Static and Dynamic Transition Compression (Preliminary version)*

Daniel Damian and Olivier Danvy

BRICS†

Department of Computer Science, University of Aarhus‡

May 21, 2001

Abstract

We specify operationally a translation from a structured to an unstructured imperative language, we point out how a compositional specification gives rise to static chains of jumps, and taking an inspiration from the notion of continuation, we state a new compositional specification that provably does not give rise to static chains of jumps. We also point out how a tail-conscious one-pass CPS transformation can avoid both static and dynamic chains of jumps. In SSA terms, these transformations naturally yield a reduced number of Φ-nodes for conditional commands and expressions, without context duplication.

1 Introduction

Part of the art of writing a compiler amounts to resolving the tensions between its various phases. Should a phase be simple, but generate redundancies that an equally simple later phase would eliminate? Or should each phase avoid redundancies in order to avoid a later phase and save the resources spent processing the redundancies in between? Picturesquely, Richard Gabriel nicknamed these canonical choices "worse is better" and "the right thing" [7]. Worse is better yields runnable code quicker, at the risk of never ending with the right thing. The right thing is the right thing, but takes longer, at the risk of funding running out. Resolving the tensions between these choices in a compiler is anything but easy considering its many circular dependencies.

In this article, we consider transition compression, i.e., collapsing chains of jumps into unique jumps. Section 2 addresses static transition compression. We formalize the translation of traditional imperative programs into labeled programs with jumps. We show how a compositional translation naturally generates

*Draft. Please do not distribute.

†Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

‡Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
E-mail:{damian,danvy}@brics.dk

$$
\begin{array}{rll}
u \in \text{UCom} & ::= & a \mid \textbf{skip} \mid u_0; u_1 \mid n : u \\
& & \mid \textbf{if } b \textbf{ then goto } n \mid \textbf{ifn } b \textbf{ then goto } n \mid \textbf{goto } n \\
a \in \text{ACom} & & \text{(atomic commands, omitted)} \\
b \in \text{Bexp} & & \text{(boolean expressions, omitted)} \\
l, n, m, r, k \in \mathbf{N} & & \text{(integer code labels)}
\end{array}
$$

Figure 1: A low-level unstructured target language

chains of jumps. We then specify a compositional translation that is parameterized by the label of the next instruction, and we prove that it generates no chain of jumps. The label of the next instruction is reminiscent of a continuation, and therefore, in Section 3, we consider the CPS transformation. We show how a naive CPS transformation gives rise to chains of continuations and how to generate no static chains of continuations (for conditional expressions) and no dynamic chains of continuations (for tail calls).

The issue of transition compression is traditional [1], but we are not aware of any other formalized characterization of generating chains of jumps and avoiding them. The issue of tail calls in the CPS transformation is also known [5], and the issue of static chain compression is a simple corollary.

2 Static transition compression

We consider the problem of translating traditional structured programs that use conditionals statements and while loops into low-level unstructured programs that use labels and jumps.

We show how a direct, compositional translation yields chains of jumps. We thus parameterize the translation with the label of the next instruction and we specify a compositional translation that provably does not generate chains of jumps.

2.1 A low-level unstructured target language

Our target language is defined in Figure 1. Unstructured commands can be atomic commands, skip commands, sequencing of commands and conditional or unconditional jumps. Commands can be labeled, and labels are used as targets of jumps. We consider two conditional jumps, which take place on either the truth (**if**) or the falsity (**ifn**) of their conditional expression.

The semantics of the language is state-based and straightforward. Valid programs are terms in which labels are unique and in which target labels of jumps always exist.

$$
\begin{array}{lll}
s \in \text{SCom} & ::= & a \mid s_0;\ s_1 \mid \textbf{if } b \textbf{ then } s \textbf{ else } s \textbf{ end} \mid \textbf{while } b \textbf{ do } s \textbf{ end} \\
a \in \text{ACom} & & \text{(atomic commands, same as in Figure 1)} \\
b \in \text{Bexp} & & \text{(boolean expressions, same as in Figure 1)}
\end{array}
$$

Figure 2: A high-level structured source language

2.2 A high-level structured source language

Our source language is defined in Figure 2. Structured commands can be atomic commands, sequencing of commands, conditional statements and while loops. Again, the semantics of the source language is state-based and straightforward.

Implementing a compiler for such language would require one to translate structured commands into a low-level language such as the one in Figure 1. This translation is the topic of the next section.

2.3 A direct translation

We start with a direct, compositional translation. For each construct, the translation generates code for all subcomponents, and combines them together with explicit conditional/unconditional jumps implementing the required flow of control.

We define the translation operationally in Figure 3. The judgment

$$\vdash s \longrightarrow_n^m u$$

holds whenever the structured command s translates into the unstructured command u, where generated labels range from n exclusively up to m inclusively. Given a program $p \in \text{SCom}$, by definition, p' is a translation of p if there exists an n such that $\vdash p \longrightarrow_0^n p'$ holds.

In Figure 3, the translation judgments are deterministic and therefore the translation of a program always exists and is unique. Using simple structural induction we can show that the semantics of a translated program agrees with the semantics of the source program.

As a matter of fact, the translation defined in Figure 3 is just one among many in the design space for the translation of while-loops and conditionals. For instance, the following alternative translation of while-loops would yield more compact programs:

$$
\frac{\vdash s \longrightarrow_{n+2}^m u}{\vdash \textbf{while } b \textbf{ do } s \textbf{ end} \longrightarrow_n^m \begin{array}{l} \quad\quad\ \ \textbf{goto } n+2; \\ n+1 : u; \\ n+2 : \textbf{if } b \textbf{ then goto } n+1 \end{array}}
$$

The two translations differ in the number of steps executed at run-time by the resulting program, but they otherwise yield semantically equivalent programs.

$$\frac{}{\vdash a \longrightarrow_n^n a} \qquad \frac{\vdash s_0 \longrightarrow_n^m u_0 \qquad \vdash s_1 \longrightarrow_m^r u_1}{\vdash s_0;\, s_1 \longrightarrow_n^r u_0;\, u_1}$$

$$\frac{\vdash s_0 \longrightarrow_{n+2}^m u_0 \quad \vdash s_1 \longrightarrow_m^r u_1}{\begin{array}{rl}\vdash \textbf{if } b \textbf{ then } s_0 \textbf{ else } s_1 \textbf{ end} \longrightarrow_n^r & \textbf{if } b \textbf{ then goto } n+1;\\ & u_1;\, \textbf{goto } n+2;\\ n+1: & u_0;\\ n+2: & \textbf{skip}\end{array}}$$

$$\frac{\vdash s \longrightarrow_{n+2}^m u}{\begin{array}{rl}\vdash \textbf{while } b \textbf{ do } s \textbf{ end} \longrightarrow_n^m n+1: & \textbf{ifn } b \textbf{ then goto } n+2;\\ & u;\, \textbf{goto } n+1;\\ n+2: & \textbf{skip}\end{array}}$$

Figure 3: A direct translation

2.4 Analysis and assessment

A compositional translation is simple to write and to reason about, but, by its very nature, it does not take into account the context of each translated term. Therefore, it gives rise to chains of jumps, e.g., for nested conditionals, as illustrated by the following example:

$$\begin{array}{llrl}
\vdash \textbf{if } b_1 & \longrightarrow_0^4 & & \textbf{if } b_1 \textbf{ then goto } 1;\\
\quad \textbf{then } a_1 & & & \textbf{if } b_2 \textbf{ then goto } 3;\\
\quad \textbf{else if } b_2 & & & a_3;\, \textbf{goto } 4;\\
\qquad \textbf{then } a_2 & & 3: & a_2;\\
\qquad \textbf{else } a_3 & & 4: & \textbf{skip};\, \textbf{goto } 2;\\
\qquad \textbf{end} & & 1: & a_1;\\
\quad \textbf{end} & & 2: & \textbf{skip}
\end{array}$$

The example illustrates how nested conditionals give rise to chains of **skip** and unconditional jumps (here, label 4). Given n nested conditionals, the direct translation generates a chain of n successive jumps. Furthermore, it gives rise to spurious **skip** instructions, as illustrated by the following translation of nested conditionals:

$$\begin{array}{llrl}
\vdash \textbf{if } b_1 & \longrightarrow_0^4 & & \textbf{if } b_1 \textbf{ then goto } 1;\\
\quad \textbf{then if } b_2 & & & a_3;\, \textbf{goto } 2;\\
\qquad \textbf{then } a_1 & & 1: & \textbf{if } b_2 \textbf{ then goto } 3;\\
\qquad \textbf{else } a_2 & & & a_2;\, \textbf{goto } 4;\\
\qquad \textbf{end} & & 3: & a_1;\\
\quad \textbf{else } a_3 & & 4: & \textbf{skip};\\
\quad \textbf{end} & & 2: & \textbf{skip}
\end{array}$$

Similar redundancies arise when translating of nested while-loops, as in the following example:

$$\begin{array}{llll} \vdash & \textbf{while } b_1 & \longrightarrow_0^4 & 1 : \textbf{ifn } b_1 \textbf{ then goto } 2; \\ & \textbf{do while } b_2 & & 3 : \textbf{ifn } b_2 \textbf{ then goto } 4; \\ & \quad \textbf{do } a & & \quad a; \textbf{goto } 3; \\ & \quad \textbf{end} & & 4 : \textbf{skip}; \textbf{goto } 1; \\ & \textbf{end} & & 2 : \textbf{skip} \end{array}$$

The transition to label 4 is redundant. It could be compressed into a transition to label 1. Alternative translations for while-loops share the same problem.

2.5 Chains of jumps

Chains of jumps are a classical issue. Compiler textbooks list two choices: (1) have a simple code generation scheme and eliminate chains of jumps in a post-processing phase (e.g., peephole optimization [1, Chapter 9]); (2) use backpatching [1, Chapter 8]. Post-unfolding chains of jumps is also known as transition compression in partial evaluation [8, Section 4.4].

On one hand, a simple code generation scheme is easier to maintain and to extend to various target platforms. On the other hand, a post-processing phase is likely to be an expensive part of the compiler, since it requires one to identify the chains of jumps. As for backpatching, it requires one to maintain an additional list structure in the memory, potentially linear in the size of the source program. Consequently, ensuring correctness becomes difficult.

Other authors [2] propose using code blocks, exploring code traces and reshuffling blocks in order to minimize the amount of redundant jumps. On one hand, such approach might lead to better code by optimizing the redundancies in the original code. On the other hand, it requires one to design and implement a code-reshuffling strategy.

In the following sections we present and prove a simple translation scheme that avoids generating chains of jumps, does not introduce additional computational steps and does not require allocation of additional space. Furthermore, the generated code is still amenable to code reshuffling.

2.6 Towards avoiding chains of jumps

Examining Figure 3 makes it clear where chain of jumps arise. For example, a conditional is always punctuated with a labeled skip instruction, to establish a program point where the conditional branches can join. Therefore, nesting conditional expressions gives rise to chains of jumps—one for each joining program point.

In the spirit of continuations [12], we parameterize the translation with the label of the next instruction. This label makes it possible to short-cut chains of jumps for conditional expressions. The parameterized translation is displayed in Figure 4. The judgment

$$\vdash p \;\&\; 0 \Longrightarrow_0^n u$$

$$\frac{}{\vdash (a)\ \&\ k \Longrightarrow_n^n a;\ \textbf{goto}\ k} \qquad \frac{\vdash s_0\ \&\ n+1 \Longrightarrow_{n+1}^m u_0 \quad \vdash s_1\ \&\ k \Longrightarrow_m^r u_1}{\vdash (s_0;\ s_1)\ \&\ k \Longrightarrow_n^r u_0;\ n+1 : u_1}$$

$$\frac{\vdash s_0\ \&\ k \Longrightarrow_{n+1}^m u_0 \quad \vdash s_1\ \&\ k \Longrightarrow_m^r u_1}{\vdash (\textbf{if}\ b\ \textbf{then}\ s_0\ \textbf{else}\ s_1\ \textbf{end})\ \&\ k \Longrightarrow_n^r \begin{array}{l} \quad\ \textbf{if}\ b\ \textbf{then goto}\ n+1; \\ \quad\ u_1; \\ n+1 : u_0 \end{array}}$$

$$\frac{\vdash s\ \&\ n+1 \Longrightarrow_{n+1}^m u}{\vdash (\textbf{while}\ b\ \textbf{do}\ s\ \textbf{end})\ \&\ k \Longrightarrow_n^m n+1 : \textbf{ifn}\ b\ \textbf{then goto}\ k;\ u}$$

Figure 4: Continuation-based translation

is satisfied whenever the program $p' = (u;\ 0 : \textbf{skip})$ is the translation of p, and where generated labels range from 0 exclusively up to n inclusively. Again, the translation judgments are deterministic and therefore the translation of a program always exists and is unique.

Compared to the direct translation from Figure 3, the continuation-based translation avoids generating chains of jumps, as illustrated by revisiting the same example of nested conditionals:

$$\begin{array}{lll} \vdash \textbf{if}\ b_1 & \&\ 0 \Longrightarrow_0^4 & \quad\ \textbf{if}\ b_1\ \textbf{then goto}\ 1; \\ \quad \textbf{then}\ a_1 & & \quad\ \textbf{if}\ b_2\ \textbf{then goto}\ 3; \\ \quad \textbf{else if}\ b_2 & & \quad\ a_3;\ \textbf{goto}\ 0; \\ \quad\quad \textbf{then}\ a_2 & & 3 : a_2;\ \textbf{goto}\ 0; \\ \quad\quad \textbf{else}\ a_3 & & 1 : a_1;\ \textbf{goto}\ 0; \\ \quad\quad \textbf{end} & & 0 : \textbf{skip} \\ \quad \textbf{end} & & \end{array}$$

The translation from Figure 4 is not optimal though. In the example above the last jump is redundant, it jumps to the next instruction. Moreover, the translation of sequencing of atomic commands inserts useless jumps:

$$\begin{array}{ll} \vdash (a_1;\ a_2)\ \&\ 0 \Longrightarrow_0^1 & \quad\ a_1;\ \textbf{goto}\ 1; \\ & 1 : a_2;\ \textbf{goto}\ 0; \\ & 0 : \textbf{skip} \end{array}$$

The problem resembles the problem of administrative redexes introduced by a Plotkin-style CPS transformation [5, 10] on higher-order languages: at each value-term, we must call the continuation with the given value. In the next section we present a translation which avoids generating useless jumps.

2.7 Avoiding chains of jumps

We duplicate the definition of the continuation-based translation to account for the cases where the continuation of the translation of a command is the next command. The idea of duplicating the translation is similar to Danvy and Filinski's duplication of the CPS transformation [5] which yields a properly tail-recursive one-pass CPS transformation which does not introduce administrative redexes.

In contrast to the translation from Figure 4, our new translation places labels on each atomic command and on each conditional jump. It does not insert labels on any unconditional jumps, however. Therefore, by construction, it does not generate any chain of jumps.

The translation rules are defined in Figure 5. The judgment

$$l \vdash s \ \&\ k \longrightarrow_n^m u$$

holds whenever u is the translation of s with starting label l, and where labels range from n exclusively up to m inclusively. The translation judgment is used when the translation is immediately sequenced with its continuation. The judgment

$$l \vdash s \ \&\ k \Longrightarrow_n^m u$$

holds whenever u is the translation of s with starting label l, and where labels range from n exclusively up to m inclusively. The judgment is used when the continuation is not immediately sequenced, in which case explicit jumps are generated, as seen in Figure 5.

By definition, the translation of a program p is the term p'; $0 : \textbf{skip}$ such that there exists an n such that $l \vdash p \ \&\ 0 \Longrightarrow_0^n p'$ holds. Again, the translation judgments are deterministic and therefore there the translation of a program always exist and is unique. Using simple structural induction we can show as well that the semantics of a translated program agrees with the semantics of the source program.

The translation of Figure 5 does not introduce spurious **skip** commands. Moreover, none of the generated unconditional jumps are labeled. Therefore, the following theorem stating the chain-free property is immediate to prove.

Theorem 2.1. *Let p be a program in* SCom *and p' be its translation, i.e., such that $p' = u$; $0 : \textbf{skip}$ and $1 \vdash p \ \&\ 0 \longrightarrow_1^n u$ holds. Then,*

1. *There exist no subterms* **goto** n *and* $n :$ **goto** m *in* p'.

2. *There exist no subterms* **ifn** b **then goto** n *and* $n :$ **goto** m *in* p'.

3. *There exist no subterms* **if** b **then goto** n *and* $n :$ **goto** m *in* p'.

Let us observe now the result of the chain-free translation. Consider the translation of the nested while example of Section 2.3:

$$\begin{array}{lll} 1 \vdash \textbf{while}\ b_1 & \&\ 0 \longrightarrow_1^3 & 1 : \textbf{ifn}\ b_1\ \textbf{then goto}\ 0; \\ \quad \textbf{do while}\ b_2 & & 2 : \textbf{ifn}\ b_2\ \textbf{then goto}\ 1; \\ \qquad \textbf{do}\ a & & 3 : a;\ \textbf{goto}\ 2; \\ \qquad \textbf{end} & & 0 : \textbf{skip} \\ \quad \textbf{end} & & \end{array}$$

$$\frac{}{l \vdash a \ \&\ k \longrightarrow_n^n l : a}$$

$$\frac{l \vdash s_0 \ \&\ n+1 \longrightarrow_{n+1}^m u_0 \qquad n+1 \vdash s_1 \ \&\ k \longrightarrow_m^r u_1}{l \vdash (s_0;\ s_1) \ \&\ k \longrightarrow_n^r u_0;\ u_1}$$

$$\frac{n+1 \vdash s_0 \ \&\ k \longrightarrow_{n+2}^m u_0 \quad n+2 \vdash s_1 \ \&\ k \Longrightarrow_m^r u_1}{l \vdash (\textbf{if}\ b\ \textbf{then}\ s_0\ \textbf{else}\ s_1\ \textbf{end}) \ \&\ k \longrightarrow_n^r \begin{array}[t]{l} l : \textbf{if}\ b\ \textbf{then goto}\ n+1; \\ u_1; \\ u_0 \end{array}}$$

$$\frac{n+1 \vdash s \ \&\ l \Longrightarrow_{n+1}^m u}{l \vdash (\textbf{while}\ b\ \textbf{do}\ s\ \textbf{end}) \ \&\ k \longrightarrow_n^m l : \textbf{ifn}\ b\ \textbf{then goto}\ k;\ u}$$

$$\frac{}{l \vdash a \ \&\ k \Longrightarrow_n^n l : a;\ \textbf{goto}\ k}$$

$$\frac{l \vdash s_0 \ \&\ n+1 \longrightarrow_{n+1}^m u_0 \qquad n+1 \vdash s_1 \ \&\ k \Longrightarrow_m^r u_1}{l \vdash (s_0;\ s_1) \ \&\ k \Longrightarrow_n^r u_0;\ u_1}$$

$$\frac{n+1 \vdash s_0 \ \&\ k \Longrightarrow_{n+2}^m u_0 \quad n+2 \vdash s_1 \ \&\ k \Longrightarrow_m^r u_1}{l \vdash (\textbf{if}\ b\ \textbf{then}\ s_0\ \textbf{else}\ s_1\ \textbf{end}) \ \&\ k \Longrightarrow_n^r \begin{array}[t]{l} l : \textbf{if}\ b\ \textbf{then goto}\ n+1; \\ u_1; \\ u_0 \end{array}}$$

$$\frac{n+1 \vdash s \ \&\ l \Longrightarrow_{n+1}^m u}{l \vdash (\textbf{while}\ b\ \textbf{do}\ s\ \textbf{end}) \ \&\ k \Longrightarrow_n^m l : \textbf{ifn}\ b\ \textbf{then goto}\ k;\ u}$$

Figure 5: Chain-free continuation-based translation

(Note that the terminating $0 : \textbf{skip}$ is not part of the translation judgment. It is added at the outset). The generated code is more compact than the result of the naive compositional translation.

As one can also observe in this example, the translation of Figure 5 may generate unused labels for either atomic commands or for conditional jumps. The unused labels are not a concern, however, for the run-time behavior of the compiled program.

2.8 Conclusions

We have presented a translation from an imperative language with structured control-flow constructs into a low-level language with unstructured control-flow constructs. The translation is compositional and operates in linear time and space on the size of the input program. The resulting program is semantically equivalent to the original program, contains the same atomic commands and does not contain chains of jumps.

3 Static and dynamic transition compression

We briefly review the issue of transition compression in the CPS transformation [5, 11]. Figure 6 displays the usual syntax of the λ-calculus in direct style, including conditional expressions. Figure 7 displays the corresponding syntax of the λ-calculus in continuation-passing style, also including conditional expressions. This grammar distinguishes between calls that were in tail position in direct style (their continuation is k) and calls that were not in tail position in direct style (their continuation is $\lambda v.e$). It also distinguishes between conditional expressions that were in tail position in direct style and conditional expressions that were not in tail position in direct style (their continuation is named). The notion of tail position is traditional [4, 6].

In a CPS-based compiler, lambdas are implemented as labels, and calls are implemented as jumps that pass arguments [11]. Therefore, a spurious eta-redex such as $\lambda v.k\ v$ is part of a chain of jumps.

Distinguishing between tail-calls and non-tail calls makes it possible to avoid spurious eta-redexes in CPS programs. These eta-redexes give rise to dynamic chains of jumps when returning from tail calls.

Naming the continuation of conditional expressions is a must to avoid code duplication. But for conditional expressions in tail position, it gives rise to spurious eta-redexes in CPS programs. These eta-redexes give rise to static chains of jumps when “returning” from nested conditional expressions.

Danvy and Filinski’s one-pass CPS transformation distinguishes between terms in tail position ($\mathcal{T}$ in Figure 8) and terms in non-tail position ($\mathcal{C}$ in Figure 8). Therefore, it yields administratively reduced terms without spurious eta-redexes for continuations, and therefore without static and dynamic chains of jumps.

4 Conclusions

Compiling nested conditional commands and expressions naturally gives rise to chains of jumps to join their control flow. We have pointed out how a to avoid generating these chains of jumps, both for an imperative language (Section 2) and for a functional language (Section 3), through the ubiquitous CPS transformation.

CPS, and more generally functional programming, have been connected to SSA [3, 9, 13]. In that light, our two transformations, seen as SSA transformations, naturally yield a reduced number of Φ-nodes for conditional commands and expressions, without context duplication.

$$
\begin{array}{lll}
r \in \text{DRoot} & \text{— terms} & r ::= e \\
e \in \text{DExp} & \text{— expressions} & e ::= e_0\, e_1 \mid \text{if } t \text{ then } e_1 \text{ else } e_2 \mid t \\
t \in \text{DTriv} & \text{— trivial expressions} & t ::= x \mid \lambda x.r \\
x \in \text{Ident} & \text{— identifiers} &
\end{array}
$$

Figure 6: BNF of the direct-style λ-calculus with conditional expressions

$$
\begin{array}{llll}
r \in \text{CRoot} & \text{— terms} & r ::= \lambda k.e & \\
e \in \text{CExp} & \text{— expressions} & e ::= k\, t & \\
t \in \text{CTriv} & \text{— trivial expressions} & \quad \mid & t_0\, t_1\, k \\
x \in \text{Ident} & \text{— source identifiers} & \quad \mid & t_0\, t_1\, \lambda v.e \\
k \in \text{Cont} & \text{— cont. identifiers} & \quad \mid & \text{if } t \text{ then } e_1 \text{ else } e_2 \\
v \in \text{Var} & \text{— parameters of cont.} & \quad \mid & (\lambda k.\text{if } t \text{ then } e_1 \text{ else } e_2)\, (\lambda v.e) \\
 & & t ::= x \mid \lambda x.r \mid v &
\end{array}
$$

Figure 7: BNF of the CPS λ-calculus with conditional expressions

$$
\begin{array}{rcl}
\mathcal{C}[\![r]\!] & : & \text{CRoot} \\
\mathcal{C}[\![e]\!] & = & \underline{\lambda} k.\mathcal{T}[\![e]\!]k \\
\\
\mathcal{T}[\![e]\!] & : & \text{Cont} \rightarrow \text{CExp} \\
\mathcal{T}[\![e_0\, e_1]\!] & = & \overline{\lambda} k.\mathcal{C}[\![e_0]\!](\overline{\lambda} t_0.\mathcal{C}[\![e_1]\!](\overline{\lambda} t_1.(t_0 \,\underline{@}\, t_1)\,\underline{@}\, k)) \\
\mathcal{T}[\![\text{if } e_0 \text{ then } e_1 \text{ else } e_2]\!] & = & \overline{\lambda} k.\mathcal{C}[\![e_0]\!](\overline{\lambda} t_0.\underline{\text{if}}\ t_0\ \underline{\text{then}}\ \mathcal{T}[\![e_1]\!]k\ \underline{\text{else}}\ \mathcal{T}[\![e_2]\!]k) \\
\mathcal{T}[\![t]\!] & = & \overline{\lambda} k.k \,\underline{@}\, (\mathcal{C}[\![t]\!]) \\
\\
\mathcal{C}[\![e]\!] & : & (\text{CTriv} \rightarrow \text{CExp}) \rightarrow \text{CExp} \\
\mathcal{C}[\![e_0\, e_1]\!] & = & \overline{\lambda} \kappa.\mathcal{C}[\![e_0]\!](\overline{\lambda} t_0.\mathcal{C}[\![e_1]\!](\overline{\lambda} t_1.(t_0 \,\underline{@}\, t_1)\,\underline{@}\,(\underline{\lambda} v.\kappa \,\overline{@}\, v))) \\
\mathcal{C}[\![\text{if } e_0 \text{ then } e_1 \text{ else } e_2]\!] & = & \overline{\lambda} \kappa.\mathcal{C}[\![e_0]\!](\overline{\lambda} t_0.(\underline{\lambda} k.\underline{\text{if}}\ t_0\ \underline{\text{then}}\ \mathcal{T}[\![e_1]\!]k\ \underline{\text{else}}\ \mathcal{C}[\![e_2]\!]k) \\
 & & \qquad\qquad\quad \underline{@}\,(\underline{\lambda} v.\kappa \,\overline{@}\, v)) \\
\mathcal{C}[\![t]\!] & = & \overline{\lambda} \kappa.\kappa \,\overline{@}\, (\mathcal{C}[\![t]\!]) \\
\\
\mathcal{C}[\![t]\!] & : & \text{CTriv} \\
\mathcal{C}[\![x]\!] & = & x \\
\mathcal{C}[\![\lambda x.r]\!] & = & \underline{\lambda} x.\mathcal{C}[\![r]\!]
\end{array}
$$

where k and the v's are fresh variables.

Figure 8: CPS transformation

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in* {*C, Java, ML*}. Cambridge University Press, New York, 1998.

[3] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, April 1998.

[4] William D. Clinger. Proper tail recursion and space efficiency. In Keith D. Cooper, editor, *Proceedings of the ACM SIGPLAN'98 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998.

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

[6] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, second edition.* The MIT Press, 2001.

[7] Richard P. Gabriel. LISP: Good news, bad news, how to win big. *AI Expert*, 6(6):30–39, June 1991.

[8] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

[9] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In Michael Ernst, editor, *ACM SIGPLAN Workshop on Intermediate Representations*, SIGPLAN Notices, Vol. 30, No 3, pages 13–22, San Francisco, California, January 1995.

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

[11] 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.

[12] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2), 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974).

[13] Mark N. Wegman and F. Ken Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 3(2):181–210, 1991.