A Correspondence between Continuation Passing Style and Static Single Assignment Form

Richard A. Kelsey
NEC Research Institute
kelsey@research.nj.nec.com

Abstract

We define syntactic transformations that convert continuation passing style (CPS) programs into static single assignment form (SSA) and vice versa. Some CPS programs cannot be converted to SSA, but these are not produced by the usual CPS transformation. The CPS→SSA transformation is especially helpful for compiling functional programs. Many optimizations that normally require flow analysis can be performed directly on functional CPS programs by viewing them as SSA programs. We also present a simple program transformation that merges CPS procedures together and by doing so greatly increases the scope of the SSA flow information. This transformation is useful for analyzing loops expressed as recursive procedures.

IR'95-1/95 San Francisco, California USA (proceedings in ACM SIGPLAN Notices 30(3), March 1995)

1 Introduction

Continuation-passing style has been used as an intermediate language in a number of compilers for functional languages [1, 8, 12]. Static single assignment form has been used in optimizations targeted towards imperative languages, for example eliminating induction variables [13] and partial redundancies [2]. In this paper we define syntactic transformations for converting continuation passing style (CPS) programs into static single assignment form (SSA) and vice versa.

The similarities between CPS and SSA have been noted by others [1, 9]. In CPS there is exactly one binding form for every variable and variable uses are lexically scoped. In SSA there is exactly one assignment statement for every variable, and that statement dominates all uses of the variable. This is also the main difference between the two: the restriction on variable references in CPS is lexical, while in SSA it is dynamic.

The two forms have generally been used in very different contexts. CPS has been used in compilers for functional languages, and SSA for imperative ones. As a result, the problem of flow analysis has come to be viewed as more difficult in CPS. This is really an artifact of the programs being compiled and not a problem with the CPS intermediate language. Functional programs express control flow through the use of procedures, resulting in large collections of small procedures, as opposed to small collections of large procedures in imperative languages (by 'large' procedure we mean one large enough to contain a loop). The writers of compilers for imperative languages have been quite successful with using SSA to express the results of intraprocedural flow-analysis and then analyzing the SSA program. CPS uses procedures to express practically everything, so anything but the most local optimization appears to require interprocedural analysis, which is hard in any language.

In this paper we are not going to concern ourselves with interprocedural flow analysis. What we will do is restrict our notion of what constitutes a procedure in CPS. The λ forms in CPS programs will be annotated to indicate which represent full procedures and which are continuations. This reduces the number of full procedures and greatly simplifies analyzing the program.

Throughout the paper we will assume that any use of lexical scoping in the source program has been implemented by the introducing explicit environments, as described in [1, 7]. We further assume that in the CPS programs continuations are created and used in a last-in/first-out manner (see section 6 for a discussion). The latter restriction only affects the way in which nonlocal returns, such as longjumps in C or call-with-current-continuation in Scheme, are expressed in CPS.

The paper proceeds as follows. Section 2 contains the definition of an SSA language. Section 3 defines a source language, an annotated CPS language, and an algorithm for converting source programs into CPS programs. The following two sections define functions that convert CPS procedures into SSA procedures and vice versa. The remainder of the paper is a discussion of practical differences between SSA and CPS, followed by a comparison to previous work.

2 Static Single Assignment

SSA is an imperative form in which there is exactly one assignment for every variable and that assignment dominates all uses of the variable (see [4] for a good overview of SSA).

To make the control-flow graph explicit, control flow is expressed entirely in terms of `if` and `goto`. We allow expressions in some nonstandard places, for example as the arguments to the ϕ-functions; removing these only requires introducing a few additional assignment statements. The syntax of expressions does not matter and is left unspecified, with the restriction that they may not contain procedure calls (and they typically don't in SSA languages).

The grammar for a procedure in our SSA language is:

$$
\begin{array}{lll}
\mathrm{P} & ::= & \texttt{proc(}x^*\texttt{)}\ \{\ \mathrm{B}\ \mathrm{L}^*\ \} \\
\mathrm{L} & ::= & l : \mathrm{I}^*\ \mathrm{B} \\
\mathrm{I} & ::= & x \leftarrow \phi\texttt{(}\mathrm{E}^*\texttt{);} \\
\mathrm{B} & ::= & x \leftarrow \mathrm{E}\texttt{;}\ \mathrm{B} \mid x \leftarrow \mathrm{E}\texttt{(}\mathrm{E}^*\texttt{);}\ \mathrm{B} \mid \\
 & & \texttt{goto}\ l_i\texttt{;} \mid \\
 & & \texttt{return}\ \mathrm{E}\texttt{;} \mid \texttt{return}\ \mathrm{E}\texttt{(}\mathrm{E}^*\texttt{);} \mid \\
 & & \texttt{if}\ \mathrm{E}\ \texttt{then}\ \mathrm{B}\ \texttt{else}\ \mathrm{B} \\
\mathrm{E} & ::= & x \mid \mathrm{E+}\ \mathrm{E} \mid \ldots \\
\text{where} & & x \in \text{variables} \\
 & & l \in \text{labels}
\end{array}
$$

The semantics is the 'obvious' one. $\leftarrow$ is assignment, E(E, ...) is a procedure call, and `return` returns from the current procedure. The ϕ-functions at the beginning of a block each take one argument for each of the `goto`'s that jump to that block. The i'th argument is returned when control reaches the block from `goto` l_i.

Cytron et al. [4] describe a translation algorithm that efficiently converts programs into SSA while introducing the minimum number of ϕ-functions.

Because every variable has a unique assignment, definition↔use chains are trivial to compute. Many analyses and optimizations are simpler when applied to a program in SSA form than to the original source program.

Below is an example SSA procedure that counts the number of times zero appears in the sequence `f(0)` ... `f(limit-1)` for some function `f` and integer `limit`. Note that some of the computation occurs in the ϕ-functions. In particular the second `if` makes sense only in the context of the ϕ-function for `c'` at label `j`. We will be using this example throughout the paper.

```
proc (f limit) {
     goto l0;
   l:i ←φ(0, i+1);
     c ←φ(0, c');
     if i = limit then
        return c;
     else
        x ←f(i);
        if x = 0 then
           goto j0;
        else
           goto j1; }
   j:c' ←φ(c+1, c);
     goto l1; }
```

3 Annotated CPS

In continuation passing style procedures do not return. Instead they are passed an additional argument, a continuation, which is applied to the procedure's return value (see chapter 8 of [5] for a full discussion).

The correspondence between CPS and SSA requires a slightly modified algorithm for converting programs into CPS. The modified algorithm annotates the λ forms in the CPS code to show how they are used. The annotations have no semantic content and, if not introduced by the CPS algorithm, could be added to the program via some suitable analysis (assuming the CPS program is in a suitable form; see section 6 below).

Our source language is a subset of Scheme [3], with the subset chosen as a compromise between simplicity and realism. For simplicity we will assume that the source and CPS languages use the same expressions as in the SSA language of the previous section. In the context of Scheme programs we will refer to the equivalents of the SSA expressions as trivial expressions, to keep from confusing them with other Scheme expressions. As described above, the main restriction on (trivial) expressions is that they cannot contain procedure calls.

To simplify the CPS algorithm the source language is restricted to allow non-trivial expressions only in tail position or as the bound value in a `let`. In an actual compiler the source program could be put in this form either by a pre-pass or as part of a more complex CPS algorithm. We also assume that every identifier is unique.

A `loop` expression is a version of Scheme's named-`let` with the restriction that calls to the label may only occur in tail position. It is included to show how iterative constructs, such as `for` and `while` loops, may be converted into CPS.

The grammar for this Scheme subset is:

$$
\begin{array}{lll}
\mathrm{M} & ::= & \mathrm{E} \mid (\mathrm{E}\ \mathrm{E}^*) \mid (\mathtt{if}\ \mathrm{E}\ \mathrm{M}\ \mathrm{M}) \mid \\
 & & (\mathtt{let}\ ((x\ \mathrm{M}))\ \mathrm{M}) \mid \\
 & & (\mathtt{loop}\ l\ ((x\ \mathrm{E})^*)\ \mathrm{M}) \mid \\
 & & (l\ \mathrm{E}^*) \\
\mathrm{E} & ::= & x \mid (+\ \mathrm{E}\ \mathrm{E}) \mid \ldots \\
\mathrm{P} & ::= & (\lambda\ (x^*)\ \mathrm{M}) \\
\text{where} & & x \in \text{variables} \\
 & & l \in \text{labels}
\end{array}
$$

The semantics of the source language is that of Scheme. This subset is sufficient to implement most of Scheme, with explicit cells added for any variables that are the targets of `set!` expressions in the source program.

As was done in the Rabbit compiler [12] our

$$
\begin{array}{l}
\mathcal{F} : \mathrm{M} \times \mathrm{C} \rightarrow \mathrm{M}' \\
\mathcal{F}([\![\mathrm{E},\, k]\!]) = \texttt{(}k\ \mathrm{E}\texttt{)} \\
\mathcal{F}([\![\mathrm{E},\, \texttt{(}\lambda_{cont}\texttt{(}x\texttt{)}\ \mathrm{M}'\texttt{)}]\!]) = \texttt{(let ((}x\ \mathrm{E}\texttt{))}\ \mathrm{M}'\texttt{)} \\
\mathcal{F}([\![\texttt{(}\mathrm{E}\ \ldots\texttt{)},\, \mathrm{C}]\!]) = \texttt{(let ((}v\ \texttt{(}\mathrm{E}\ \ldots\texttt{)))}\ \texttt{(}j\ v\texttt{))} \text{ if } \mathrm{C} = j \text{ and } j \text{ is bound by } \texttt{letrec} \\
\qquad\qquad\qquad\quad = \texttt{(}\mathrm{E}\ \ldots\ \mathrm{C}\texttt{)} \qquad\qquad\qquad \text{otherwise} \\
\mathcal{F}([\![\texttt{(let ((}x\ \mathrm{M}_1\texttt{))}\ \mathrm{M}_2\texttt{)},\, \mathrm{C}]\!]) = \mathcal{F}([\![\mathrm{M}_1,\, \texttt{(}\lambda_{cont}\texttt{(}x\texttt{)}\ \mathcal{F}([\![\mathrm{M}_2,\, \mathrm{C}]\!])\texttt{)}]\!]) \\
\mathcal{F}([\![\texttt{(if}\ \mathrm{E}\ \mathrm{M}_1\ \mathrm{M}_2\texttt{)},\, k]\!]) = \texttt{(if}\ \mathrm{E}\ \mathcal{F}([\![\mathrm{M}_1,\, k]\!])\ \mathcal{F}([\![\mathrm{M}_2,\, k]\!])\texttt{)} \\
\mathcal{F}([\![\texttt{(if}\ \mathrm{E}\ \mathrm{M}_1\ \mathrm{M}_2\texttt{)},\, \texttt{(}\lambda_{cont}\texttt{(}x\texttt{)}\ \mathrm{M}'\texttt{)}]\!]) = \\
\qquad \texttt{(letrec ((}j\ \texttt{(}\lambda_{jump}\ \texttt{(}j\texttt{)}\ \mathrm{M}'\texttt{)))}\ \texttt{(if}\ \mathrm{E}\ \mathcal{F}([\![\mathrm{M}_1,\, j]\!])\ \mathcal{F}([\![\mathrm{M}_2,\, j]\!])\texttt{))} \text{ where } j \text{ is new} \\
\mathcal{F}([\![\texttt{(loop}\ l\ \texttt{((}x\ \mathrm{E}_{initial}\texttt{)}\ \ldots\texttt{)}\mathrm{M}\texttt{)},\, \mathrm{C}]\!]) = \\
\qquad \texttt{(letrec ((}l\ \texttt{(}\lambda_{jump}\ \texttt{(}x\ \ldots\texttt{)}\ \mathcal{F}([\![\mathrm{M},\, \mathrm{C}]\!])\texttt{)))}\ \texttt{(}l\ \mathrm{E}_{initial}\ \ldots\texttt{))} \\
\mathcal{F}([\![\texttt{(}l\ \mathrm{E}\ \ldots\texttt{)},\, \mathrm{C}]\!]) = \texttt{(}l\ \mathrm{E}\ \ldots\texttt{)} \\
\\
\mathcal{V} : \mathrm{P} \rightarrow \mathrm{P}' \\
\mathcal{V}([\![\texttt{(}\lambda\texttt{(}x\ \ldots\texttt{)}\ \mathrm{M}\texttt{)}]\!]) = \texttt{(}\lambda_{proc}\ \texttt{(}x\ \ldots k\texttt{)}\ \mathcal{F}([\![\mathrm{M},\, k]\!])\texttt{)}
\end{array}
$$

Figure 1: Conversion to CPS

conversion to CPS will treat trivial expressions as values and not introduce continuations for them.

In the CPS grammar all λ's are annotated as being either *proc*, *cont*, or *jump*. The annotations indicate how the λ's are used, and, equivalently, how they can be compiled.

Proc is used as the translation of the λ forms in the source program. These are full procedures that eventually return a value and for this reason take a continuation as an argument. The *cont* and *jump* forms are continuations that are used in slightly different ways. The *cont* continuations are return points for calls to *proc*'s. *Jump* indicates that the continuation is called within the current procedure instead of being passed to another one. The CPS algorithm introduces λ_{jump} continuations when the two arms of a conditional have to rejoin at a common point and for the bodies of `loop`'s.

In terms of compilation strategy, *cont* λ's are return points, *jump*'s can be compiled as gotos, and *proc*'s require a complete procedure-call mechanism.

The grammar for the CPS language is as follows:

$$
\begin{array}{lcl}
\mathrm{M}' & ::= & \texttt{(}\mathrm{E}\ \mathrm{E}^*\ \mathrm{C}\texttt{)} \mid \\
 & & \texttt{(}k\ \mathrm{E}^*\texttt{)} \mid \\
 & & \texttt{(if}\ \mathrm{E}\ \mathrm{M}'\ \mathrm{M}'\texttt{)} \mid \\
 & & \texttt{(let ((}x\ \mathrm{E}\texttt{))}\ \mathrm{M}'\texttt{)} \mid \\
 & & \texttt{(letrec ((}x\ \mathrm{P}'\texttt{)*)}\ \mathrm{M}'\texttt{)} \\
\mathrm{C} & ::= & k \mid \texttt{(}\lambda_{cont}\texttt{(}x\texttt{)}\ \mathrm{M}'\texttt{)} \\
\mathrm{E} & ::= & x \mid \texttt{(}+\ \mathrm{E}\ \mathrm{E}\texttt{)} \mid \ldots \\
\mathrm{P}' & ::= & \texttt{(}\lambda_{proc}\ \texttt{(}x^*\ k\texttt{)}\ \mathrm{M}'\texttt{)} \mid \\
 & & \texttt{(}\lambda_{jump}\ \texttt{(}x^*\texttt{)}\ \mathrm{M}'\texttt{)} \\
\multicolumn{3}{l}{\text{where } x, k \in \text{variables}}
\end{array}
$$

The semantics is the obvious call-by-value semantics. The annotated λ's are all just λ, `let` is syntactic sugar for λ, and so on.

The identifiers x and k used in the grammar are members of the same syntactic class. Making them syntactically distinct, as is sometimes done [8], restricts how continuations are used and makes CPS and SSA entirely equivalent (see Section 6).

The function $\mathcal{F}$ defined in Figure 1 translates

```
(λ_proc (f limit k)
  (letrec ((l (λ_jump (i c)
                (if (= i limit)
                    (k c)
                    (f i (λ_cont (x)
                           (letrec ((j (λ_jump (c′)
                                         (l (+ i 1) c′))))
                             (if (= x 0)
                                 (j (+ c 1))
                                 (j c)))))))))
    (l 0 0)))
```

Figure 2: Example program in CPS

source expressions M to CPS expressions M′. It takes a continuation C, which is either an identifier or a λ_{cont}, as its second argument.

$\mathcal{F}$ has two rules for applications. The first rule is used when the continuation argument is an identifier bound by a `letrec` in the rule for `if` and ensures that all uses of such identifiers are called directly. The second is used in all other cases. For `let` a new continuation is created for the value. Identifier continuations are propagated through `if`'s; to avoid code expansion λ continuations (as opposed to continuations that are just an identifier) are marked as *jump* and bound to an identifier. `Loop` is translated into a recursive continuation. The loop begins by calling the continuation on the initial values of the iterative variables. Calls to the loop's label become calls to the recursive continuation, ignoring the current continuation.

As we are not interested in interprocedural analysis we will treat each λ_{proc} as a separate program (here we depend on the assumption that explicit environments have been introduced to take care of lexical scoping for any nested λ_{proc}'s). Because they are called at the point they are created, λ_{cont}'s can be considered as inlined procedures. The call sites of λ_{jump}'s are easily found: the λ_{jump}'s only occur as the bound value in `letrec`'s. Furthermore, all references to the variables to which the λ_{jump}'s are bound are calls to those variables.

The following is the sample SSA procedure from above written in the source language for the CPS transformation:

```
(λ (f limit)
  (loop l ((i 0) (c 0))
    (if (= i limit)
        c
        (let ((x (f i)))
          (let ((c′ (if (= x 0)
                        (+ c 1)
                        c)))
            (l (+ i 1) c′))))))
```

Figure 2 presents the CPS version of the same procedure. A λ_{cont} is introduced as the continuation for the call to `f` and λ_{jump}'s are used as the join point for the `if` and for the loop.

4 Converting CPS to SSA

In this section we define a syntactic translation that converts CPS procedures into SSA procedures.

The function $\mathcal{G}$ in Figure 3 translates non-trivial CPS expressions to SSA statements. The

$$
\begin{array}{l}
\mathcal{G} : \mathrm{M}' \to \mathrm{B} \\
\mathcal{G}(\llbracket \texttt{(let ((}x\ \mathrm{E}\texttt{))}\ \mathrm{M}'\texttt{)} \rrbracket) = x \leftarrow \mathrm{E}\texttt{;}\ \mathcal{G}(\llbracket \mathrm{M}' \rrbracket) \\
\mathcal{G}(\llbracket \texttt{(}\mathrm{E} \ldots \texttt{(}\lambda_{cont}\ \texttt{(}x\texttt{)}\ \mathrm{M}'\texttt{))} \rrbracket) = x \leftarrow \mathrm{E}\texttt{(} \ldots \texttt{);}\ \mathcal{G}(\llbracket \mathrm{M}' \rrbracket) \\
\mathcal{G}(\llbracket \texttt{(}\mathrm{E} \ldots k\texttt{)} \rrbracket) = \texttt{return}\ \mathrm{E}\texttt{(}\ldots\texttt{);} \\
\mathcal{G}(\llbracket \texttt{(}k\ \mathrm{E}\texttt{)} \rrbracket) = \texttt{return}\ \mathrm{E}\texttt{;} \\
\mathcal{G}(\llbracket \texttt{(}j\ \mathrm{E}_{0,i}\ \mathrm{E}_{1,i} \ldots \texttt{)} \rrbracket) = \texttt{goto}\ j_i\texttt{;} \\
\mathcal{G}(\llbracket \texttt{(if}\ \mathrm{E}\ \mathrm{M}'_1\ \mathrm{M}'_2\texttt{)} \rrbracket) = \texttt{if}\ \mathrm{E}\ \texttt{then}\ \mathcal{G}(\llbracket \mathrm{M}'_1 \rrbracket)\ \texttt{else}\ \mathcal{G}(\llbracket \mathrm{M}'_2 \rrbracket) \\
\mathcal{G}(\llbracket \texttt{(letrec (} \ldots \texttt{)}\ \mathrm{M}'\texttt{)} \rrbracket) = \mathcal{G}(\llbracket \mathrm{M}' \rrbracket) \\
\mathcal{G}_{proc} : \mathrm{P}' \to \mathrm{P} \\
\mathcal{G}_{proc}(\llbracket \texttt{(}\lambda_{proc}\ \texttt{(}x \ldots\texttt{)}\ \mathrm{M}'\texttt{)} \rrbracket) = \texttt{proc(}x \ldots\ k\texttt{)}\{\ \mathcal{G}(\llbracket \mathrm{M}' \rrbracket)\ \mathcal{G}_{jump}(\llbracket \texttt{(}\lambda_{jump}\ \ldots\ \texttt{)} \rrbracket) \ldots \} \\
\mathcal{G}_{jump} : j \times \texttt{(}\lambda_{jump}\ \texttt{(}x \ldots\texttt{)}\ \mathrm{M}'\texttt{)} \to \mathrm{L} \\
\mathcal{G}_{jump}(\llbracket j,\ \texttt{(}\lambda_{jump}\ \texttt{(}x \ldots\texttt{)}\ \mathrm{M}'\texttt{)} \rrbracket) = j : x \leftarrow \phi(\mathrm{E}_{0,0}, \mathrm{E}_{0,1}, \ldots)\texttt{;}\ \ldots\ \mathcal{G}(\llbracket \mathrm{M}' \rrbracket))
\end{array}
$$

Figure 3: Translation of CPS to SSA. The $\mathrm{E}_{i,j}$ on the right-hand side of the definition of $\mathcal{G}_{jump}$ are those from the left-hand side of the definition of $\mathcal{G}(\llbracket \texttt{(}j \ldots\texttt{)} \rrbracket)$.

simple binding forms, `let` and λ_{cont}, become assignments. `If` is essentially the same in both syntaxes. Uses of a λ_{proc}'s continuation variable become `return`'s while calls to `letrec`-bound continuation variables are translated as `goto`s.

The λ_{jump}'s in the program are ignored when found by $\mathcal{G}$. Each λ_{jump} is instead lifted up to become a labeled block in the SSA procedure. The arguments to the λ_{jump}'s, which are also ignored by $\mathcal{G}$, become the arguments to the ϕ-function that defines the value of the corresponding variable in the SSA program. In the definition of $\mathcal{G}_{jump}$, $\mathrm{E}_{0,1}$ is the first argument to the second call (in some arbitrary ordering) to j, the variable bound to $\texttt{(}\lambda_{jump}\ \texttt{(}x \ldots\ \texttt{)}\ \mathrm{M}'\texttt{)}$. That call is translated as `goto` j_1.

The translated program is syntactically correct, and it obeys the SSA restriction that there is exactly one assignment per variable (since each variable is bound exactly once in the CPS program). Variables in the translated program are assigned the values of the same expressions they were bound to in the CPS program, and evaluation order is preserved, so the two programs produce identical results.

Applying $\mathcal{G}_{proc}$ to the CPS example produces the original SSA example from above.

5 Converting SSA to CPS

We would like to produce an inverse of $\mathcal{G}$ to convert SSA programs to CPS. The function $\mathcal{H}$ in Figure 4, produced by a simple editing of the definition of $\mathcal{G}$, is almost an inverse of $\mathcal{G}$. The difficulty is the rule for the `letrec`'s that bind λ_{jump}'s. In translating λ_{jump}'s to labeled blocks $\mathcal{G}$ ignores their position in the CPS program. Translating a labeled block back into a λ_{jump} is straightforward, but we need to bind the λ_{jump} with a `letrec` somewhere in the newly created CPS program. The λ_{jump}'s need to be placed such that no variable is referred to outside the form that binds it.

Our placement algorithm uses the dominator tree of the SSA program. Statement M_0 is said to dominate M_1 if M_0 appears in every execution path between the start of the program and M_1. If M_2 also dominates M_1 then either M_2 dominates M_0 or vice versa. The dominator relation thus organizes a program's statements into a tree. The immediate dominator of M is M's parent in the dominator tree.

Given a labeled block, $J : \mathrm{I}^*\ \mathrm{B}$, in the SSA

$$\mathcal{H} : \mathrm{B} \rightarrow \mathrm{M}'$$
$$\mathcal{H}([\![x \leftarrow \mathrm{E}\texttt{;}\ \mathrm{B}]\!]) = \texttt{(let ((}x\ \mathrm{E}\texttt{))}\ \mathcal{H}([\![\mathrm{B}]\!])\texttt{)}$$
$$\mathcal{H}([\![x \leftarrow \mathrm{E}\texttt{(}\ \ldots\texttt{);}\ \mathrm{B}]\!]) = \texttt{(}\mathrm{E} \ldots \texttt{(}\lambda_{cont}\ \texttt{(}x\texttt{)}\ \mathcal{H}([\![\mathrm{B}]\!])\texttt{)))}$$
$$\mathcal{H}([\![\texttt{if}\ \mathrm{E}\ \texttt{then}\ \mathrm{B}_1\ \texttt{else}\ \mathrm{B}_2]\!]) = \texttt{(if}\ \mathrm{E}\ \mathcal{H}([\![\mathrm{B}_1]\!])\ \mathcal{H}([\![\mathrm{B}_2]\!])\texttt{)}$$
$$\mathcal{H}([\![\texttt{return}\ \mathrm{E}\texttt{(}\ldots\texttt{);}]\!]) = \texttt{(}\mathrm{E} \ldots k\texttt{)}$$
$$\mathcal{H}([\![\texttt{return}\ \mathrm{E}\texttt{;}]\!]) = \texttt{(}k\ \mathrm{E}\texttt{)}$$
$$\mathcal{H}([\![\texttt{goto}\ j_i\texttt{;}]\!]) = \texttt{(}j\ \mathrm{E}_{0,i}\ \mathrm{E}_{1,i}\ \ldots\texttt{)}$$
$$\mathcal{H}_{proc} : \mathrm{P} \rightarrow \mathrm{P}'$$
$$\mathcal{H}_{proc}([\![\texttt{proc(}x \ldots\ k\texttt{)}\{\mathrm{B\ L}^*\}]\!]) = \texttt{(}\lambda_{proc}\ \texttt{(}x \ldots\texttt{)}\ \mathcal{G}([\![\mathrm{B}]\!])\texttt{)}$$
$$\mathcal{H}_{jump} : \mathrm{L} \rightarrow \texttt{(}\lambda_{jump}\ \texttt{(}x \ldots\texttt{)}\ \mathrm{M}'\texttt{)}$$
$$\mathcal{H}_{jump}([\![j : x \leftarrow \phi\texttt{(}\mathrm{E}_{0,0},\ \mathrm{E}_{0,1},\ \ldots\texttt{);}\ \ldots\ \mathrm{B}]\!]) = \texttt{(}\lambda_{jump}\ \texttt{(}x \ldots\texttt{)}\ \mathcal{H}([\![\mathrm{B}]\!])\texttt{)}$$

Figure 4: Translation of SSA to CPS. The $\mathrm{E}_{i,j}$ on the right-hand side of the definition of $\mathcal{H}([\![\texttt{goto}\ J_i\texttt{;}]\!])$ are those from the left-hand side of the definition of $\mathcal{H}_{jump}$.

program, with M the immediate dominator of I* B, we will replace $\mathcal{H}([\![\mathrm{M}]\!])$ with

$\texttt{(letrec ((j}\ \mathcal{H}_{jump}([\![\mathrm{I}^*\ \mathrm{B}]\!])\texttt{))}\ \mathcal{H}([\![\mathrm{M}]\!])\texttt{)}$

in the CPS procedure. If two or more labels have the same immediate dominator their λ_{jump}'s are placed in a single `letrec`.

Theorem: all uses of variables in the CPS program produced by $\mathcal{H}$ are properly in scope.

Proof: We know that every use of a variable in the SSA program is dominated by the variable's assignment statement. Assignment statements in the SSA program correspond to binding forms in the CPS program. By inspection $\mathcal{H}$ preserves the dominator tree, so it is sufficient to show that every statement in the CPS program is lexically inferior to its immediate dominator, and thus to all its dominators, including the binding forms for any referenced variables.

Again inspecting the definition of $\mathcal{H}$, we can see that when the M' produced by $\mathcal{H}$ is the body of either a `LET`, a λ_{proc}, or a λ_{cont}, or if it is either arm of an `if`, then it is lexically inferior to its immediate dominator. The remaining possible location for an M' is as the body of a λ_{jump}, and each λ_{jump} is by construction lexically inferior to the dominators of its immediate dominator. This would lead to a problem if the immediate dominator were a binding form, but this cannot happen. If $\texttt{(let ((}x\ \mathrm{E}\texttt{))}\ \mathrm{M}'\texttt{)}$ or $\texttt{(}\mathrm{E} \ldots \texttt{(}\lambda_{cont}\ \texttt{(}x\texttt{)}\ \mathrm{M}'\texttt{))}$ dominates a λ_{jump}, then M' does as well.

The arguments to the ϕ-functions are also placed so as to be lexically inferior to their dominators. QED.

Using $\mathcal{H}$ to convert the original SSA example to CPS produces a program identical to the CPS program shown in Figure 2.

6 Why CPS and SSA Are Not Identical

The function $\mathcal{G}$ cannot be applied to all CPS programs. It depends on the λ's being correctly annotated, which in turn depends on continuations being used in a somewhat restricted fashion. If non-local returns, such as longjumps in C or call-with-current-continuation in Scheme, are translated into uses of continuations in the CPS program there is no way to label the continuations so used. They are neither simply passed as a procedure's continuation, nor are all of their calls tail-recursive calls within the procedure in which they were created, so neither λ_{cont} or λ_{jump} is appropriate. A fourth annotation could be introduced

```
(λ_proc (f limit k)
  (letrec ((l (λ_proc (i c k′)
                 (if (= i limit)
                     (k′ c)
                     (f i (λ_cont (x)
                            (letrec ((j (λ_jump (c′)
                                           (l (+ i 1) c′ k′))))
                              (if (= x 0)
                                  (j (+ c 1))
                                  (j c)))))))))
    (l 0 0 k)))
```

Figure 5: Example program using Scheme's named-`let` translated into CPS.

for continuations that are created in one procedure and called in another. There would still be no direct way to represent the program in SSA.

7 Compiling Imperative Programs

We have shown that programs can be translated into CPS and from there to SSA programs by a series of syntactic transformations. Producing an SSA program normally requires flow-analysis, as the program directly expresses use↔definition chaining. Where is the flow information in the CPS program coming from?

The source program for the CPS transformation is a functional program, and as such contained the required flow information. If we had started with an imperative program, meaning one using `set!` to modify the values of variables, translation into our CPS source language would have required adding explicit cells to hold the values of all variables that were the targets of `set!` expressions. The λ_{jump}'s in the resulting CPS program would take no arguments and there would be no ϕ-functions in the corresponding SSA program, only a lot of stores and fetches. Transforming such a program into a more useful form would require doing some flow-analysis, similar to that done in translating imperative programs to SSA.

8 Compiling (Mostly) Functional Programs

For programs that do not side-affect the values of variables, viewing CPS procedures as SSA procedures shows that CPS nicely reflects intraprocedural definition↔use associations without any need for flow analysis. There is still a problem. Programs written in languages such as ML and Scheme tend to have few side-affected variables, making dataflow visible as discussed above. The difficulty is that these same languages use recursion to express iteration. They do not use iterative constructs like `loop` in the CPS source language used above. So while we get intraprocedural flow information more or less for free, what we really want is interprocedural information.

Interprocedural analysis is difficult, but we can instead take the approach of increasing the size of the procedures. The idea is to find a set of procedures all of which are always called with the same continuation, and then to substitute that continuation for the procedures' continuation variables. The procedures are then themselves continuations nested within a single large procedure, removing any need for interprocedural information.

More precisely, given a set procedures $\mathrm{P}_0 \ldots \mathrm{P}_n$ bound by one or more `letrec` forms to identifiers $f_0 \ldots f_n$ and a continuation C such that every use of f_i is either a tail-recursive call from within P_j or a non-tail-recursive call being passed continuation C, we can perform the following transformation.

Let P_i = (λ_{proc} (... k_i) M_i). If C is (λ_{cont} ...), let j be a new identifier; if C is an identifier, let j be that identifier.

1. Replace all references to the k_i with j and remove them from the variable lists of the P_i, changing the P_i from λ_{proc}'s to λ_{jump}'s.

2. Remove the continuation from all calls to the f_i.

3. Let M be the smallest form containing every non-tail-recursive call to the f_i. Replace M with (letrec (...) M) where (...) binds j to C if C = (λ_{cont} ...) and also binds f_i to P_i for every f_i whose original binding form is lexically apparent at M.

Because we restricted ourselves to having the f_i be bound by `letrec` and being called directly (as opposed to being passed to another procedure and then being called, for example), this is a purely syntactic transformation. More sophisticated versions are clearly possible. The simple syntactic version presented here applies to many common uses of recursive procedures.

If our CPS source program used Scheme's named-`let` syntax instead of the restricted `loop` version used above, applying $\mathcal{G}$ to it would produce the program shown in Figure 5. The recursive procedure is now a full procedure. Applying the above transformation to this program, with P_0 = (λ_{proc} (i c k′) ...) and C = j = 1, produces the procedure shown in Figure 2.

We need to show that the transformation is correct and preserves the map to SSA. Substituting the value C (or an identifier bound to C) for the variables k_i is safe, because the k_i are always bound to that value. Scoping is preserved, because any free identifiers in C are in scope at M, otherwise they would be out of scope at some occurrence of C in the original program. Moving the P_i to a lexically inferior form also preserves scoping, and all uses of the f_i are at or below the inserted `letrec` that now binds them.

The function $\mathcal{G}$ can be applied to the transformed program to produce an equivalent SSA version. We have created more λ_{jump}'s, but they obey the same restrictions as the ones created by the CPS algorithm: the λ_{jump}'s are bound by `letrec`, all uses of the variables to which they are bound are tail calls, and the bindings and uses all lie within a single λ_{proc} (because any outlying P_i were moved in step 3 of the transformation).

9 Related Work and Conclusion

The similarity between CPS and SSA has been noted by others. Appel [1] and O'Donnell [9] both briefly describe the relationship between ϕ-functions and parameters to procedures whose call sites are known. Here we have shown that the correspondence is exact, and, more importantly, goes both ways. The main difference between a CPS procedure and an SSA procedure is the syntax. Compiler writers can use a single representation and take advantage of both the work on intraprocedural optimizations that has been done using SSA and the work on interprocedural optimizations done using CPS.

The other contribution of this paper is that the correspondence can be made to be useful. As long as procedures are too small to contain loops, considering them as SSA procedures does not help much. We have shown how to merge looping procedures together such that the result can treated as a single procedure (a restricted and somewhat more complex version of this transformation is mentioned in two of our earlier papers, [6, 7], but without any discussion of the implications for program analysis). Without this transformation analyzing loops that are expressed as recursive procedures requires interprocedural analysis, as in [11], and in this case using CPS may be more difficult than necessary [10].

References

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

[2] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170, 1994.

[3] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3):1–55, 1991.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[5] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, 1992. Also McGraw-Hill, Chicago, 1992.

[6] Richard Kelsey. Compilation by program transformation. Technical Report YALEU/DCS/TR-702, Department of Computer Science, Yale University, New Haven, CT, 1989.

[7] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Conf. Rec. 16 ACM Symposium on Principles of Programming Languages*, pages 281–292, 1989.

[8] David A. Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. Orbit: An optimizing compiler for scheme. In *Proceedings SIGPLAN '86 Symposium on Compiler Construction*, 1986. *SIGPLAN Notices 21*(7), July, 1986, 219-223.

[9] Ciaran O'Donnell. *High level compiling for low level machines*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, 1994.

[10] Amr Sabry and Matthias Felleisen. Is continutation-passing useful for data flow analysis? In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 1–12, 1994.

[11] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie-Mellon University, 1991.

[12] Guy L. Steele. Rabbit: A compiler for Scheme. Technical Report 474, Massachusetts Institute of Technology, Cambridge, MA, May 1978.

[13] Michael Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 162–174, 1992.