The Essence of Compiling with Continuations

Cormac Flanagan* Amr Sabry* Bruce F. Duba Matthias Felleisen*

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

In order to simplify the compilation process, many compilers for higher-order languages use the continuation-passing style (CPS) transformation in a first phase to generate an intermediate representation of the source program. The salient aspect of this intermediate form is that all procedures take an argument that represents the rest of the computation (the "continuation"). Since the naïve CPS transformation considerably increases the size of programs, CPS compilers perform reductions to produce a more compact intermediate representation. Although often implemented as a part of the CPS transformation, this step is conceptually a second phase. Finally, code generators for typical CPS compilers treat continuations specially in order to optimize the interpretation of continuation parameters.

A thorough analysis of the abstract machine for CPS terms shows that the actions of the code generator *invert* the naïve CPS translation step. Put differently, the combined effect of the three phases is equivalent to a source-to-source transformation that simulates the compaction phase. Thus, fully developed CPS compilers do not need to employ the CPS transformation but can achieve the same results with a simple source-level transformation.

1 Compiling with Continuations

A number of prominent compilers for applicative higher-order programming languages use the language of continuation-passing style (CPS) terms as their intermediate representation for programs [2, 14, 18, 19]. This strategy apparently offers two major advantages. First, Plotkin [16] showed that the λ-value calculus based on the β-value rule is an operational semantics for the source language, that the conventional *full* λ-calculus is a semantics for the intermediate language, and, most importantly, that the λ-calculus proves more equations between CPS terms than the λ_v-calculus does between corresponding terms of the source language. Translated into practice, a compiler can perform more transformations on the intermediate language than on the source language [2:4–5]. Second, the language of CPS terms is basically a stylized assembly language, for which it is easy to generate actual assembly programs for different machines [2, 13, 20]. In short, the CPS transformation provides an organizational principle that simplifies the construction of compilers.

To gain a better understanding of the role that the CPS transformation plays in the compilation process, we recently studied the precise connection between the λ_v-calculus for source terms and the λ-calculus for CPS terms. The result of this research [17] was an extended λ_v-calculus that precisely corresponds to the λ-calculus of the intermediate CPS language and that is still semantically sound for the source language. The extended calculus includes a set of reductions, called the A-reductions, that simplify source terms in the same manner as realistic CPS transformations simplify the output of the naïve transformation. The effect of these reductions is to name all intermediate results and to merge code blocks across declarations and conditionals. Direct compilers typically perform these reductions on an *ad hoc* and incomplete basis.[1]

The goal of the present paper is to show that the true purpose of using CPS terms as an intermediate representation is also achieved by using A-normal forms. We base our argument on a formal development of the abstract machine for the intermediate code of a CPS-based compiler. The development shows that this machine is

*Supported in part by NSF grants CCR 89-17022 and CCR 91-22518 and Texas ATP grant 91-003604014.

[1]Personal communication: H. Boehm (also [4]), K. Dybvig, R. Hieb (April 92).

0

$$
\begin{array}{rcl}
M & ::= & V \\
 & & \mid (\textbf{let}\ (x\ M_1)\ M_2) \\
 & & \mid (\textbf{if0}\ M_1\ M_2\ M_3) \\
 & & \mid (M\ M_1\ \ldots\ M_n) \\
 & & \mid (O\ M_1\ \ldots\ M_n) \\
 & & \\
V & ::= & c \mid x \mid (\lambda x_1 \ldots x_n.M)
\end{array}
\qquad
\begin{array}{lcl}
V & \in & \mathit{Values} \\
c & \in & \mathit{Constants} \\
x & \in & \mathit{Variables} \\
O & \in & \mathit{Primitive\ Operations}
\end{array}
$$

Figure 1: Abstract Syntax of Core Scheme (CS)

identical to a machine for A-normal forms. Thus, the back end of an A-normal form compiler can employ the same code generation techniques that a CPS compiler uses. In short, A-normalization provides an organizational principle for the construction of compilers that combines various stages of fully developed CPS compilers in one straightforward transformation.

The next section reviews the syntax and semantics of a typical higher-order applicative language. The following section analyses CPS compilers for this language. Section 4 introduces the A-reductions and describes A-normal form compilers. Section 5 proves the equivalence between A-normal form compilers and realistic CPS compilers. The benefits of using A-normal form terms as an intermediate representation for compilers is the topic of Section 6. The appendix includes a linear A-normalization algorithm.

2 Core Scheme

The source language is a simple higher-order applicative language. For our purposes, it suffices to consider the language of abstract syntax trees that is produced by the lexical and syntactic analysis module of the compiler: see Figure 1 for the context-free grammar of this language. The terms of the language are either values or non-values. Values include constants, variables, and procedures. Non-values include **let**-expressions (blocks), conditionals, function applications and primitive operations.[2] The sets of constants and primitive procedures are intentionally unspecified. For our purposes, it is irrelevant whether the language is statically typed like ML or dynamically typed like Scheme.

The language Core Scheme has the following context-sensitive properties, which are assumed to be checked by the front-end of the compiler. In the procedure $(\lambda x_1 \ldots x_n.M)$ the parameters $x_1, \ldots, x_n$ are mutually distinct and *bound* in the body M. Similarly, the expression $(\textbf{let}\ (x\ M_1)\ M_2)$ binds x in M_2. A variable that is not bound by a λ or a **let** is *free*; the set of free variables in a term M is $FV(M)$. Like Barendregt [3:ch 2,3], we identify terms modulo bound variables and we assume that free and bound variables of distinct terms do not interfere in definitions or theorems.

The semantics of the language is a partial function from programs to answers. A *program* is a term with no free variables and an *answer* is a member of the syntactic category of constants. Following conventional tradition [1], we specify the operational semantics of Core Scheme with an abstract machine. The machine we use, the CEK machine [10], has three components: a control string C, an environment E that includes bindings for all free variables in C, and a continuation K that represents the "rest of the computation".

The CEK machine changes state according to the transition function in Figure 2. For example, the state transition for the block $(\textbf{let}\ (x\ M_1)\ M_2)$ starts the evaluation of M_1 in the current environment E and modifies the continuation register to encode the rest of the computation $\langle \textbf{lt}\ x, M_2, E, K\rangle$. When the new continuation receives a value, it extends the environment with a value for x and proceeds with the evaluation of M_2. The remaining clauses have similarly intuitive explanations.

The relation $\longmapsto^*$ is the reflexive transitive closure of the transition function. The function γ constructs machine values from syntactic values and environments. The notation $E(x)$ refers to an algorithm for looking up the value of x in the environment E. The operation $E[x_1 := V_1^*, \ldots, x_n := V_n^*]$ extends the environment E such that subsequent lookups of x_i return the value V_i^*. The object $\langle \textbf{cl}\ x_1 \ldots x_n, M, E\rangle$ is a *closure*, a record that contains the code for M and values for the free variables of $(\lambda x_1 \ldots x_n.M)$. The partial function δ abstracts the semantics of the primitive operations.

The CEK machine provides a model for designing direct compilers [6, 11, 15]. A compiler based on the CEK machine implements an efficient representation for environments, *e.g.*, displays, and for continuations, *e.g.*, a stack.[3] The machine code produced by such a compiler

[2] The language is overly simple but contains all ingredients that are necessary to generate our result for full ML or Scheme. In particular, the introduction of assignments, and even control operators, is orthogonal to the analysis of the CPS-based compilation strategy.

[3] The machine also characterizes compilers for first-order languages, *e.g.*, Fortran. In this case, the creation and deletion of the environment and continuation components always follows a stack-like behavior. Hence the machine reduces to a traditional stack machine.

Semantics: Let $M \in CS$,

$$eval_d(M) = c \quad \text{if} \quad \langle M, \emptyset, \mathbf{stop}\rangle \longmapsto^* \langle \mathbf{stop}, c\rangle.$$

Data Specifications:

$$\begin{array}{rcrcll}
S & \in & State_d & = & CS \times Env_d \times Cont_d \mid Cont_d \times Value_d & \text{(machine states)} \\
E & \in & Env_d & = & Variables \longrightarrow Value_d & \text{(environments)} \\
V^* & \in & Value_d & = & c \mid \langle \mathbf{cl}\ x_1 \ldots x_n, M, E\rangle & \text{(machine values)} \\
K & \in & Cont_d & = & \mathbf{stop} \mid \langle \mathbf{ap}\ \langle \ldots, V^*, \bullet, M, \ldots\rangle, E, K\rangle \mid \langle \mathbf{lt}\ x, M, E, K\rangle & \text{(continuations)} \\
 & & & & \mid \langle \mathbf{if}\ M_1, M_2, E, K\rangle \mid \langle \mathbf{pr}\ O, \langle \ldots, V^*, \bullet, M, \ldots\rangle, E, K\rangle &
\end{array}$$

Transition Rules:

$$\begin{array}{rcl}
\langle V, E, K\rangle & \longmapsto & \langle K, \gamma(V, E)\rangle \\
\langle (\mathbf{let}\ (x\ M_1)\ M_2), E, K\rangle & \longmapsto & \langle M_1, E, \langle \mathbf{lt}\ x, M_2, E, K\rangle\rangle \\
\langle (\mathbf{if0}\ M_1\ M_2\ M_3), E, K\rangle & \longmapsto & \langle M_1, E, \langle \mathbf{if}\ M_2, M_3, E, K\rangle\rangle \\
\langle (M\ M_1\ \ldots\ M_n), E, K\rangle & \longmapsto & \langle M, E, \langle \mathbf{ap}\ \langle \bullet, M_1, \ldots, M_n\rangle, E, K\rangle\rangle \\
\langle (O\ M_1\ M_2\ \ldots\ M_n), E, K\rangle & \longmapsto & \langle M_1, E, \langle \mathbf{pr}\ O, \langle \bullet, M_2, \ldots, M_n\rangle, E, K\rangle\rangle \\
 & & \\
\langle \langle \mathbf{lt}\ x, M, E, K\rangle, V^*\rangle & \longmapsto & \langle M, E[x := V^*], K\rangle \\
\langle \langle \mathbf{if}\ M_1, M_2, E, K\rangle, 0\rangle & \longmapsto & \langle M_1, E, K\rangle \\
\langle \langle \mathbf{if}\ M_1, M_2, E, K\rangle, V^*\rangle & \longmapsto & \langle M_2, E, K\rangle \quad \text{where } V^* \neq 0 \\
\langle \langle \mathbf{ap}\ \langle \ldots, V_i^*, \bullet, M, \ldots\rangle, E, K\rangle, V_{i+1}^*\rangle & \longmapsto & \langle M, E, \langle \mathbf{ap}\ \langle \ldots, V_i^*, V_{i+1}^*, \bullet, \ldots\rangle, E, K\rangle\rangle \\
\langle \langle \mathbf{ap}\ V^*, V_1^*, \ldots, \bullet\rangle, E, K\rangle, V_n^*\rangle & \longmapsto & \langle M', E'[x_1 := V_1^*, \ldots, x_n := V_n^*], K\rangle \qquad \text{if } V^* = \langle \mathbf{cl}\ x_1 \ldots x_n, M', E'\rangle \\
\langle \langle \mathbf{pr}\ O, \langle \ldots, V_i^*, \bullet, M, \ldots\rangle, E, K\rangle, V_{i+1}^*\rangle & \longmapsto & \langle M, E, \langle \mathbf{pr}\ O, \langle \ldots, V_i^*, V_{i+1}^*, \bullet, \ldots\rangle, E, K\rangle\rangle \\
\langle \langle \mathbf{pr}\ O, \langle V_1^*, \ldots, \bullet\rangle, E, K\rangle, V_n^*\rangle & \longmapsto & \langle K, \delta(O, V_1^*, \ldots, V_n^*)\rangle \qquad \text{if } \delta(O, V_1^*, \ldots, V_n^*) \text{ is defined}
\end{array}$$

Converting syntactic values to machine values:

$$\begin{array}{rcl}
\gamma(c, E) & = & c \\
\gamma(x, E) & = & E(x) \\
\gamma((\lambda x_1 \ldots x_n.M), E) & = & \langle \mathbf{cl}\ x_1 \ldots x_n, M, E\rangle
\end{array}$$

Figure 2: The CEK-machine

realizes the abstract operations specified by the CEK machine by manipulating these concrete representations of environments and continuations.

3 CPS Compilers

Several compilers map source terms to a CPS intermediate representation before generating machine code. The function $\mathcal{F}$ [12] in Figure 3 is the basis of CPS transformations used in various compilers [2, 14, 19]. It uses special λ-expressions or continuations to encode the rest of the computation, thus shifting the burden of maintaining control information from the abstract machine to the code. The notation $(\overline{\lambda} x. \cdots)$ marks the *administrative* λ-expressions introduced by the CPS transformation. The primitive operation O' used in the CPS language is equivalent to the operation O for the source language, except that O' takes an extra continuation argument, which receives the result once it is computed.

The transformation $\mathcal{F}$ introduces a large number of administrative λ-expressions. For example, $\mathcal{F}$ maps the code segment

$$N \stackrel{df}{=} (+\ (+\ 2\ 2)\ (\mathbf{let}\ (x\ 1)\ (f\ x)))$$

into the CPS term

$$\begin{array}{l}
((\overline{\lambda} k_2.\ ((\overline{\lambda} k_3.\ (k_3\ 2)) \\
\qquad (\overline{\lambda} t_1.\ ((\overline{\lambda} k_4.\ (k_4\ 2)) \\
\qquad\qquad (\overline{\lambda} t_2.\ (+'\ k_2\ t_1\ t_2)))))) \\
\ (\overline{\lambda} t_3.\ ((\overline{\lambda} k_5. \\
\qquad ((\overline{\lambda} k_6.\ (k_6\ 1)) \\
\qquad\ (\overline{\lambda} t_4.\ (\mathbf{let}\ (x\ t_4) \\
\qquad\qquad ((\overline{\lambda} k_7.\ ((\overline{\lambda} k_8.\ (k_8\ f)) \\
\qquad\qquad\qquad (\overline{\lambda} t_5.\ ((\overline{\lambda} k_9.\ (k_9\ x)) \\
\qquad\qquad\qquad\qquad (\overline{\lambda} t_6.\ (t_5\ k_7\ t_6)))))) \\
\qquad\qquad k_5))))) \\
\qquad (\overline{\lambda} t_7.\ (+'\ k\ t_3\ t_7))))))
\end{array}$$

By convention, we ignore the context $(\lambda k.[\,])$ enclosing all CPS *programs*.

To decrease the number of administrative λ-abstractions, realistic CPS compilers include a simplification phase for compacting CPS terms [2:68–69, 14:5–6, 19:49–51]. For an analysis of this simplification phase,

$$
\begin{aligned}
\mathcal{F}[\![V]\!] &= \overline{\lambda} k.(k\ \Phi[\![V]\!]) \\
\mathcal{F}[\![(\mathbf{let}\ (x\ M_1)\ M_2)]\!] &= \overline{\lambda} k.(\mathcal{F}[\![M_1]\!]\ \overline{\lambda} t.(\mathbf{let}\ (x\ t)\ (\mathcal{F}[\![M_2]\!]\ k))) \\
\mathcal{F}[\![(\mathbf{if0}\ M_1\ M_2\ M_3)]\!] &= \overline{\lambda} k.(\mathcal{F}[\![M_1]\!]\ \overline{\lambda} t.(\mathbf{if0}\ t\ (\mathcal{F}[\![M_2]\!]\ k)\ (\mathcal{F}[\![M_3]\!]\ k))) \\
\mathcal{F}[\![(M\ M_1\ \cdots\ M_n)]\!] &= \overline{\lambda} k.(\mathcal{F}[\![M]\!]\ \overline{\lambda} t.(\mathcal{F}[\![M_1]\!]\ \overline{\lambda} t_1.\cdots\ (\mathcal{F}[\![M_n]\!]\ \overline{\lambda} t_n.(t\ k\ t_1\ \ldots\ t_n)))) \\
\mathcal{F}[\![(O\ M_1\ \cdots\ M_n)]\!] &= \overline{\lambda} k.(\mathcal{F}[\![M_1]\!]\ \overline{\lambda} t_1.\cdots\ (\mathcal{F}[\![M_n]\!]\ \overline{\lambda} t_n.(O'\ k\ t_1\ \ldots\ t_n))) \\
\\
\Phi[\![c]\!] &= c \\
\Phi[\![x]\!] &= x \\
\Phi[\![\lambda x_1 \cdots x_n.M]\!] &= \lambda k x_1 \cdots x_n.\mathcal{F}[\![M]\!]
\end{aligned}
$$

Figure 3: The CPS transformation $\mathcal{F}$

its optimality, and how it can be combined with $\mathcal{F}$, we refer the reader to Danvy and Filinski [9] and Sabry and Felleisen [17]. This phase simplifies administrative redexes of the form $((\overline{\lambda} x.P)\ Q)$ according to the rule:

$$((\overline{\lambda} x.P)\ Q) \longrightarrow P[x := Q] \qquad (\overline{\beta})$$

The term $P[x := Q]$ is the result of the capture-free substitution of all free occurrences of x in P by Q; for example, $(\lambda x.xz)[z := (\lambda y.x)] = (\lambda u.u(\lambda y.x))$. Applying the reduction $\overline{\beta}$ to all the administrative redexes in our previous example produces the following $\overline{\beta}$-normal form term:

$$cps(N) = (+'\ (\overline{\lambda} t_1.\ (\mathbf{let}\ (x\ 1)\ (f\ (\overline{\lambda} t_2.\ (+'\ k\ t_1\ t_2))\ x)))\ 2\ 2)$$

The reduction $\overline{\beta}$ is strongly-normalizing on the language of CPS terms [17]. Hence, the simplification phase of a CPS compiler can remove all $\overline{\beta}$-redexes from the output of the translation $\mathcal{F}$.[4] After the simplification phase, we no longer need to distinguish between regular and administrative λ-expressions, and use the notation $(\lambda.\cdots)$ for both classes of λ-expression. With this convention, the language of $\overline{\beta}$-normal forms, CPS(CS), is the following [17]:

$$
\begin{array}{lll}
P ::= & (k\ W) & (return) \\
 & \mid (\mathbf{let}\ (x\ W)\ P) & (bind) \\
 & \mid (\mathbf{if0}\ W\ P_1\ P_2) & (branch) \\
 & \mid (W\ k\ W_1\ \ldots\ W_n) & (tail\ call) \\
 & \mid (W\ (\lambda x.P)\ W_1\ \ldots\ W_n) & (call) \\
 & \mid (O'\ k\ W_1\ \ldots\ W_n) & (prim\text{-}op) \\
 & \mid (O'\ (\lambda x.P)\ W_1\ \ldots\ W_n) & (prim\text{-}op) \\
W ::= & c \mid x \mid (\lambda k x_1 \ldots x_n.P) & (values)
\end{array}
$$

Indeed, this language is typical of the intermediate representation used by CPS compilers [2, 14, 19].

Naïve CPS Compilers The abstract machine that characterizes the code generator of a naïve CPS compiler is the $\mathrm{C_{cps}E}$ machine. Since terms in CPS(CS) contain an encoding of control-flow information, the machine does not require a continuation component (K) to record the rest of the computation. Evaluation proceeds according to the state transition function in Figure 4. For example, the state transition for the tail call $(W\ k\ W_1\ \ldots\ W_n)$ computes a closure $\langle \mathbf{cl}\ k'x_1 \ldots x_n, P', E'\rangle$ corresponding to W, extends E' with the values of $k, W_1, \ldots, W_n$ and starts the interpretation of P'.

Realistic CPS Compilers Although the $\mathrm{C_{cps}E}$ machine describes what a naïve CPS compiler *would* do, typical compilers deviate from this model in two regards.

First, the naïve abstract machine for CPS code represents the continuation as an ordinary closure. Yet, realistic CPS compilers "mark" the continuation closure as a special closure. For example, Shivers partitions procedures and continuations in order to improve the data flow analysis of CPS programs [18:sec 3.8.3]. Also, in both Orbit [14] and Rabbit [19], the allocation strategy of a closure changes if the closure is a continuation. Similarly, Appel [2:114–124] describes various techniques for closure allocation that treat the continuation closure in a special way.

In order to reflect these changes in the machine, we tag continuation closures with a special marker "**ar**" that describes them as **a**ctivation **r**ecords.

Second, the CPS representation of any user-defined procedure receives a continuation argument. However, Steele [19] modifies the CPS transformation with a "continuation variable hack" [19:94] that recognizes instances of CPS terms like $((\lambda k_1 \cdots .P)\ k_2\ \ldots)$ and trans-

[4] The CPS translation of a conditional expression contains two references to the continuation variable k. Thus, the $\overline{\beta}$-normalization phase can produce exponentially larger output. Modifying the CPS algorithm to avoid duplicating k removes the potential for exponential growth. The rest of our technical development can be adapted *mutatis mutandis*.

Semantics: Let $P \in \text{CPS}(CS)$,

$$eval_n(P) = c \quad \text{if} \quad \langle P, \emptyset[k := \langle \mathbf{cl}\ x, (k\ x), \emptyset[k := \mathbf{stop}]\rangle]\rangle \longmapsto^* \langle (k\ x), \emptyset[x := c, k := \mathbf{stop}]\rangle.$$

Data Specifications:

$$\begin{array}{rcrcll}
S_n & \in & State_n & = & \text{CPS}(CS) \times Env_n & \text{(machine states)} \\
E & \in & Env_n & = & Variables \longrightarrow Value_n & \text{(environments)} \\
W^* & \in & Value_n & = & c \mid \langle \mathbf{cl}\ kx_1 \ldots x_n, P, E\rangle \mid \langle \mathbf{cl}\ x, P, E\rangle \mid \mathbf{stop} & \text{(machine values)}
\end{array}$$

Transition Rules:

$$\begin{array}{rcl}
\langle (k\ W), E\rangle & \longmapsto & \langle P', E'[x := \mu(W, E)]\rangle \quad \text{where } E(k) = \langle \mathbf{cl}\ x, P', E'\rangle \\
\langle (\mathbf{let}\ (x\ W)\ P), E\rangle & \longmapsto & \langle P, E[x := \mu(W, E)]\rangle \\
\langle (\mathbf{if0}\ W\ P_1\ P_2), E\rangle & \longmapsto & \langle P_1, E\rangle \quad \text{where } \mu(W, E) = 0 \\
 & \text{or} & \langle P_2, E\rangle \quad \text{where } \mu(W, E) \neq 0 \\
\langle (W\ k\ W_1\ \ldots\ W_n), E\rangle & \longmapsto & \langle P', E'[k' := E(k), x_1 := W_1^*, \ldots, x_n := W_n^*]\rangle \\
 & & \text{where } \mu(W, E) = \langle \mathbf{cl}\ k'x_1 \ldots x_n, P', E'\rangle \text{ and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E) \\
\langle (W\ (\lambda x.P)\ W_1\ \ldots\ W_n), E\rangle & \longmapsto & \langle P', E'[k' := \langle \mathbf{cl}\ x, P, E\rangle, x_1 := W_1^*, \ldots, x_n := W_n^*]\rangle \\
 & & \text{where } \mu(W, E) = \langle \mathbf{cl}\ k'x_1 \ldots x_n, P', E'\rangle \text{ and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E) \\
\langle (O'\ k\ W_1\ \ldots\ W_n), E\rangle & \longmapsto & \langle P', E'[x := \delta_c(O', W_1^*, \ldots, W_n^*)]\rangle \quad \text{if } \delta_c(O', W_1^*, \ldots, W_n^*) \text{ is defined,} \\
 & & \text{where } E(k) = \langle \mathbf{cl}\ x, P', E'\rangle \text{ and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E) \\
\langle (O'\ (\lambda x.P)\ W_1\ \ldots\ W_n), E\rangle & \longmapsto & \langle P, E[x := \delta_c(O', W_1^*, \ldots, W_n^*)]\rangle \quad \text{if } \delta_c(O', W_1^*, \ldots, W_n^*) \text{ is defined,} \\
 & & \text{and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E)
\end{array}$$

Converting syntactic values to machine values:

$$\begin{array}{rcl}
\mu(c, E) & = & c \\
\mu(x, E) & = & E(x) \\
\mu((\lambda kx_1 \ldots x_n.P), E) & = & \langle \mathbf{cl}\ kx_1 \ldots x_n, P, E\rangle
\end{array}$$

Figure 4: The naïve CPS abstract machine: the C_{cps}E machine.

forms them to $((\lambda \cdots .P[k_1 := k_2])\ \ldots)$. This "optimization" eliminates "some of the register shuffling" [19:94] during the evaluation of the term. Appel [2] achieves the same effect without modifying the CPS transformation by letting the variables k_1 and k_2 *share* the same register during the procedure call.

In terms of the CPS abstract machine, the optimization corresponds to a modification of the operation $E'[k' := E(k), x_1 := W_1^*, \ldots, x_n := W_n^*]$ to $E'[x_1 := W_1^*, \ldots, x_n := W_n^*]$ such that E and E' share the binding of k. In order to make the sharing explicit, we split the environment into two components: a component E^k that includes the binding for the continuation, and a component E^- that includes the rest of the bindings, and treat each component independently. This optimization relies on the fact that every control string has exactly one free continuation variable, which implies that the corresponding value can be held in a special register.[5]

Performing these modifications on the naïve abstract machine produces the realistic CPS abstract machine in Figure 5. The new $\text{C}_{\text{cps}}\text{EK}$ machine extracts the information regarding the continuation from the CPS terms and manages the continuation in an optimized way. For example, the state transition for the tail call $(W\ k\ W_1\ \ldots\ W_n)$ evaluates W to a closure $\langle \mathbf{cl}\ kx_1 \ldots x_n, P', E_1^-\rangle$, extends E_1^- with the values of $W_1, \ldots, W_n$ and starts the execution of P'. In particular, there is no need to extend E_1^- with the value of k as this value remains in the environment component E^k.

[5] This fact also holds in the presence of control operators as there is always one identifiable *current* continuation.

4 A-Normal Form Compilers

A close inspection of the $\text{C}_{\text{cps}}\text{EK}$ machine reveals that the control strings often contain redundant information considering the way instructions are executed. First, a *return* instruction, *i.e.*, the transition $\xmapsto{(1)}_c$, dispatches on the term $(k\ W)$, which informs the machine that the "return address" is denoted by the value of the variable k. The machine ignores this information since a *return* instruction automatically uses the value of register E^k as the "return address". Second, the *call* instructions, *i.e.*, transitions $\xmapsto{(4)}_c$ and $\xmapsto{(5)}_c$, invoke closures that expect, among other arguments, a continuation k.

Semantics: Let $P \in \mathrm{CPS}(CS)$,

$$eval_c(P) = c \quad \text{if} \quad \langle P, \emptyset, \langle \mathbf{ar}\ x, (k\ x), \emptyset, \mathbf{stop}\rangle\rangle \longmapsto_c^* \langle (k\ x), \emptyset[x := c], \mathbf{stop}\rangle.$$

Data Specifications:

$$\begin{array}{rclll}
S_c & \in & State_c & = & \mathrm{CPS}(CS) \times Env_c \times Cont_c \quad \text{(machine states)} \\
E^- & \in & Env_c & = & Variables \multimap Value_c \quad \text{(environments)} \\
W^* & \in & Value_c & = & c \mid \langle \mathbf{cl}\ kx_1 \ldots x_n, P, E^- \rangle \quad \text{(machine values)} \\
E^k & \in & Cont_c & = & \mathbf{stop} \mid \langle \mathbf{ar}\ x, P, E^-, E^k \rangle \quad \text{(continuations)}
\end{array}$$

Transition Rules:

$$\begin{array}{rcl}
\langle (k\ W), E^-, E^k \rangle & \stackrel{(1)}{\longmapsto}_c & \langle P', E_1^-[x := \mu(W, E^-)], E_1^k \rangle \qquad \text{where } E^k = \langle \mathbf{ar}\ x, P', E_1^-, E_1^k \rangle \\
\langle (\mathbf{let}\ (x\ W)\ P), E^-, E^k \rangle & \stackrel{(2)}{\longmapsto}_c & \langle P, E^-[x := \mu(W, E^-)], E^k \rangle \\
\langle (\mathbf{if0}\ W\ P_1\ P_2), E^-, E^k \rangle & \stackrel{(3)}{\longmapsto}_c & \langle P_1, E^-, E^k \rangle \quad \text{where } \mu(W, E^-) = 0 \\
& \text{or} & \langle P_2, E^-, E^k \rangle \quad \text{where } \mu(W, E^-) \neq 0 \\
\langle (W\ k\ W_1\ \ldots\ W_n), E^-, E^k \rangle & \stackrel{(4)}{\longmapsto}_c & \langle P', E_1^-[x_1 := W_1^*, \ldots, x_n := W_n^*], E^k \rangle \\
& & \text{where } \mu(W, E^-) = \langle \mathbf{cl}\ k'x_1 \ldots x_n, P', E_1^- \rangle \text{ and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E^-) \\
\langle (W\ (\lambda x.P)\ W_1\ \ldots\ W_n), E^-, E^k \rangle & \stackrel{(5)}{\longmapsto}_c & \langle P', E_1^-[x_1 := W_1^*, \ldots, x_n := W_n^*], \langle \mathbf{ar}\ x, P, E^-, E^k \rangle \rangle \\
& & \text{where } \mu(W, E^-) = \langle \mathbf{cl}\ k'x_1 \ldots x_n, P', E_1^- \rangle \text{ and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E^-) \\
\langle (O'\ k\ W_1\ \ldots\ W_n), E^-, E^k \rangle & \stackrel{(6)}{\longmapsto}_c & \langle P', E_1^-[x := \delta_c(O', W_1^*, \ldots, W_n^*)], E_1^k \rangle \qquad \text{if } \delta_c(O', W_1^*, \ldots, W_n^*) \text{ is defined,} \\
& & \text{where } E^k = \langle \mathbf{ar}\ x, P', E_1^-, E_1^k \rangle \text{ and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E^-) \\
\langle (O'\ (\lambda x.P)\ W_1\ \ldots\ W_n), E^-, E^k \rangle & \stackrel{(7)}{\longmapsto}_c & \langle P, E^-[x := \delta_c(O', W_1^*, \ldots, W_n^*)], E^k \rangle \qquad \text{if } \delta_c(O', W_1^*, \ldots, W_n^*) \text{ is defined,} \\
& & \text{and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E^-)
\end{array}$$

Figure 5: The realistic CPS abstract machine: the C_{cps}EK machine.

Again, the machine ignores the continuation parameter in the closures and manipulate the "global" register E^k instead.

Undoing CPS The crucial insight is that the elimination of the redundant information from the $\mathrm{C}_{\mathrm{cps}}$EK machine corresponds to an inverse CPS transformation [7, 17] on the intermediate code. The function $\mathcal{U}$ in Figure 6 realizes such an inverse [17]. The inverse transformation formalizes our intuition about the redundancies in the $\mathrm{C}_{\mathrm{cps}}$EK machine. It eliminates the variable k from *return* instructions as well as the parameter k from procedures. The latter change implies that continuations are not passed as arguments in function calls but rather become *contexts* surrounding the calls. For example, the code segment $cps(N)$ in Section 3 becomes:

$$\begin{array}{l}
A(N) = (\mathbf{let}\ (t_1\ (+\ 2\ 2)) \\
\qquad\qquad (\mathbf{let}\ (x\ 1) \\
\qquad\qquad\quad (\mathbf{let}\ (t_2\ (f\ x)) \\
\qquad\qquad\qquad (+\ t_1\ t_2))))
\end{array}$$

Based on the above argument, it appears that CPS compilers perform a sequence of three steps:

$$\begin{tikzcd}
CS \bullet \arrow[r, "\text{CPS}"] \arrow[d, dashed, "A"'] & \bullet \arrow[d, "\overline{\beta}\text{-normalization}"] \\
A(CS) \bullet & \bullet\ \mathrm{CPS}(CS) \arrow[l, "\text{un-CPS}"']
\end{tikzcd}$$

The diagram naturally suggests a direct translation A that combines the effects of the three phases. The identification of the translation A requires a theorem relating $\overline{\beta}$-reductions on CPS terms to reductions on the source language. This correspondence of reductions was the subject of our previous paper [17]. The resulting set of source reductions, the A-reductions, is in Figure 7.[6] Since the A-reductions are strongly normalizing, we can characterize the translation A as any function that applies the A-reductions to a source term until it reaches a normal form [17:Theorem 6.4].

The definition of the A-reductions refers to the concept of evaluation contexts. An *evaluation context* is a term with a "hole" (denoted by [ ]) in the place of one subterm. The location of the hole points to the next

[6] Danvy [8] and Weise [21] also recognize that the compaction of CPS terms can be expressed in the source language, but do not explore this topic systematically.

The inverse CPS transformation:

$$
\begin{array}{rcl}
\mathcal{U} : \mathrm{CPS}(CS) & \rightarrow & A(CS) \\
\mathcal{U}[\![(k\ W)]\!] & = & \Psi[\![W]\!] \\
\mathcal{U}[\![(\mathbf{let}\ (x\ W)\ P)]\!] & = & (\mathbf{let}\ (x\ \Psi[\![W]\!])\ \mathcal{U}[\![P]\!]) \\
\mathcal{U}[\![(\mathbf{if0}\ W\ P_1\ P_2)]\!] & = & (\mathbf{if0}\ \Psi[\![W]\!]\ \mathcal{U}[\![P_1]\!]\ \mathcal{U}[\![P_2]\!]) \\
\mathcal{U}[\![(W\ k\ W_1\ \ldots\ W_n)]\!] & = & (\Psi[\![W]\!]\ \Psi[\![W_1]\!]\ \ldots\ \Psi[\![W_n]\!]) \\
\mathcal{U}[\![(W\ (\lambda x.P)\ W_1\ \ldots\ W_n)]\!] & = & (\mathbf{let}\ (x\ (\Psi[\![W]\!]\ \Psi[\![W_1]\!]\ \ldots\ \Psi[\![W_n]\!]))\ \mathcal{U}[\![P]\!]) \\
\mathcal{U}[\![(O'\ k\ W_1\ \ldots\ W_n)]\!] & = & (O\ \Psi[\![W_1]\!]\ \ldots\ \Psi[\![W_n]\!]) \\
\mathcal{U}[\![(O'\ (\lambda x.P)\ W_1\ \ldots\ W_n)]\!] & = & (\mathbf{let}\ (x\ (O\ \Psi[\![W_1]\!]\ \ldots\ \Psi[\![W_n]\!]))\ \mathcal{U}[\![P]\!])
\end{array}
$$

$$
\begin{array}{rcl}
\Psi : W & \rightarrow & V \\
\Psi[\![c]\!] & = & c \\
\Psi[\![x]\!] & = & x \\
\Psi[\![\lambda k x_1 \ldots x_n.P]\!] & = & \lambda x_1 \ldots x_n.\mathcal{U}[\![M]\!]
\end{array}
$$

The language $A(CS)$

:

$$
\begin{array}{rcll}
M & ::= & V & (\textit{return}) \\
& & \mid (\mathbf{let}\ (x\ V)\ M) & (\textit{bind}) \\
& & \mid (\mathbf{if0}\ V\ M\ M) & (\textit{branch}) \\
& & \mid (V\ V_1\ \ldots\ V_n) & (\textit{tail call}) \\
& & \mid (\mathbf{let}\ (x\ (V\ V_1\ \ldots\ V_n))\ M) & (\textit{call}) \\
& & \mid (O\ V_1\ \ldots\ V_n) & (\textit{prim-op}) \\
& & \mid (\mathbf{let}\ (x\ (O\ V_1\ \ldots\ V_n))\ M) & (\textit{prim-op}) \\
\\
V & ::= & c \mid x \mid (\lambda x_1 \ldots x_n.M) & (\textit{values})
\end{array}
$$

Figure 6: The inverse CPS transformation and its output

Evaluation Contexts:

$$\mathcal{E} \ ::= \ [\,] \mid (\mathbf{let}\ (x\ \mathcal{E})\ M) \mid (\mathbf{if0}\ \mathcal{E}\ M\ M) \mid (F\ V\ \cdots\ V\ \mathcal{E}\ M\ \cdots\ M) \qquad \text{where } F = V \text{ or } F = O$$

The A-reductions:

$$
\begin{array}{rcll}
\mathcal{E}[(\mathbf{let}\ (x\ M)\ N)] & \longrightarrow & (\mathbf{let}\ (x\ M)\ \mathcal{E}[N]) \quad \text{where } \mathcal{E} \neq [\,], x \notin FV(\mathcal{E}) & (A_1) \\
\mathcal{E}[(\mathbf{if0}\ V\ M_1\ M_2)] & \longrightarrow & (\mathbf{if0}\ V\ \mathcal{E}[M_1]\ \mathcal{E}[M_2]) \quad \text{where } \mathcal{E} \neq [\,] & (A_2) \\
\mathcal{E}[(F\ V_1\ \ldots\ V_n)] & \longrightarrow & (\mathbf{let}\ (t\ (F\ V_1\ \ldots\ V_n))\ \mathcal{E}[t]) & (A_3) \\
& & \text{where } F = V \text{ or } F = O, \mathcal{E} \neq \mathcal{E}'[(\mathbf{let}\ (z\ [\,])\ M)], \mathcal{E} \neq [\,], t \notin FV(\mathcal{E}) &
\end{array}
$$

Figure 7: Evaluation contexts and the set of A-reductions

subexpression to be evaluated according to the CEK semantics. For example, in an expression (**let** $(x\ M_1)\ M_2$), the next reducible expression must occur within M_1, hence the definition of evaluation contexts includes the clause (**let** $(x\ \mathcal{E})\ M$).

The A-reductions transform programs in a natural and intuitive manner. The first two reductions *merge* code segments across declarations and conditionals. The last reduction *lifts* redexes out of evaluation contexts and *names* intermediate results. Using evaluation contexts and the A-reductions, we can

Semantics: Let $M \in A(CS)$,

$$eval_a(M) = c \quad \text{if} \quad \langle M, \emptyset, \langle \mathbf{ar}\ x, x, \emptyset, \mathbf{stop}\rangle\rangle \longmapsto_a^* \langle x, \emptyset[x := c], \mathbf{stop}\rangle.$$

Data Specifications:

$$\begin{array}{rcrcll}
S_a & \in & State_a & = & A(CS) \times Env_a \times Cont_a & \text{(machine states)} \\
E & \in & Env_a & = & Variables \multimap Value_a & \text{(environments)} \\
V^* & \in & Value_a & = & c \mid \langle \mathbf{cl}\ x_1 \ldots x_n, M, E\rangle & \text{(machine values)} \\
K & \in & Cont_a & = & \mathbf{stop} \mid \langle \mathbf{ar}\ x, M, E, K\rangle & \text{(continuations)}
\end{array}$$

Transition Rules:

$$\begin{array}{rcl}
\langle V, E, K\rangle & \xmapsto{(1)}_a & \langle M', E'[x := \gamma(V,E)], K'\rangle \qquad \text{where } K = \langle \mathbf{ar}\ x, M', E', K'\rangle \\
\langle(\mathbf{let}\ (x\ V)\ M), E, K\rangle & \xmapsto{(2)}_a & \langle M, E[x := \gamma(V,E)], K\rangle \\
\langle(\mathbf{if0}\ V\ M_1\ M_2), E, K\rangle & \xmapsto{(3)}_a & \langle M_1, E, K\rangle \quad \text{where } \gamma(V,E) = 0 \\
 & \text{or} & \langle M_2, E, K\rangle \quad \text{where } \gamma(V,E) \neq 0 \\
\langle(V\ V_1\ \ldots\ V_n), E, K\rangle & \xmapsto{(4)}_a & \langle M', E'[x_1 := V_1^*, \ldots, x_n := V_n^*], K\rangle \\
 & & \text{where } \gamma(V,E) = \langle \mathbf{cl}\ x_1 \ldots x_n, M', E'\rangle \text{ and for } 1 \leq i \leq n,\ V_i^* = \gamma(V_i, E) \\
\langle(\mathbf{let}\ (x\ (V\ V_1\ \ldots\ V_n))\ M), E, K\rangle & \xmapsto{(5)}_a & \langle M', E'[x_1 := V_1^*, \ldots, x_n := V_n^*], \langle \mathbf{ar}\ x, M, E, K\rangle\rangle \\
 & & \text{where } \gamma(V,E) = \langle \mathbf{cl}\ x_1 \ldots x_n, M', E'\rangle \text{ and for } 1 \leq i \leq n,\ V_i^* = \gamma(V_i, E) \\
\langle(O\ V_1\ \ldots\ V_n), E, K\rangle & \xmapsto{(6)}_a & \langle M', E'[x := \delta(O, V_1^*, \ldots, V_n^*)], K'\rangle \qquad \text{if } \delta(O, V_1^*, \ldots, V_n^*) \text{ is defined,} \\
 & & \text{where } K = \langle \mathbf{ar}\ x, M', E', K'\rangle \text{ and for } 1 \leq i \leq n,\ V_i^* = \gamma(V_i, E) \\
\langle(\mathbf{let}\ (x\ (O\ V_1\ \ldots\ V_n))\ M), E, K\rangle & \xmapsto{(7)}_a & \langle M, E[x := \delta(O, V_1^*, \ldots, V_n^*)], K\rangle \qquad \text{if } \delta(O, V_1^*, \ldots, V_n^*) \text{ is defined,} \\
 & & \text{and for } 1 \leq i \leq n,\ V_i^* = \gamma(V_i, E)
\end{array}$$

Figure 8: The C_aEK machine

rewrite our sample code segment N in Section 3 as follows. For clarity, we surround the reducible term with a box:

$$\begin{array}{rll}
N \ = & \boxed{(+\ (+\ 2\ 2)\ (\mathbf{let}\ (x\ 1)\ (f\ x)))} & \\
\longrightarrow & (\mathbf{let}\ (t_1\ (+\ 2\ 2)) & (A_3) \\
 & \quad \boxed{(+\ t_1\ (\mathbf{let}\ (x\ 1)\ (f\ x)))}\,) & \\
\longrightarrow & (\mathbf{let}\ (t_1\ (+\ 2\ 2)) & (A_1) \\
 & \quad (\mathbf{let}\ (x\ 1) & \\
 & \qquad \boxed{(+\ t_1\ (f\ x))}\,)) & \\
\longrightarrow & (\mathbf{let}\ (t_1\ (+\ 2\ 2)) & (A_3) \\
 & \quad (\mathbf{let}\ (x\ 1) & \\
 & \qquad (\mathbf{let}\ (t_2\ (f\ x)) & \\
 & \qquad\quad (+\ t_1\ t_2)))) &
\end{array}$$

The appendix includes a linear algorithm that maps Core Scheme terms to their normal form with respect to the A-reductions.

Compilers In order to establish that the A-reductions generate the actual intermediate code of CPS compilers, we design an abstract machine for the language of A-normal forms, the C_aEK machine, and prove that this machine is "equivalent" to the CPS machine in Figure 5.

The C_aEK machine is a CEK machine specialized to the subset of Core Scheme in A-normal form (Figure 6). The machine (see Figure 8) has only two kinds of continuations: the continuation **stop**, and continuations of the form $\langle \mathbf{ar}\ x, M, E, K\rangle$. Unlike the CEK machine, the C_aEK machine only needs to build a continuation for the evaluation of a non-tail function call. For example, the transition rule for the tail call $(V\ V_1\ \ldots\ V_n)$ evaluates V to a closure $\langle \mathbf{cl}\ x_1 \ldots x_n, M', E'\rangle$, extends the environment E' with the values of $V_1, \ldots, V_n$ and continues with the execution of M'. The continuation component remains in the register K. By comparison, the CEK machine would build a seperate continuation for the evaluation of each sub-expression $V, V_1, \ldots, V_n$.

5 Equivalence of Compilation Strategies

A comparison of Figures 5 and 8 suggests a close relationship between the C_{cps}EK machine and the C_aEK machine. In fact, the two machines are identical modulo the *syntax* of the control strings, as corresponding state transitions on the two machines perform the same abstract operations. Currently, the transition rules for these machines are defined using pattern matching on the syntax of terms. Once we reformulate these rules using *predicates* and *selectors* for abstract syntax, we can see the correspondence more clearly.

For example, we can abstract the transition rules $\overset{(5)}{\longmapsto}_a$ and $\overset{(5)}{\longmapsto}_c$ from the term syntax as the higher-order functional $\mathcal{T}_5$:

$$
\begin{array}{l}
\mathcal{T}_5[\mathit{call\text{-}var}, \mathit{call\text{-}body}, \mathit{call?}, \mathit{call\text{-}args}, \mathit{call\text{-}fn}] = \\
\quad \langle C, E, K\rangle \longmapsto \cdots \qquad \text{if } \mathit{call?}(C) \\
\qquad\qquad \begin{array}{rrcl}
\text{where} & x & = & \mathit{call\text{-}var}(C) \\
 & M & = & \mathit{call\text{-}body}(C) \\
 & V & = & \mathit{call\text{-}fn}(C) \\
 & V_1, \ldots, V_n & = & \mathit{call\text{-}args}(C)
\end{array}
\end{array}
$$

The arguments to $\mathcal{T}_5$ are abstract-syntax functions for manipulating terms in a syntax-independent manner. Applying $\mathcal{T}_5$ to the appropriate functions produces either the transition rule $\overset{(5)}{\longmapsto}_a$ of the C_aEK machine or the rule $\overset{(5)}{\longmapsto}_c$ of the $\mathrm{C_{cps}}$EK machine, *i.e.*,

$$
\begin{array}{rcl}
\overset{(5)}{\longmapsto}_a & = & \mathcal{T}_5[\mathit{A\text{-}call\text{-}var}, \ldots, \mathit{A\text{-}call\text{-}fn}] \\
\overset{(5)}{\longmapsto}_c & = & \mathcal{T}_5[\mathit{cps\text{-}call\text{-}var}, \ldots, \mathit{cps\text{-}call\text{-}fn}]
\end{array}
$$

Suitable definitions of the syntax-functions for the language $A(CS)$ are:

$$
\begin{array}{rcl}
\mathit{A\text{-}call\text{-}var}[\![(\mathbf{let}\ (x\ (V\ V_1\ \ldots\ V_n))\ M)]\!] & = & x \\
\mathit{A\text{-}call\text{-}body}[\![(\mathbf{let}\ (x\ (V\ V_1\ \ldots\ V_n))\ M)]\!] & = & M \\
 & \cdots & \\
\mathit{A\text{-}call\text{-}fn}[\![(\mathbf{let}\ (x\ (V\ V_1\ \ldots\ V_n))\ M)]\!] & = & V
\end{array}
$$

Definitions for the language CPS(CS) follow a similar pattern:

$$
\begin{array}{rcl}
\mathit{cps\text{-}call\text{-}var}[\![(W\ (\lambda x.P)\ W_1\ \ldots\ W_n)]\!] & = & x \\
\mathit{cps\text{-}call\text{-}body}[\![(W\ (\lambda x.P)\ W_1\ \ldots\ W_n)]\!] & = & P \\
 & \cdots & \\
\mathit{cps\text{-}call\text{-}fn}[\![(W\ (\lambda x.P)\ W_1\ \ldots\ W_n)]\!] & = & W
\end{array}
$$

In the same manner, we can abstract each pair of transition rules $\overset{(n)}{\longmapsto}_a$ and $\overset{(n)}{\longmapsto}_c$ as a higher-order functional $\mathcal{T}_n$.

Let $\mathcal{S}_a$ and $\mathcal{S}_c$ be abstract-syntax functions appropriate for A-normal forms and CPS terms, respectively. Then the following theorem characterizes the relationship between the two transition functions.

Theorem 5.1 (Machine Equivalence) *For* $1 \leq n \leq 7$, $\overset{(n)}{\longmapsto}_a = \mathcal{T}_n[\mathcal{S}_a]$ *and* $\overset{(n)}{\longmapsto}_c = \mathcal{T}_n[\mathcal{S}_c]$.

The theorem states that the transition functions of the C_aEK and $\mathrm{C_{cps}}$EK machines are identical modulo syntax. However, in order to show that the evaluation of an A-normal form term M and its CPS counterpart on the respective machines produces exactly the same behavior, we also need to prove that there exists a bijection $\mathcal{M}$ between machine states that commutes with the transition rules.

Definition 5.2. ($\mathcal{M}$, $\mathcal{R}$, $\mathcal{V}$, *and* $\mathcal{K}$)

$$
\begin{array}{l}
\mathcal{M} : \mathit{State}_c \longrightarrow \mathit{State}_a \\
\quad \mathcal{M}(\langle P, E^-, E^k\rangle) = \langle \mathcal{U}[\![P]\!], \mathcal{R}(E^-), \mathcal{K}(E^k)\rangle \\[1em]
\mathcal{R} : \mathit{Env}_c \longrightarrow \mathit{Env}_a \\
\quad \mathcal{R}(E^-) = E \qquad \text{where } E(x) = \mathcal{V}(E^-(x)) \\[1em]
\mathcal{V} : \mathit{Value}_c \longrightarrow \mathit{Value}_a \\
\quad \mathcal{V}(c) = c \\
\quad \mathcal{V}(\langle \mathbf{cl}\ kx_1 \ldots x_n, P, E^-\rangle) = \\
\qquad \langle \mathbf{cl}\ x_1 \ldots x_n, \mathcal{U}[\![P]\!], \mathcal{R}(E^-)\rangle \\[1em]
\mathcal{K} : \mathit{Cont}_c \longrightarrow \mathit{Cont}_a \\
\quad \mathcal{K}(\mathbf{stop}) = \mathbf{stop} \\
\quad \mathcal{K}(\langle \mathbf{ar}\ x, P, E^-, E^k\rangle) = \\
\qquad \langle \mathbf{ar}\ x, \mathcal{U}[\![P]\!], \mathcal{R}(E^-), \mathcal{K}(E^k)\rangle
\end{array}
$$

∎

Intuitively, the function $\mathcal{M}$ maps $\mathrm{C_{cps}}$EK machine states to C_aEK machine states, and $\mathcal{R}$, $\mathcal{V}$ and $\mathcal{K}$ perform a similar mapping for environments, machine values and continuations respectively. We can now formalize the previously stated requirement that O and O' behave in the same manner.

Requirement For all $W_1^*, \ldots, W_n^* \in \mathit{Value}_c$,

$$\mathcal{V}(\delta_c(O', W_1^*, \ldots, W_n^*)) \ = \ \delta(O, \mathcal{V}(W_1^*), \ldots, \mathcal{V}(W_n^*)).$$

The function $\mathcal{M}$ commutes with the state transition functions.

Theorem 5.3 (Commutativity Theorem) *Let* $S \in \mathit{State}_c$*:* $S \overset{(n)}{\longmapsto}_c S'$ *if and only if* $\mathcal{M}(S) \overset{(n)}{\longmapsto}_a \mathcal{M}(S')$.

$$
\begin{tikzcd}
S \arrow[r, mapsto, "(n)", "c"'] \arrow[d, dashed, leftrightarrow, "\mathcal{M}"'] & S' \arrow[d, dashed, leftrightarrow, "\mathcal{M}"'] \\
\mathcal{M}(S) \arrow[r, mapsto, "(n)", "a"'] & \mathcal{M}(S')
\end{tikzcd}
$$

Proof: The inverse CPS transformations $\mathcal{U}$ is bijective [17]. Hence by structural induction, the functions $\mathcal{M}$, $\mathcal{R}$, $\mathcal{V}$ and $\mathcal{K}$ are also bijective. The proof proceeds by case analysis on the transition rules. ∎

Intuitively, the evaluation of a CPS term P on the $\mathrm{C_{cps}}$EK machine proceeds in the same fashion as the evaluation of $\mathcal{U}[\![P]\!]$ on the C_aEK machine. Together with the machine equivalence theorem, this implies that both machines perform the same sequence of abstract operations, and hence compilers based on these abstract machines can produce identical code for the same input. The A-normal form compiler achieves its goal in fewer passes.

6 A-Normal Forms as an Intermediate Language

Our analysis suggests that the language of A-normal forms is a good intermediate representation for compilers. Indeed, most direct compilers use transformations similar to the A-reductions on an *ad hoc* and incomplete basis. It is therefore natural to modify such compilers to perform a complete A-normalization phase, and analyze the effects. We have conducted such an experiment with the non-optimizing, direct compiler CAML Light [15]. This compiler translates ML programs into bytecode via a λ-calculus based intermediate language, and then interprets this bytecode. By performing A-normalization on the intermediate language and rewriting the interpreter as a C_aEK machine, we achieved speedups of between 50% and 100% for each of a dozen small benchmarks. Naturally, we expect the speedups to be smaller when modifying an optimizing compiler.

A major advantage of using a CPS-based intermediate representation is that many optimizations can be expressed as sequences of β and η reductions. For example, CPS compilers can transform the non-tail call $(W\ (\lambda x.kx)\ W_1\ \ldots\ W_n)$ to the tail-recursive call $(W\ k\ W_1\ \ldots\ W_n)$ using an η-reduction on the continuation [2]. An identical transformation [17] on the language of A-normal forms is the reduction β_{id}:

$$(\mathbf{let}\ (x\ (V\ V_1\ \ldots\ V_n))\ x) \longrightarrow (V\ V_1\ \ldots\ V_n),$$

where $V, V_1, \ldots, V_n$ are the A-normal forms corresponding to $W, W_1, \ldots, W_n$ respectively. Every other optimization on CPS terms that corresponds to a sequence of $\beta\eta$-reductions is also expressible on A-normal form terms [17].

The A-reductions also expose optimization opportunities by merging code segments across block declarations and conditionals. In particular, partial evaluators rely on the A-reductions to improve their specialization phase [5]. For example, the addition operation and the constant 0 are apparently unrelated in the following term:

$$(\mathit{add1}\ (\mathbf{let}\ (x\ (f\ 5))\ 0))$$

The A-normalization phase produces:

$$(\mathbf{let}\ (x\ (f\ 5))\ (\mathit{add1}\ 0)),$$

which specializes to $(\mathbf{let}\ (x\ (f\ 5))\ 1)$.

In summary, compilation with A-normal forms characterizes the critical aspects of the CPS transformation relevant to compilation. Moreover, it formulates these aspects in a way that direct compilers can easily use. Thus, our result should lead to improvements for both traditional compilation strategies.

A Linear A-Normalization

The linear A-normalization algorithm in Figure 9 is written in Scheme extended with a special form **match**, which performs pattern matching on the syntax of program terms. It employs a programming technique for CPS algorithms pioneered by Danvy and Filinski [9]. To prevent possible exponential growth in code size, the algorithm avoids duplicating the evaluation context enclosing a conditional expression. We assume the front-end uniquely renames all variables, which implies that the condition $x \notin FV(\mathcal{E})$ of the reduction A_1 holds.

Acknowledgments We thank Olivier Danvy, Preston Briggs, and Keith Cooper for comments on an early version of the paper.

References

[1] Aho, A., Sethi, R., and Ullman, J. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1985.

[2] Appel, A. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Barendregt, H. *The Lambda Calculus: Its Syntax and Semantics*, revised ed. Studies in Logic and the Foundations of Mathematics 103. North-Holland, 1984.

[4] Boehm, H.-J., and Demers, A. Implementing Russel. In *Proceedings of the ACM SIGPLAN 1986 Symposium on Compiler Construction* (1986), vol. 21(7), Sigplan Notices, pp. 186–195.

[5] Bondorf, A. Improving binding times without explicit CPS-conversion. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming* (1992), pp. 1–10.

[6] Clinger, W. The Scheme 311 compiler: An exercise in denotational semantics. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming* (1984), pp. 356–364.

[7] Danvy, O. Back to direct style. In *Proceedings of the 4th European Symposium on Programming* (Rennes, 1992), Lecture Notes in Computer Science, 582, Springer Verlag, pp. 130–150.

[8] Danvy, O. Three steps for the CPS transformation. Tech. Rep. CIS-92-2, Kansas State University, 1992.

[9] Danvy, O., and Filinski, A. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 4 (1992), 361–391.

```
(define normalize-term (lambda (M) (normalize M (lambda (x) x))))

(define normalize
  (lambda (M k)
    (match M
      [`(lambda ,params ,body) (k `(lambda ,params ,(normalize-term body)))]
      [`(let (,x ,M1) ,M2) (normalize M1 (lambda (N1) `(let (,x ,N1) ,(normalize M2 k))))]
      [`(if0 ,M1 ,M2 ,M3) (normalize-name M1 (lambda (t) (k `(if0 ,t ,(normalize-term M2) ,(normalize-term M3)))))]
      [`(,Fn . ,M*) (if (PrimOp? Fn)
                        (normalize-name* M* (lambda (t*) (k `(,Fn . ,t*))))
                        (normalize-name Fn (lambda (t) (normalize-name* M* (lambda (t*) (k `(,t . ,t*)))))))]
      [V (k V)])))

(define normalize-name
  (lambda (M k)
    (normalize M (lambda (N) (if (Value? N) (k N) (let ([t (newvar)]) `(let (,t ,N) ,(k t))))))))

(define normalize-name*
  (lambda (M* k)
    (if (null? M*)
        (k '())
        (normalize-name (car M*) (lambda (t) (normalize-name* (cdr M*) (lambda (t*) (k `(,t . ,t*)))))))))
```

Figure 9: A linear-time A-normalization algorithm

[10] Felleisen, M., and Friedman, D. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III* (Amsterdam, 1986), M. Wirsing, Ed., Elsevier Science Publishers B.V. (North-Holland), pp. 193–217.

[11] Fessenden, C., Clinger, W., Friedman, D. P., and Haynes, C. T. Scheme 311 version 4 reference manual. Computer Science Technical Report 137, Indiana University, Bloomington, Indiana, Feb. 1983.

[12] Fischer, M. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions About Programs* (1972), vol. 7(1), Sigplan Notices, pp. 104–109.

[13] Kelsey, R., and Hudak, P. Realistic compilation by program transformation. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages* (Austin, TX, Jan. 1989), pp. 281–292.

[14] Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN 1986 Symposium on Compiler Construction* (1986), vol. 21(7), Sigplan Notices, pp. 219–233.

[15] Leroy, X. The Zinc experiment: An economical implementation of the ML language. Tech. Rep. 117, INRIA, 1990.

[16] Plotkin, G. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science 1* (1975), 125–159.

[17] Sabry, A., and Felleisen, M. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming* (1992), pp. 288–298. Technical Report 92-180, Rice University.

[18] Shivers, O. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie-Mellon University, 1991.

[19] Steele, G. L. RABBIT: A compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[20] Wand, M. Correctness of procedure representations in higher-order assembly language. In *Proceedings of the 1991 Conference on the Mathematical Foundations of Programing Semantics* (1992), S. Brookes, Ed., vol. 598 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 294–311.

[21] Weise, D. Advanced compiling techniques. Course Notes at Stanford University, 1990.