A context-based approach to proving termination of evaluation

Małgorzata Biernacka [1]

Institute of Computer Science
University of Wrocław
Wrocław, Poland

Dariusz Biernacki [2]

Institute of Computer Science
University of Wrocław
Wrocław, Poland

Abstract

We show 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 that proved useful in expressing various reduction strategies concisely [15,16,17] even though the notion of a (general) context as a term with a hole has been already used before [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 [11]. Not only are contexts useful for defining reduction

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

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

semantics of a language, but they have been shown to facilitate efficient proofs of type soundness, by Felleisen and Wright [23]. 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 (Section 2).

The benefits of using contexts can be seen perhaps most prominently in languages with control operators, i.e., syntactic constructs that manipulate "the rest of the computation" [15]. In evidence, 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 (Section 3).

The method of proof we apply in this work – using a context-based variant of Tait-style reducibility predicates – 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,**?**]. 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 he also indirectly 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 strong normalization [5], and 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 also to proofs of weak head normalization by Coquand and Dybjer and by Biernacka et al. [9,10]. 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.

2 The simply typed lambda calculus

In this section we present two proofs of weak head normalization of closed programs in the simply typed lambda calculus under call by value and call by name, using a variant of reducibility predicates à la Tait. 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 evaluation contexts is different for each evaluation strategy considered (obviously), but the proof itself seems to be even easier to carry out than the proof using the standard reducibility predicates. Finally, an – expected – consequence of this approach is that the computational content of the proof (i.e., the extracted program) is an evaluator in continuation-passing style. Moreover, the CPS evaluator can be otherwise obtained by CPS-translating the extracted evaluator from the standard proof (in both, call-by-value and call-by-name strategies).

2.1 Terms: syntax and typing

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

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

and the syntax of reduction contexts depends on the strategy we choose for reduction (in fact, the grammar of reduction contexts *defines* 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 differing 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:

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

and the typing relation on terms is given by the following inference system:

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t_0 : A \to B \quad \Gamma \vdash t_1 : A}{\Gamma \vdash t_0\, t_1 : B}$$

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:

$$\textit{CBV contexts}\ \mathrm{E} ::= \cdot \mid v\,\mathrm{E} \mid \mathrm{E}\,t$$

$$\textit{values}\qquad v ::= \lambda x.t$$

where values (v) form a subcategory of terms and are used to designate normal forms in the language.

Contexts are part of the syntax and not just a metarepresentation of "terms with a hole". They are represented inside-out, i.e.: $\cdot$ 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 the hole" $\mathrm{E}\,[\,[\,]\,t\,]$. The meaning of contexts we are defining is standard, but we do not use them in the same way as it is usually done. In particular, we do not identify a term in the context with the term it represents according to the informal description above. We will clarify the role of contexts shortly.

We say a reduction context is closed, if its constituant terms are all closed, i.e..:

- the empty context is closed
- if t is a closed term and E is a closed context, then $\mathrm{E}\,t$ is a closed context
- if v is a closed value and E is a closed context, then $v\,\mathrm{E}$ is a closed context.

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

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

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, contexts and values, we now define a *program* in the call-by-value language as the pair of a term and a call-by-value reduction context. Informally, such a program represents the term obtained by plugging the given term into the given context.

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

This definition differs from the usual definition of a program as arbitrary closed term in that we explicitly state the "boundary" of a program (or, top level): note that we do not have a way to compose reduction contexts, so we cannot obtain a "bigger" program by plugging one program into another reduction context. While it 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 this definition, various pairs (term, context) can represent the same program, or "plugged term", if we apply the function *plug* to the pair. 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 (term, context) defined as follows:

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

For example, the program $\langle (\lambda\, x.r)\ s, \cdot\rangle$ can be equally represented by another program $\langle \lambda\, x.r, (\cdot\ s)\rangle$ or by $\langle s, ((\lambda\, x.r)\ \cdot)\rangle$. All these representations correspond to different *decompositions* of the same term.

Since we want to consider only well-typed programs (in the sense of the plugged terms they represent), we introduce a typing relation not only on terms but also on reduction contexts. The crucial issue in typing contexts is the type of the "hole", i.e.,

the type of the term that will be plugged in the context – we only allow programs where the type of the term and the type of the context hole are the same.
Types of contexts are defined using the following syntax:

$$\textit{context types } C ::= \mathsf{cont}\, A$$

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

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

It is not difficult to see that the *plug* function 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. □

It may not be immediately clear from this presentation what the type of a program is, when we only have a pair of a term and a context. As a consequence of Lemma 2.1, all programs belonging to the same abstraction class can be assigned the same type: the type of the plugged term. Therefore we define the type of the program $\langle t, \mathrm{E}\rangle$ to be the type of the term $\mathrm{E}[t]$; the following rule for typing programs is well defined:

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

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 [19]). 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: by Lemma 2.1 each program corresponds to a well-typed lambda term and each well-typed lambda term can be represented by a well-typed program, if we pair it with the empty context.

2.2.2 Reduction

The grammar of contexts defined in the previous subsection determines that the language there defined will be given a call-by-value reduction strategy. We define a one-step reduction relation on (abstraction classes of) 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 substitution of a term 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, the relation $\to_v$ is deterministic and it is a function on abstraction classes. We state the unique decomposition

property without proof, as it is standard for this language.

Property 1 (Unique decomposition (CBV)) *For all terms t, t is either a value, or it decomposes* uniquely *into a CBV reduction context* E *and a redex 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 value; here—values are programs of the form $p_v := \langle v, \cdot \rangle$.

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 in the standard CBV reduction strategy to the term $\mathrm{E}'\,[\,t'\,]$.*

The reduction relation preserves the type of the program, 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

In general, all simply typed lambda terms normalize into normal forms. Evaluation is a weak form of normalization that does not enter inside lambda abstraction, and its normal forms are traditionally called values. In this section we show a new proof of termination for call-by-value evaluation, using logical predicates in the style of Tait but based on contexts as well as on terms rather than on terms only.

In this section, for simplicity, we only consider closed programs, although the method generalizes to all 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:

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

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

where

$$\mathcal{N}\,(p) := \exists p_v.\,p \rightarrow_v^* p_v.$$

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 entails that it normalizes (in the weak sense considered here). 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, in the call-by-value case, 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 plugged into a reducible context, it normalizes (as a program). Simultaneously, we define a reducibility predicate on well-typed reduction contexts, $\mathcal{C}_{\mathsf{cont}\,A}$, saying that a reducible context plugged with a reducible value normalizes (as a program). The typing properties ensure that the programs resulting from these pluggings are well typed, but we do not need to know their type in order to prove the normalization theorem. What is more, for the pure lambda calculus we do not really need to define $\mathcal{R}_b$, because there are no values of base type in this language; this predicate simply is not used anywhere. However, we state the full definition to show the possibility of extending the language and to show similarity with the standard definition considered in previous work.

Theorem 2.4 (Termination of CBV evaluation) *Let $x_1 : B_1, \ldots, x_n : B_n \vdash t : A$. Next, let $\vec{v_i}$ 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 $\vdash \mathrm{E} : \mathsf{cont}\,A$ and $\mathcal{C}_{\mathsf{cont}\,A}(\mathrm{E})$, the program $\langle t\{\vec{v_i}/\vec{x_i}\}, \mathrm{E}\rangle$ normalizes, i.e., $\mathcal{N}(\langle t\{\vec{v_i}/\vec{x_i}\}, \mathrm{E}\rangle)$ holds. (Notation $t\{\vec{v_i}/\vec{x_i}\}$ stands for 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 $t\{\vec{v_i}/\vec{x_i}\} = v_i$. 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 $r' = r\{\vec{v_i}/\vec{x_i}\}$, we have $(\lambda\, x.r)\{\vec{v_i}/\vec{x_i}\} = \lambda\, x.r'$. We will show that $\mathcal{R}_A(\lambda\, x.r')$ holds, and from this fact it follows that the required $\mathcal{N}(\langle(\lambda\, x.r)\{\vec{v_i}/\vec{x_i}\}, \mathrm{E}\rangle)$ 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 $\langle r\{\vec{v_i}/\vec{x_i}, v/x\}, \mathrm{E}\rangle$. By induction hypothesis, $\mathcal{N}(\langle r\{\vec{v_i}/\vec{x_i}, v/x\}, \mathrm{E}\rangle)$ 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 $t_0' = t_0\{\vec{v_i}/\vec{x_i}\}$ and $t_1' = t_1\{\vec{v_i}/\vec{x_i}\}$, we have $(t_0\, t_1)\{\vec{v_i}/\vec{x_i}\} = t_0'\, t_1'$. 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.

□

It is straightforward to see that the empty context satisfies property $\mathcal{C}_{\mathsf{cont}\,A}$ for any type A. From Theorem 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. Hence, 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 2.4 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. Several such formalizations for normalization problems have been done: partial formalizations in minimal intuitionistic logic, using Kreisel's modified realizability interpretation to extract a program (e.g., Berger formalized the proof of strong normalization for simply typed lambda terms [5,3] and obtained an NbE normalization algorithm; Biernacka et al. formalized the proof of weak head normalization for the same language with different reduction strategies and obtained a call-by-value and a call-by-name evaluator, again instances of NbE [9]). While the cited formalizations are conceptually simple, minimal logic is not expressive enough to encode the problem entirely, therefore a number of other formalizations have been done in more complex formal systems, with the support of automated tools, such as interactive proof assistants Coq or Isabelle/HOL (e.g., the problem considered by Berger and Schwichtenberg has been formalized by Letouzey et al. [4] and the weak head normalization problem has been formalized in Coq as well [6]). The Coq proof assistant comes with an extraction mechanism, yielding code in OCaml or Haskell programming languages. Of course, whether we consider partial formalization in minimal logic, or full formalization in the Coq's underlying Calculus of Constructions, the structure of the proof remains the same; consequently, the programs obtained by extraction "do the same" in both cases. What is more, the structure of the programs is similar; the differences may occur at the level of syntax of the metalanguage used to write these programs and at the level of efficiency (in most cases, programs obtained by extraction can be further optimized by hand, especially those obtained by automatic tools [20]).

In this work, our interest lies not in completely formalizing the problem–it can easily be done 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 present the "skeleton" of the program that can be extracted from the proof of Theorem 2.4. The basic idea of program extraction relies on the Curry-Howard correspondence between proofs and programs: roughly, we can view the proof of Theorem 2.4 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 to the proof of Theorem 2.4, we obtain a program that normalizes simply-typed lambda terms into values according to the call-by-value strategy. This program is in continuation-passing style and its structure is the following:

$$\begin{aligned}
\mathrm{eval}^{\vec{x}}\ x_i &= \lambda\,\vec{v}\vec{u}\kappa.\kappa\ v_i\ u_i \\
\mathrm{eval}^{\vec{x}}\ \lambda\,x.t &= \lambda\,\vec{v}\vec{u}\kappa.\kappa\ ((\lambda\,x.t)\{\vec{v}/\vec{x}\})\ (\lambda\,vu\kappa.\mathrm{eval}^{\vec{x}x}\ t\ (\vec{v}v)\ (\vec{u}u)\ \kappa) \\
\mathrm{eval}^{\vec{x}}\ t_0\ t_1 &= \lambda\,\vec{v}\vec{u}\kappa.\mathrm{eval}^{\vec{x}}\ t_0\ \vec{v}\vec{u}\ (\lambda\,vu.\mathrm{eval}^{\vec{x}}\ t_1\ \vec{v}\vec{u}\ (\lambda\,v'u'.u\ v'u'\ \kappa))
\end{aligned}$$

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 free variables in terms, and one ($\vec{u}$) containing functions (the computational content of the relation $\mathcal{R}_A$). The substitutions are only made in the final step of computation when we have to return a value as a closed term. 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. Therefore, this evaluator is an instance of normalization by evaluation – normalization (reduction) in the source language is done by evaluation at the metalevel.

Continuations (κ) in the evaluator arise as the computational content of the relation $\mathcal{C}_{\mathsf{cont}\,A}$. The syntactic 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].

The function eval is the computational content of the proof of Theorem 2.4. In order to normalize a closed term t, we apply the theorem with the empty sequence of terms and with the empty context to obtain the proof of $\mathcal{N}\ (\langle t, \cdot\rangle)$. Thus the program extracted from the proof of the fact $\mathcal{C}_{\mathsf{cont}\,A}\ (\cdot)$ is the initial continuation with which we activate the eval function. It is easy to observe that this initial continuation is the function $\lambda\,vu.v$.

The complete evaluator therefore can be written as follows:

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

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

According to the normalization-by-evaluation nomenclature, the eval function "reflects" object-level terms at the metalevel and the application to the initial continuation is the "reification" of metaobjects at the object level.

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:

$$\textit{CBN contexts}\ \mathrm{E} ::= \cdot \mid \mathrm{E}\ t$$

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 the CBN contexts is a subset of the inference rule for the 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:

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

All the above adjustments are standard. Next we need to define the logical relations needed for the proof of termination for the call-by-name case.

$$\mathcal{R}_b\ (t) := \mathcal{N}\ (\langle t, \cdot\rangle)$$

$$\mathcal{R}_{A\to B}\ (t_0) := \mathcal{N}\ (\langle t_0, \cdot\rangle) \wedge \forall t_1.\, \mathcal{Q}_A\ (t_1) \rightarrow \mathcal{Q}_B\ (t_0\ t_1)$$

$$\mathcal{Q}_A\ (t) := \forall \mathrm{E}.\, \mathcal{C}_{\mathsf{cont}\ A}\ (\mathrm{E}) \rightarrow \mathcal{N}\ (\langle t, \mathrm{E}\rangle)$$

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

$$\mathcal{N}\ (p) := \exists p_v.\, p \rightarrow_n^* p_v$$

Here, we also define two main logical predicates: $\mathcal{R}_A$ on closed terms of type A (and not on closed values as in call by value) and $\mathcal{C}_{\mathsf{cont}\ A}$ on closed contexts of type $\mathsf{cont}\ A$. Another difference is that we explicitly require terms satisfying $\mathcal{R}_A$ to normalize when put in the empty context (in the call-by-value case, this part was redundant because values are already normalized). The auxiliary predicate $\mathcal{Q}_A$ is defined on closed terms of type A and it expresses the property that a term plugged in any context satisfying $\mathcal{C}_{\mathsf{cont}\ A}$ normalizes (as a program).

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

Theorem 2.5 (Termination of CBN evaluation) *Let* $x_1 : B_1, \ldots, x_n : B_n \vdash t : A$. *Next, let* $\vec{t_i}$ *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* $\langle t\{\vec{t_i}/\vec{x_i}\}, \mathrm{E}\rangle$ *normalizes, i.e.,* $\mathcal{N}\,(\langle t\{\vec{t_i}/\vec{x_i}\}, \mathrm{E}\rangle)$ *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 $t\{\vec{v_i}/\vec{x_i}\} = v_i$. 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 $r' = r\{\vec{t_i}/\vec{x_i}\}$, we have $(\lambda\, x.r)\{\vec{t_i}/\vec{x_i}\} = \lambda\, x.r'$. 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 $\mathcal{N}\,(\langle(\lambda\, x.r)\{\vec{t_i}/\vec{x_i}\}, \mathrm{E}\rangle)$ holds. In order to prove $\mathcal{R}_A\,(\lambda\, x.r')$, we observe that $\mathcal{N}\,(\langle \lambda\, x.r', \cdot\rangle)$, so 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 $\langle r\{\vec{t_i}/\vec{x_i}, s/x\}, \mathrm{E}\rangle$. By induction hypothesis, $\mathcal{N}\,(\langle r\{\vec{t_i}/\vec{x_i}, s/x\}, \mathrm{E}\rangle)$ 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 $t_0' = t_0\{\vec{t_i}/\vec{x_i}\}$ and $t_1' = t_1\{\vec{t_i}/\vec{x_i}\}$, we have $(t_0\, t_1)\{\vec{t_i}/\vec{x_i}\} = t_0'\, t_1'$. 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 s be a term of type $C \to A$ and such that $\mathcal{R}_{C\to A}\,(s)$ holds. We need to show that $\mathcal{N}\,(\langle s, \mathrm{E}\; t_1'\rangle)$. Here again we can use another representative of the class of programs equal to $\langle s, \mathrm{E}\; t_1'\rangle$, such as $\langle s\, t_1', \mathrm{E}\rangle$. From the definition of $\mathcal{R}_{C\to A}\,(s)$, 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 $\mathcal{Q}_C\,(t_1')$.

□

2.3.3 Extracted evaluator

The program we obtain by extraction from the proof of Theorem 2.5 is as follows:

$$\begin{aligned}
\mathrm{eval}^{\vec{x}}\; x_i &= \lambda\, \vec{t}\vec{u}\kappa.u_i\; \kappa \\
\mathrm{eval}^{\vec{x}}\; \lambda\, x.t &= \lambda\, \vec{t}\vec{u}\kappa.\kappa\; \langle((\lambda\, x.t)\{\vec{t}/\vec{x}\}), (\lambda\, su\kappa.\mathrm{eval}^{\vec{x}x}\; t\; (\vec{t}s)\; (\vec{u}u)\; \kappa)\rangle \\
\mathrm{eval}^{\vec{x}}\; t_0\; t_1 &= \lambda\, \vec{t}\vec{u}\kappa.\mathrm{eval}^{\vec{x}}\; t_0\; \vec{t}\vec{u}\; (\lambda\, u.(\mathrm{snd}\; u)\; (t_1\{\vec{t}/\vec{x}\})\; (\lambda\, \kappa.\mathrm{eval}^{\vec{x}}\; t_1\; \vec{t}\vec{u}\; \kappa))
\end{aligned}$$

As in call by value, the evaluator is in continuation-passing style and it threads two environments: $\vec{t}$ with unevaluated closed terms to be substituted in the final value, and $\vec{u}$ with delayed computations waiting to be activated with a continuation (κ). Unlike the call-by-value case, the initial continuation depends on the type of its argument: if it is of base type, then the initial continuation simply returns its argument (although it can never happen here since we do not have values of base type); if it is of an arrow type, then the initial continuation returns the first component of the pair it gets as argument.

The complete evaluator for call by name can be written as follows:

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

where $\kappa_{init} = \lambda v.v$ if $\vdash\ t\ :\ b$ and $\kappa_{init} = \lambda v.\text{fst}\ v$ if $\vdash\ t\ :\ A \to B$ for some types A, B.

2.4 Comparison with the standard approach

In a previous work by Biernacka et al. the authors have formalized the problem of weak head normalization for the simply typed lambda calculus using "standard" logical predicates á la Tait [9]. By extraction using modified realizability, they have 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 exactly the CPS-translated call-by-name evaluator from the cited work. In the call-by-value case, the evaluator is also in CPS but it is not directly a CPS-translated version of the "direct-style" CBV evaluator, 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 (as in the call-by-name case) in the definition of the predicate, because it is trivially satisfied for values. If we had defined the logical predicates on terms, we would have obtained an evaluator that would be exactly the CPS-translated version of the direct-style CBV 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 is the simply typed lambda calculus extended with the binder version of the operator *callcc* ($\mathcal{K}k.t$), introduced by Reynolds [22], and by a construct to apply a captured continuation ($k \hookleftarrow t$) akin to the operator *throw* known from the Standard ML of New Jersey [13]. In order to account for the reduction semantics of *callcc*, we also include in the syntax 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 therefore reads as follows:

$$\textit{terms}\ t ::= x\ \mid\ \lambda x.t\ \mid\ t\ t\ \mid\ \mathcal{K}k.t\ \mid\ k \hookleftarrow t\ \mid\ \mathrm{E} \hookleftarrow t$$

The context variables (or, continuation variables) k are drawn from a separate set than the 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 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:

$$\textit{CBV contexts}\ \ \mathrm{E} ::= \cdot \mid v\,\mathrm{E} \mid \mathrm{E}\,t \mid \mathrm{E}'\,\mathrm{E}$$

$$\textit{values}\qquad\qquad v ::= \lambda x.t$$

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

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

The grammar of types of terms and contexts remains unchanged. However, in the presence of continuation variables (k) the typing judgements 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 [13,23]. In particular, if we interpret the type $\mathsf{cont}\,A$ as $\neg A$, the rule for *callcc* gives rise to the weak Peirce's law through the Curry-Howard correspondence [1].

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

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

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

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 it does not contain free neither object variables nor continuation variables.

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

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

$$\langle \mathcal{K}k.t, \mathrm{E}\rangle \to_v \langle t\{\mathrm{E}/k\}, \mathrm{E}\rangle$$

$$\langle \mathrm{E}' \hookleftarrow v, \mathrm{E}\rangle \to_v \langle v, \mathrm{E}'\rangle$$

Besides the usual β_v rule modelling 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.

Although our main goal here is to prove termination of evaluation of well-typed programs, for completeness we 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* [23].

Because of the typing and reduction rules for context application, in general 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. 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 pure programs. We say that a program p is pure, if it contains no subterms of the form $\mathrm{E} \hookleftarrow t$. Such programs capture and subsequently apply contexts only of one unique answer type. However, in the course of computation, contexts get captured and are substituted for continuation variables, which leads to impure programs, so we cannot hope for a standard subject reduction property, but rather we should aim at a property stating that the types of a pure program and of its final value are the same. Similarly, we define pure contexts as contexts not containing terms of the form $\mathrm{E} \hookleftarrow t$.

Let us start with defining an annotated type system, where the annotation on the turnstyle specifies the type of the entire program.

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

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

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

The contexts are typed as follows:

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

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

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 [23]:

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 pure and* $\Gamma;\Delta \vdash t : A$*, then* $\Gamma;\Delta \vdash_C t : A$ *for any type* C*.*

(ii) *If* E *is pure 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 pure and* $\Gamma;\Delta \vdash p : A$*, then* $\Gamma;\Delta \vdash_A p$*.*

Conversely, we can erase type annotations from typing judgements 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 judgements 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 [23]:

Proposition 3.7 (Preservation) *If* p *is pure,* $\Gamma;\Delta \vdash p : A$ *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 pure terms. The logical predicates for the language with *callcc* are exactly the same as for the simply typed lambda calculus and then we can state the 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 contexts to be substituted for free continuation variables.

Theorem 3.8 (Termination of CBV evaluation) *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 pure term. Next, let* $\vec{v_i}$ *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* $\vec{\mathrm{E}_i}$ *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* $\langle t\{\vec{v_i}/\vec{x_i}\}\{\vec{\mathrm{E}_i}/\vec{k_i}\}, \mathrm{E}\rangle$ *normalizes, i.e.,* $\mathcal{N}(\langle t\{\vec{v_i}/\vec{x_i}\}\{\vec{\mathrm{E}_i}/\vec{k_i}\}, \mathrm{E}\rangle)$ *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 $t' = t\{\vec{v_i}/\vec{x_i}\}\{\vec{\mathrm{E}_i}/\vec{k_i}\}$, we have $(\mathcal{K}k.t)\{\vec{v_i}/\vec{x_i}\}\{\vec{\mathrm{E}_i}/\vec{k_i}\} = \mathcal{K}k.t'$. 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 of type $\mathsf{cont}\, C_i$ and $(k_i \hookleftarrow t)\{\vec{v_i}/\vec{x_i}\}\{\vec{\mathrm{E}_i}/\vec{k_i}\} = \mathrm{E}_i \hookleftarrow t\{\vec{v_i}/\vec{x_i}\}\{\vec{\mathrm{E}_i}/\vec{k_i}\}$. Let $t' = t\{\vec{v_i}/\vec{x_i}\}\{\vec{\mathrm{E}_i}/\vec{k_i}\}$. 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 $\mathcal{C}_{\mathsf{cont}\, C_i}(\mathrm{E}')$ which concludes the proof in this case.

□

3.1.4 Extracted evaluator

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

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

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 and by a vector of free continuation variables. It uses two additional environments: one for keeping track of contexts to be substituted in the final value ($\vec{\mathrm{E}_i}$) and one

for storing continuations associated with these contexts – these continuations are waiting to be activated by a *throw* construct.

The complete evaluator can be written as follows:

$$\text{norm}\ t = \text{eval}^{\epsilon,\epsilon}\ t\ \epsilon\epsilon\epsilon\epsilon \cdot \kappa_{init}$$

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

3.2 The call-by-name reduction strategy

In the call-by-name reduction strategy, the 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:

$$\langle (\lambda\, x.r)\ t, \mathrm{E}\rangle \rightarrow_n \langle r\{t/x\}, \mathrm{E}\rangle$$
$$\langle \mathcal{K}k.t, \mathrm{E}\rangle \rightarrow_n \langle t\{\mathrm{E}/k\}, \mathrm{E}\rangle$$
$$\langle \mathrm{E}' \hookleftarrow t, \mathrm{E}\rangle \rightarrow_n \langle t, \mathrm{E}'\rangle$$

Analogously to the call-by-value case, it can be shown that the pure 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*.

Theorem 3.9 (Termination of CBN evaluation) *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 pure term. Next, let $\vec{t_i}$ be a sequence of closed well-typed value terms such that $\vdash v_i : B_i$ and $\mathcal{Q}_{B_i}(t_i)$ for $1 \leq i \leq n$, and let $\vec{\mathrm{E}_i}$ 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 $\langle t\{\vec{t_i}/\vec{x_i}\}\{\vec{\mathrm{E}_i}/\vec{k_i}\}, \mathrm{E}\rangle$ normalizes, i.e., $\mathcal{N}(\langle t\{\vec{t_i}/\vec{x_i}\}\{\vec{\mathrm{E}_i}/\vec{k_i}\}, \mathrm{E}\rangle)$ 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:

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

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

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* ($\mathcal{A}$) [23], which can be defined in our setting by the following reduction and typing rules:

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

Another control operator widely studied in the literature is Felleisen's generalization of *callcc* – the control operator $\mathcal{C}$ [15], for the uniformity of the presentation accompanied here by the *throw* construct (whose dynamic and static semantics are as in the case of *callcc*). The reduction semantics of $\mathcal{C}$ and its type assignment are defined by the rules:

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

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:

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

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). 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 systems 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 the 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 evaluation contexts.

There seems to be at least two possible directions for future work concerning the proof method developed in this paper. First, it should be possible to extend our results to delimited-control operators [7,12,14]. Second, it would be interesting

to extend our proof method to languages with polymorphic type assignment [23].

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 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.

[13] Bruce F. Duba, Robert Harper, and David B. MacQueen. Typing first-class continuations in ML. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Orlando, Florida, January 1991. 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] 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.

[20] Pierre Letouzey. A new extraction for coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, International Workshop TYPES'02*, number 2646 in Lecture Notes in Computer Science, Berg en Dal, The Netherlands, April 2002. Springer-Verlag.

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

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

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