide_k * cont) list ->
           ctxt -> cont -> term
  *)

  let rec eval t ts us cs ks c k =
    match t with
      | Var x ->
          (lookup x us) c k
      | Lam(x, t) ->
          k (subst_all (Lam(x,t)) ts cs)
            (F (fun s u c k -> eval t ((x,s)::ts) ((x,u)::us) cs ks c k))
      | App(t0, t1) ->
          let t1' = subst_all t1 ts cs
          in eval t0 ts us cs ks (App_c(c, t1'))
                  (fun v u -> match u with
                                F f -> f t1' (fun c k -> eval t1 ts us cs ks c k) c k)
      | Callcc(xk, t) ->
          eval t ts us ((xk,c)::cs) ((xk,k)::ks) c k
      | Throw(xk, t) ->
          eval t ts us cs ks (lookup xk cs) (lookup xk ks)

  (*
    norm : term -> term
  *)

  let norm t =
    eval t [] [] [] [] Mt_c (fun v u -> v)
```