On the Overhead of CPS *

Olivier Danvy and Belmina Dzafic
BRICS †
Department of Computer Science
Aarhus University ‡
{danvy,belmina}@brics.dk

Frank Pfenning
School of Computer Science
Carnegie Mellon University §
fp@cs.cmu.edu

November 18, 1996

Abstract

Continuation-passing style (CPS) is often criticized to be more expensive than the usual direct style of functional programming. By structure, CPS functions indeed are passed one extra argument (the continuation), and each intermediate result indeed occurs in a function call (to the continuation). As higher-order functions, continuations are also more expensive.

However, by structure also, CPS exhibits a great deal of syntactic regularity. We show how to exploit this regularity to implement CPS with two stacks — one for continuation parameters and one for the parameters of continuations — in a way that reduces the extra price of CPS to managing those two stacks. In effect, the stack for continuation parameters acts as a control stack for calls and returns, and the stack for parameters of continuations acts as a data stack for intermediate results.

This demonstrates that CPS is just about as expensive as direct style, where calls, returns, and intermediate results also have to be dealt with.

To this end, we present four abstract machines for CPS λ-terms — a bare one, one with a control stack, one with a data stack, and one with both a control stack and a data stack — and we prove their equivalence.

Our result also applies to A-normal forms, i.e., monadic style.

Keywords: direct style, continuation-passing style, λ-calculus, abstract machine, logical framework.

*Draft Summary

†Basic Research in Computer Science,
Centre of the Danish National Research Foundation.
‡Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark.
Phone: (+45) 89 42 33 69. Fax: (+45) 89 42 32 55.
§5000 Forbes Ave., Pittsburgh, PA 15213-3891, USA.
This work is supported by NSF Grant CCR-9303383.

1 Introduction

Continuation-passing style (CPS), as a functional encoding of control, imposes a run-time overhead: each function takes an additional argument (the continuation), each return is metamorphosed into a call (to the continuation), and, while each expression is evaluated iteratively and independently of the evaluation order, it is at the expense of creating new continuations. Field and Harrison put it eloquently:

> "But the price to pay for so easily deriving an iterative implementation is the continuations required, which are themselves higher-order functions. As we have seen, such objects are much less efficient to implement than data, and the continuations passed as parameters in an application of the transformed function become increasingly complex, rendering the transformation worthless." [11, Page 498]

CPS thus is expensive, as illustrated in Figure 1, where we display an abstract machine reducing CPS terms. The extra expense is revealed by two supplementary substitutions, compared to an abstract machine reducing ordinary, direct-style λ-terms:

- applying a function requires us to substitute its continuation;
- applying a continuation requires us to substitute its argument.

However, and this is our point here, CPS programs obey a certain structure.

Example 1: If a source program uses no control operators such as call/cc, in its CPS counterpart, continuation parameters are not only linear but they denote a single-threaded resource, which (as is well known) obeys a stack discipline. We can thus reduce CPS programs with a specialized abstract machine (see Figure 2) where continuation parameters are implemented with a "control" stack rather than with the overly general mechanism of substitution.

Example 2: If a CPS program encodes a particular evaluation order (call-by-value, call-by-name, etc.), the parameters of continuations are not only linear but they obey a stack discipline. We can thus reduce CPS programs with a specialized abstract machine (see Figure 5) where parameters of continuations are implemented with a "data" stack rather than with the overly general mechanism of substitution.

These two specialized implementations of substitutions — of continuation parameters and of parameters of continuations — can be combined into an abstract machine (see Figure 6) with two stacks. This abstract machine only uses substitution to implement β-reduction, just like abstract machines evaluating direct-style λ-terms.[1] Since calls, returns, and intermediate results also have to be dealt with in direct style, the reputation of CPS being inherently more expensive than direct style is thus unfounded.

Our result directly applies to A-normal forms [12, Figure 9], monadic normal forms [15], nqCPS, etc., a monadic style that originates in Moggi's work [20] and has been transposed in the functional-programming world by Wadler [27]. Namely, let-bound variables in a monadic program encoding a particular evaluation order obey a stack discipline. Programs in monadic style can thus be reduced with a specialized abstract machine where let-bound variables are implemented with a stack rather than with the overly general mechanism of substitution.

[1] Further optimizations, such as the construction of closures, are orthogonal. They apply equally to our CPS machines and to standard machines and we do not detail them here.

The rest of this article is organized as follows. Section 2 presents the grammars of direct-style and of CPS terms, the CPS transformation, and the standard, stackless abstract machine for CPS terms. Section 3 presents an equivalent abstract machine for CPS terms where continuation parameters are implemented with a stack. Section 4 presents an equivalent abstract machine for CPS terms where parameters of continuations are implemented with a stack. Section 5 presents an equivalent compound abstract machine for CPS terms with two stacks. Throughout, we consider left-to-right call by value. In Section 6, however, we consider right-to-left call by value and call by name. Monadic style is addressed in Section 7. Section 8 reviews related work. Section 9 concludes.

We outline two of our equivalence proofs in appendix.

For the sake of conciseness, we only consider the pure λ-calculus here, but our results should hold for a functional programming language with tuples, conditional expressions, block structure, recursive definitions, side effects, etc., given its associated CPS transformation. We have not (yet) extended our proofs to such a larger language, but we did prototype the corresponding abstract machine and experimented with it.

2 CPS

The BNF of the pure λ-calculus reads as follows. We refer to it as *direct style* (DS) to distinguish it from the *continuation-passing style* (CPS) calculus introduced just below.

$r \in \mathrm{DRoot}$	— DS terms	$r ::= e$
$e \in \mathrm{DExp}$	— DS expressions	$e ::= e_0\, e_1 \mid t$
$t \in \mathrm{DTriv}$	— DS trivial expressions	$t ::= x \mid \lambda x.r$
$x \in \mathrm{Ide}$	— identifiers	

Direct-style terms are transformed into continuation-passing style by CPS transformation [1, 7, 16, 17, 24, 25]. (We consider left-to-right call by value, and Plotkin-style CPS, i.e., with continuations last. Fischer-style CPS follows *mutatis mutandis*, and other evaluation orders are addressed in Section 6.)

The BNF of CPS terms reads as follows. (NB: We distinguish between the original identifiers x coming from the direct-style term, and the fresh identifiers v and k introduced by the CPS transformation.)

$r \in \mathrm{CRoot}$	— CPS terms	$r ::= \lambda k.e$
$e \in \mathrm{CExp}$	— CPS (serious) expressions	$e ::= t_0\, t_1\, \lambda v.e \mid k\, t \mid (\lambda v.e)\, t$
$t \in \mathrm{CTriv}$	— CPS trivial expressions	$t ::= x \mid \lambda x.r \mid v$
$x \in \mathrm{Ide}$	— source identifiers	
$k \in \mathrm{Cont}$	— fresh continuation parameters	
$v \in \mathrm{Var}$	— fresh parameters of continuations	

NB: CPS transformers usually do not produce expressions of the form $(\lambda v.e)\, t$, but such expressions occur as intermediate forms during evaluation. We thus include them here.

CPS terms are remarkable in that they satisfy the three properties of indifference, simulation, and translation [22]. Indifference: CPS terms are evaluation-order independent. Simulation: the CPS transformation encodes an evaluation order. Translation: an equational correspondence exists between DS and CPS calculi.

Figure 1 displays the bare CPS abstract machine, which follows the tradition of Landin's SECD machine [18] and operational semantics [13, 22, 23]. We have obtained it by specializing a call-by-value abstract machine for direct-style terms to CPS terms (call-by-value because of the indifference property, which states that call-by-name and call-by-value evaluation coincide on terms in the image of the CPS transformation).

$$\frac{\vdash^{\mathrm{CExp}}_{\mathrm{bare}} e[k_{init}/k] \hookrightarrow a}{\vdash^{\mathrm{CRoot}}_{\mathrm{bare}} \lambda k.e \hookrightarrow a}$$

$$\frac{}{\vdash^{\mathrm{CExp}}_{\mathrm{bare}} k_{init}\, t \hookrightarrow t} \qquad \frac{\vdash^{\mathrm{CExp}}_{\mathrm{bare}} e[t/v] \hookrightarrow a}{\vdash^{\mathrm{CExp}}_{\mathrm{bare}} (\lambda v.e)\, t \hookrightarrow a}$$

$$\frac{\vdash^{\mathrm{CExp}}_{\mathrm{bare}} e[t_1/x][\lambda v.e'/k] \hookrightarrow a}{\vdash^{\mathrm{CExp}}_{\mathrm{bare}} (\lambda x.\lambda k.e)\, t_1\, (\lambda v.e') \hookrightarrow a}$$

Figure 1: Bare CPS abstract machine

$$\frac{k_{init} \vdash^{\mathrm{CExp}}_{\mathrm{cstack}} e \hookrightarrow a}{\vdash^{\mathrm{CRoot}}_{\mathrm{cstack}} \lambda k.e \hookrightarrow a}$$

$$\frac{}{k_{init} \vdash^{\mathrm{CExp}}_{\mathrm{cstack}} k\, t \hookrightarrow t} \qquad \frac{\kappa \vdash^{\mathrm{CExp}}_{\mathrm{cstack}} e[t/v] \hookrightarrow a}{\kappa,\, \lambda v.e \vdash^{\mathrm{CExp}}_{\mathrm{cstack}} k\, t \hookrightarrow a}$$

$$\frac{\kappa,\, \lambda v.e' \vdash^{\mathrm{CExp}}_{\mathrm{cstack}} e[t_1/x] \hookrightarrow a}{\kappa \vdash^{\mathrm{CExp}}_{\mathrm{cstack}} (\lambda x.\lambda k.e)\, t_1\, (\lambda v.e') \hookrightarrow a}$$

Figure 2: CPS abstract machine with a global control stack for the continuation parameters

3 One control stack for the continuation parameters

As is well known, if the direct-style world contains no control operators, continuation parameters denote a single-threaded resource, which furthermore obeys a stack discipline. Figure 2 displays a CPS abstract machine with one stack for continuation parameters. Here we use

$\kappa \in \mathrm{CStack}$ — Control stacks $\qquad\qquad \kappa ::= \; k_{init} \mid \kappa,\, \lambda v.e$

We prove its equivalence with the stackless abstract machine of Figure 1 by showing that the proof trees for each abstract machine are in bijective correspondence. This proof takes advantage of the characterization of occurrences of continuation parameters given in Figure 3, which we have proven elsewhere [8]. Since it is simpler than the two proofs given in the appendix, we omit it here.

$$\frac{k \models^{\mathrm{CExp}}_{\mathrm{Cont}} e}{\models^{\mathrm{CRoot}}_{\mathrm{Cont}} \lambda k.e}$$

$$\frac{\models^{\mathrm{CTriv}}_{\mathrm{Cont}} t_1 \qquad \models^{\mathrm{CTriv}}_{\mathrm{Cont}} t_0 \qquad k \models^{\mathrm{CExp}}_{\mathrm{Cont}} e}{k \models^{\mathrm{CExp}}_{\mathrm{Cont}} t_0\, t_1\, \lambda v.e} \qquad \frac{\models^{\mathrm{CTriv}}_{\mathrm{Cont}} t}{k \models^{\mathrm{CExp}}_{\mathrm{Cont}} k\, t} \qquad \frac{k \models^{\mathrm{CExp}}_{\mathrm{Cont}} e \qquad \models^{\mathrm{CTriv}}_{\mathrm{Cont}} t}{k \models^{\mathrm{CExp}}_{\mathrm{Cont}} (\lambda v.e)\, t}$$

$$\frac{}{\models^{\mathrm{CTriv}}_{\mathrm{Cont}} x} \qquad \frac{\models^{\mathrm{CRoot}}_{\mathrm{Cont}} r}{\models^{\mathrm{CTriv}}_{\mathrm{Cont}} \lambda x.r} \qquad \frac{}{\models^{\mathrm{CTriv}}_{\mathrm{Cont}} v}$$

Figure 3: Occurrence conditions over the continuation parameters in a CPS term

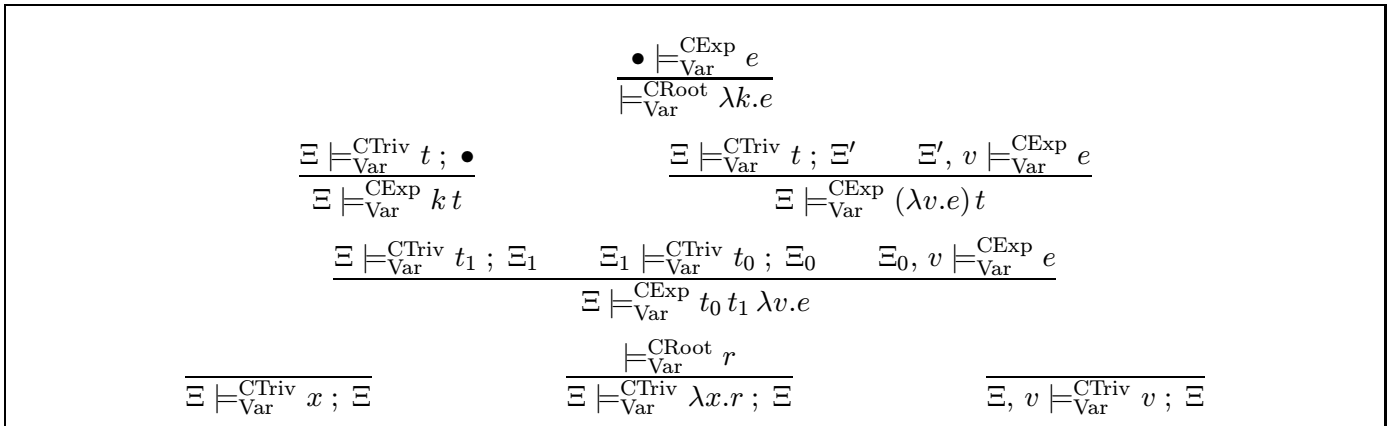

Figure 4: Occurrence conditions over the parameters of continuations in a CPS term

4 One data stack for the parameters of continuations

An earlier work on the direct-style transformation [6] made it necessary to characterize the occurrences of the parameters of continuations in CPS terms (see Figure 4). The point is that two terms such as

$$r_1 \;=\; \lambda k.f\,x\,\lambda v_1.g\,y\,\lambda v_2.v_1\,v_2\,\lambda v_3.k\,v_3$$

and

$$r_2 \;=\; \lambda k.g\,y\,\lambda v_2.f\,x\,\lambda v_1.v_1\,v_2\,\lambda v_3.k\,v_3$$

should not be mapped to the same DS term. The first is a legal output for the left-to-right call-by-value CPS transformation (i.e., $\models^{\mathrm{CRoot}}_{\mathrm{Var}} r_1$ is satisfied), but the second is not (i.e., $\models^{\mathrm{CRoot}}_{\mathrm{Var}} r_2$ is not satisfied).

It turned out to be quite challenging to prove that the CPS transformation yields terms that satisfy the occurrence conditions over the parameters of continuations. We did meet this challenge after formalizing both the CPS transformation and the occurrence conditions in Elf, and we report this case of machine-assisted proof discovery elsewhere [8]. Using the same proof technique, the second author has shown that the occurrence conditions are closed under beta and eta reduction [10].

This previous work makes it clear that for each CPS λ-abstraction, intermediate results are produced and consumed in a stack-like fashion. One could thus implement CPS programs by allocating one stack per closure, to store its intermediate results, and deallocating this stack before returning or performing a tail-call. While this strategy might well be interesting on its own right [26], the new insight which is at the core of the present work is that *these stacks can be implemented using one global stack of intermediate results.*

To this end we introduce *parameter stacks* Ξ. During the execution of the abstract machine we will have corresponding *value stacks* ξ.

$\Xi \in \mathrm{PStack}$	— Parameter stacks	$\Xi ::= \bullet \mid \Xi, v$
$\xi \in \mathrm{VStack}$	— Data stacks	$\xi ::= \bullet \mid \xi, t$

By convention, no variable v may occur more than once in a stack Ξ. This can always be achieved by renaming bound variables before parameters are added to the stack.

Figure 5 displays a CPS abstract machine with a global stack for the parameters of continuations. We prove its equivalence with the stackless abstract machine of Figure 1 by showing that the proof trees for each abstract machine are in bijective correspondence (see Theorem 2 in the appendix).

$$\frac{\bullet \vdash^{\mathrm{CExp}}_{\mathrm{vstack}} e[k_{init}/k] \hookrightarrow a}{\vdash^{\mathrm{CRoot}}_{\mathrm{vstack}} \lambda k.e \hookrightarrow a}$$

$$\frac{\xi \vdash^{\mathrm{CTriv}}_{\mathrm{vstack}} t \hookrightarrow t' ; \bullet}{\xi \vdash^{\mathrm{CExp}}_{\mathrm{vstack}} k_{init}\, t \hookrightarrow t'} \qquad \frac{\xi \vdash^{\mathrm{CTriv}}_{\mathrm{vstack}} t \hookrightarrow t' ; \xi' \qquad \xi',\, t' \vdash^{\mathrm{CExp}}_{\mathrm{vstack}} e \hookrightarrow a}{\xi \vdash^{\mathrm{CExp}}_{\mathrm{vstack}} (\lambda v.e)\, t \hookrightarrow a}$$

$$\frac{\xi \vdash^{\mathrm{CTriv}}_{\mathrm{vstack}} t_1 \hookrightarrow t_1' ; \xi_1 \qquad \xi_1 \vdash^{\mathrm{CTriv}}_{\mathrm{vstack}} t_0 \hookrightarrow \lambda x.\lambda k.e ; \xi_0 \qquad \xi_0 \vdash^{\mathrm{CExp}}_{\mathrm{vstack}} e[t_1'/x][\lambda v.e'/k] \hookrightarrow a}{\xi \vdash^{\mathrm{CExp}}_{\mathrm{vstack}} t_0\, t_1\, (\lambda v.e') \hookrightarrow a}$$

$$\frac{}{\xi \vdash^{\mathrm{CTriv}}_{\mathrm{vstack}} \lambda x.\lambda k.e \hookrightarrow \lambda x.\lambda k.e ; \xi} \qquad \frac{}{\xi,\, t \vdash^{\mathrm{CTriv}}_{\mathrm{vstack}} v \hookrightarrow t ; \xi}$$

Figure 5: CPS abstract machine with a global data stack for the parameters of continuations

$$\frac{k_{init} ; \bullet \vdash^{\mathrm{CExp}}_{\mathrm{cvstack}} e \hookrightarrow a}{\vdash^{\mathrm{CRoot}}_{\mathrm{cvstack}} \lambda k.e \hookrightarrow a}$$

$$\frac{\xi \vdash^{\mathrm{CTriv}}_{\mathrm{cvstack}} t \hookrightarrow t' ; \bullet}{k_{init} ; \xi \vdash^{\mathrm{CExp}}_{\mathrm{cvstack}} k\, t \hookrightarrow t'} \qquad \frac{\xi \vdash^{\mathrm{CTriv}}_{\mathrm{cvstack}} t \hookrightarrow t' ; \xi' \qquad \kappa ; \xi',\, t' \vdash^{\mathrm{CExp}}_{\mathrm{cvstack}} e \hookrightarrow a}{\kappa,\, \lambda v.e ; \xi \vdash^{\mathrm{CExp}}_{\mathrm{cvstack}} k\, t \hookrightarrow a}$$

$$\frac{\xi \vdash^{\mathrm{CTriv}}_{\mathrm{cvstack}} t_1 \hookrightarrow t_1' ; \xi_1 \qquad \xi_1 \vdash^{\mathrm{CTriv}}_{\mathrm{cvstack}} t_0 \hookrightarrow \lambda x.\lambda k.e ; \xi_0 \qquad \kappa,\, \lambda v.e' ; \xi_0 \vdash^{\mathrm{CExp}}_{\mathrm{cvstack}} e[t_1'/x] \hookrightarrow a}{\kappa ; \xi \vdash^{\mathrm{CExp}}_{\mathrm{cvstack}} t_0\, t_1\, (\lambda v.e') \hookrightarrow a}$$

$$\frac{}{\xi \vdash^{\mathrm{CTriv}}_{\mathrm{cvstack}} \lambda x.\lambda k.e \hookrightarrow \lambda x.\lambda k.e ; \xi} \qquad \frac{}{\xi,\, t \vdash^{\mathrm{CTriv}}_{\mathrm{cvstack}} v \hookrightarrow t ; \xi}$$

Figure 6: CPS abstract machine with both a control stack and a data stack

5 Two stacks

The control stack (Section 3) and the data stack (Section 4) do not interfere. It is thus simple to specify a compound abstract machine for CPS terms with two stacks: one for the continuation parameters, and one for the parameters of continuations. Figure 6 displays this two-stack machine. In the appendix, Theorem 4 proves its equivalence with the data-stack machine.

6 Other evaluation orders

6.1 Right-to-left call by value

We still consider call by value. Right-to-left evaluation order gives rise to similar occurrence conditions. These conditions can be similarly exploited to specify stack machines for CPS programs.

6.2 Call by name

In the output of the call-by-name CPS transformation, the CBV occurrence conditions are trivially satisfied. Therefore no occurrence conditions are needed to specify stack machines for CPS programs.[2]

7 Monadic style

A-normal forms essentially amount to CPS without continuations [12]. Their BNF reads as follows.

$r \in \mathrm{MRoot}$	— monadic-style terms	$r ::= e$
$e \in \mathrm{MExp}$	— monadic-style (serious) expressions	$e ::= \mathrm{let}\, v = t_0\, t_1\, \mathrm{in}\, e \mid \mathrm{return}(t)$
$t \in \mathrm{MTriv}$	— monadic-style trivial expressions	$t ::= x \mid \lambda x.r \mid v$
$x \in \mathrm{Ide}$	— source identifiers	
$v \in \mathrm{Var}$	— fresh let parameters	

This BNF is in bijective correspondence with the BNF of CPS [6, 12, 15]. It is thus simple to state occurrence conditions over let parameters, given a monadic-style transformation [12, Figure 9], and to write the corresponding stack-based abstract machine, similar to the one of Figure 5.

Note that we have been careful to write "monadic *style*". Our analogy is purely syntactic, and based on the bijective correspondence between CPS and A-normal forms. We specifically do *not* consider the type and the categorical implications of monadic style here. We merely want to point out that the monadic-style transformation gives rise to an occurrence property over let parameters that can be exploited to specify a stack-based abstract machine for A-normal forms. Doubtlessly it would be interesting to recast this work in a categorical setting, but that is not our point here.

8 Related work

Since Milne and Strachey [19], virtually everybody uses a control stack for continuation parameters. Exceptions include Appel, who uses the heap as his unique memory resource [2].

CPS-compiler writers have not been without noticing that the free variables of the continuation could naturally be implemented with the target-machine registers [1, 4, 17, 28]. We are not aware, however, of any implementation of parameters of continuations with an independent data stack.

[2] John Hatcliff made the same observation for the call-by-name direct-style transformation (personal communication to the first author, fall 1996).

We are not aware either of dedicated abstract machines for CPS terms, and certainly of none that implements continuation-passing with two stacks. (We are currently investigating how our new machine compare with traditional abstract machines for direct style.)

Tofte and Talpin have suggested to implement the λ-calculus with a stack of regions and no garbage collector [26]. Their basic idea is to associate a region for each lexical block, and to garbage-collect it on block exit. This scheme is of course very much allergic to CPS (which "never returns"). Our work shows that all is not lost for CPS programs when it comes to stackability.

Let us conclude with a word on proper tail-recursion. In some situations, it is essential to process tail-calls properly, e.g., in the implementation of a programming language such as Scheme [5]. As analyzed elsewhere [6, Section 3], it is in the CPS transformation itself that tail-calls need to be treated specially. For example, a direct-style term such as $\lambda f.f\,x$ should be CPS-transformed into

$$\lambda k.k\,(\lambda f.\lambda k.f\,x\,\boxed{k})$$

and not into

$$\lambda k.k\,(\lambda f.\lambda k.f\,x\,\boxed{\lambda v.k\,v})$$

(which requires of course the obvious adaptation of the BNF of CPS terms and of the corresponding CPS abstract machines). The stack machine of Figure 2 makes it clear that in the latter term, the call $f\,x$ is processed with $\lambda v.k\,v$ pushed on the control stack, which is not a properly tail-recursive behaviour. Conversely, with the obvious adaptation of Figure 2, in the former term, $f\,x$ is processed with the same control stack, i.e., in a properly tail-recursive manner.[3]

9 Conclusion

CPS has the reputation of being more expensive than direct style, by its structure. We have observed that the structure of CPS can be exploited to reduce its implementation cost essentially to the one of direct style. To this end, we have presented four abstract machines: (1) a bare one for CPS terms, obtained by specializing a left-to-right, call-by-value abstract machine to the BNF of CPS terms, and using substitution for β-reduction; (2) one that does not use substitution for continuation parameters, but a control stack instead; (3) one that does not use substitution for parameters of continuations, but a data stack instead; and (4) one that uses substitution neither for continuation parameters nor for parameters of continuations, but two stacks instead. We have proven the equivalence of these four abstract machines (the equivalence proof between (1) and (3) and (3) and (4) is outlined in the appendix of the present submission). We have formalized all the abstract machines in Elf [21], a constraint logic-programming language based on the logical framework LF [14] and we are currently implementing their equivalence proofs based on the techniques outlined in our earlier work [8].

The expensive reputation of CPS is thus unfounded. These results also apply to other evaluation orders than left-to-right call by value, and also to A-normal forms (monadic style).

The enabling technologies of our work are the occurrence conditions of parameters in CPS terms, and the proof technique for establishing the correctness of these conditions (a unary logical relation) [8].

As outlined at the end of Section 1, we believe that the conditions and the abstract machines can be extended for a full-fledged Scheme-like language. The facts that the control stack contains the skeleton of

[3] In fact, a properly tail-recursive CPS transformation can make a significant difference in a CPS compiler such as the one for Standard ML of New Jersey (Trevor Jim and Andrew Appel, personal communication to the first author, San Francisco, California, June 1992).

the continuation, and that the data stack contains all the free variables of the continuation makes it obvious how to handle call/cc. Furthermore, the development should also apply to statically typed languages such as ML, as long as polymorphism is handled correctly as suggested by Duba, Harper, and MacQueen [9].

A Equivalence proofs

A.1 Equivalence between the abstract machines of Figure 1 and Figure 5

We use the notation $t\{\xi\} = t' \ ; \ \xi'$ to mean that we retrieve the value of t from the top of the stack if it is a parameter of continuation and we return t identically otherwise.

$$\begin{array}{rcl} v\{\xi,\, t\} & = & t \ ; \ \xi \\ (\lambda x.\lambda k.e)\{\xi\} & = & \lambda x.\lambda k.e \ ; \ \xi \\ x\{\xi\} & = & x \ ; \ \xi \end{array}$$

The result is a CPS trivial expression t' and a new data stack ξ'.

We use the notation $e\{\xi\}$ to mean that each "v" in e is substituted with the corresponding intermediate result from the data stack ξ. Substitution is carried out as follows:

$$\frac{t_1\{\xi\} = t_1' \ ; \ \xi_1 \qquad t_0\{\xi_1\} = t_0' \ ; \ \xi_0 \qquad e\{\xi_0,\, v\} = e'}{(t_0\, t_1\, (\lambda v.e))\{\xi\} = t_0'\, t_1'\, (\lambda v.e')}$$

$$\frac{t\{\xi\} = t' \ ; \ \bullet}{(k_{init}\, t)\{\xi\} = k_{init}\, t'}$$

$$\frac{t\{\xi\} = t' \ ; \ \xi_1 \qquad e\{\xi_1,\, v\} = e'}{((\lambda v.e)\, t)\{\xi\} = (\lambda v.e')\, t'}$$

The result is an expression e'.

In the lemmas and theorems below we always assume that the occurrence conditions for continuations and continuation parameters in expressions are satisfied for an appropriate parameter stack. For example, when we write $e\{\xi\}$ we assume that $\Xi \models^{\mathrm{CExp}}_{\mathrm{Var}} e$ for a parameter stack of the same length as ξ.

We begin with the fundamental properties of the substitution $e\{\xi\}$ which are central in the proof of the main theorems. In each case, the proof proceeds by a simple structural induction on e.

Lemma 1 *The following properties hold for stack substitution.*

1. $e\{\Xi\} = e$.
2. $(e\{\xi, v, \xi'\})[t/v] = e\{\xi, t, \xi'\}$.
3. $e[(\lambda v.e'\{\xi', v\})/k]\{\xi\} = e[\lambda v.e'/k]\{\xi', \xi\}$.

We also need Dzafic's result that the occurrence conditions are closed under substitution [10]. From this we can prove the equivalence of the machines by structural induction on the derivations of the evaluation judgment. In addition to the occurrence conditions, we now also assume that the data stacks ξ are closed, that is, they contain no free variables x or v.

Theorem 2 *The bare machine and the data-stack machine are equivalent:*

1. $\xi \vdash^{\mathrm{CTriv}}_{\mathrm{vstack}} t \hookrightarrow t' \, ; \, \xi'$ *iff* $t\{\xi\} = t' \ ; \ \xi'$.
2. $\xi \vdash^{\mathrm{CExp}}_{\mathrm{vstack}} e \hookrightarrow a$ *iff* $\vdash^{\mathrm{CExp}}_{\mathrm{bare}} e\{\xi\} \hookrightarrow a$.
3. $\vdash^{\mathrm{CRoot}}_{\mathrm{vstack}} r \hookrightarrow a$ *iff* $\vdash^{\mathrm{CRoot}}_{\mathrm{bare}} r \hookrightarrow a$.

$$\frac{}{\bullet \models^{\mathrm{CStack}}_{\mathrm{Var}} k_{init}} \qquad \frac{\Xi,\, v \models^{\mathrm{CExp}}_{\mathrm{Var}} e \qquad \Xi' \models^{\mathrm{CStack}}_{\mathrm{Var}} \kappa}{\Xi',\, \Xi \models^{\mathrm{CStack}}_{\mathrm{Var}} \kappa,\, \lambda v.e}$$

$$\frac{}{\models^{\mathrm{CStack}}_{\mathrm{Cont}} k_{init}} \qquad \frac{k \models^{\mathrm{CExp}}_{\mathrm{Cont}} e \qquad \models^{\mathrm{CStack}}_{\mathrm{Cont}} \kappa}{\models^{\mathrm{CStack}}_{\mathrm{Cont}} \kappa,\, \lambda v.e} \text{for some appropriate } k$$

Figure 7: Occurrence properties of control stacks

A.2 Equivalence between the abstract machines of Figure 5 and Figure 6

We use the notation $e\{\kappa\}$ to mean that each "k" in e is substituted with the corresponding continuation from the control stack κ. Substitution is carried out as follows:

$$\begin{array}{rcl} (k\,t)\{k_{init}\} & = & k_{init}\,t \\ (k\,t)\{\kappa,\, \lambda v_1.e_1\} & = & (\lambda v_1.e_1\{\kappa\})\,t \\ ((\lambda v_1.e_1)\,t)\{\kappa\} & = & (\lambda v_1.e_1\{\kappa\})\,t \\ (t_0\,t_1\,(\lambda v_1.e_1))\{\kappa\} & = & t_0\,t_1\,(\lambda v_1.e_1\{\kappa\}) \end{array}$$

The result is a CPS expression e'.

For the proof, we need to define the validity of a continuation stack, which reduces to the occurrence conditions on the continuations it contains (see Figure 7).

The main theorem requires only one simple lemma regarding the substitution for continuation parameters. As before, we assume all control stacks to satisfy the occurrence conditions with respect to appropriate data stacks.

Lemma 3 *The following properties hold for continuation substitutions:*

1. *If* $\Xi \models^{\mathrm{CExp}}_{\mathrm{Var}} e$ *and* $\Xi' \models^{\mathrm{CStack}}_{\mathrm{Var}} \kappa$ *then* $\Xi',\, \Xi \models^{\mathrm{CExp}}_{\mathrm{Var}} e\{\kappa\}$.
2. $e[(\lambda v.e'\{\kappa\})/k] = e\{\kappa,\, \lambda v.e'\}$.

The main theorem once again follows by induction on the structure of the given derivations.

Theorem 4 *The data-stack machine and the two-stacks machine are equivalent:*

1. $\kappa\,;\, \xi, \xi' \vdash^{\mathrm{CExp}}_{\mathrm{cvstack}} e \hookrightarrow a$ *iff* $\xi, \xi' \vdash^{\mathrm{CExp}}_{\mathrm{vstack}} e\{\kappa\} \hookrightarrow a$.
2. $\vdash^{\mathrm{CRoot}}_{\mathrm{vstack}} r \hookrightarrow a$ *iff* $\vdash^{\mathrm{CRoot}}_{\mathrm{cvstack}} r \hookrightarrow a$.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991.

[3] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[4] William Clinger. The Scheme 311 compiler, an exercise in Denotational Semantics. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, Austin, Texas, August 1984.

[5] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[6] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

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

[8] Olivier Danvy and Frank Pfenning. The occurrence of continuation parameters in CPS terms. Technical report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1995.

[9] 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.

[10] Belmina Dzafic. Closure properties for an occurrence condition in CPS terms. Student report, DAIMI, Computer Science Department, Aarhus University, Aarhus, Denmark, October 1996.

[11] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison Wesley, 1988.

[12] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[13] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, December 1992.

[14] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. A preliminary version appeared in the proceedings of the First IEEE Symposium on Logic in Computer Science, pages 194–204, June 1987.

[15] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [3], pages 458–471.

[16] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 1996. To appear.

[17] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986.

[18] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[19] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London, and John Wiley, New York, 1976.

[20] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[21] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

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

[23] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[24] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3/4):289–360, December 1993.

[25] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[26] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In Boehm [3], pages 188–201.

[27] Philip Wadler. The essence of functional programming (tutorial). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

[28] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.