

CS107: CPS Translation and Other IRs

Guannan Wei

guannan.wei@tufts.edu

October 8, 2026

Fall 2026

Tufts University

The translation presented before has two shortcomings:

- 1) It produces terms containing useless continuations, and
- 2) It produces suboptimal MiniScala/CPS code for some conditionals.

Solution: improving the translation by defining different translations depending on the source (i.e. MiniScala) context surrounding the expression being translated.

Useless Continuations

The first problem can be illustrated with the MiniScala term:

```
def f(g: () => Int) = g(); f
```

which in the empty context gets translated to:

```
deff f(c, g) = {  
  defc j(r) = { c(r) };  
  g(j)  
};  
f
```

Useless Continuations

The first problem can be illustrated with the MiniScala term:

```
def f(g: () => Int) = g(); f
```

which in the empty context gets translated to:

```
deff f(c, g) = {  
  defc j(r) = { c(r) };  
  g(j)  
};  
f
```

instead of the equivalent and more compact:

```
deff f(c, g) = { g(c) };  
f
```

Useless Continuations

The first problem can be illustrated with the MiniScala term:

```
def f(g: () => Int) = g(); f
```

which in the empty context gets translated to:

```
deff f(c, g) = {  
  defc j(r) = { c(r) };  
  g(j)  
};  
f
```

instead of the equivalent and more compact:

```
deff f(c, g) = { g(c) };  
f
```

Improvement: the function call could be translated more efficiently since it appears as the *last expression of the function* (i.e. the **tail position**).

Suboptimal Conditionals (1)

The second problem can be illustrated with the MiniScala term:

```
if (if (a) b else false) x else y
```

which, in the empty context, gets translated to:

```
defc ci1(v1) = { v1 };           // receives the final value
defc ct1() = { ci1(x) };          // outer then branch
defc cf1() = { ci1(y) };          // outer else branch
vall f1 = false;
defc ci2(v2) = {                  // join point of the inner if
  if (v2 != f1) ct1 else cf1 };
defc ct2() = { ci2(b) };          // inner then branch
defc cf2() = {                     // inner else branch
  vall i1 = false; ci2(i1) };
vall f2 = false;
if (a != f2) ct2 else cf2
```

Suboptimal Conditionals (2)

A much better translation for:

```
if (if (a) b else false) x else y
```

would be:

```
defc ci1(v1) = { v1 };
defc ct1() = { ci1(x) };
defc cf1() = { ci1(y) };
defc ca1() = {                                // if a is true, test b
    vall i1 = false;
    if (b != i1) ct1 else cf1 }; // use b directly to determine the outer branch
vall i2 = false;
if (a != i2) ca1 else cf1 // inner-else branch take a shortcut to outer-else
```

which immediately applies continuation cf_1 if a is false.

Improvement: one branch of the inner **if** is a simple boolean literal, so we can propagate the inner condition to outer branches.

Common to the two problems: the translation could be better if the translation is aware of the surrounding contexts.

- In the first example, the function call could be translated more efficiently since it appears as the *last expression of the function* (i.e. the **tail position**).
- In the second example, the nested **if** expression could be translated more efficiently, since it appears in the condition of another **if** expression and one of its branches is a simple boolean literal (here `false`).

Therefore, instead of having one translation function, we should have several: one per source context worth considering!

A Better Translation

We split the single translation function into three separate ones:

- 1) $\llbracket \cdot \rrbracket_N \mathcal{C}$, taking as before a term to translate and a context $\mathcal{C}$. The context $\mathcal{C}$'s hole must be plugged with a *name/symbol* bound to the value of the term.

A Better Translation

We split the single translation function into three separate ones:

- 1) $\llbracket \cdot \rrbracket_N \mathcal{C}$, taking as before a term to translate and a context $\mathcal{C}$. The context $\mathcal{C}$'s hole must be plugged with a *name/symbol* bound to the value of the term.
- 2) $\llbracket \cdot \rrbracket_T \mathcal{C}$, taking a term to translate and a symbol $\mathcal{C}$ for the continuation. The symbol represents a continuation in the translated program and is *applied* (in the target program) to the value of the term.

A Better Translation

We split the single translation function into three separate ones:

- 1) $\llbracket \cdot \rrbracket_N \mathcal{C}$, taking as before a term to translate and a context $\mathcal{C}$. The context $\mathcal{C}$'s hole must be plugged with a *name/symbol* bound to the value of the term.
- 2) $\llbracket \cdot \rrbracket_T \mathcal{C}$, taking a term to translate and a symbol $\mathcal{C}$ for the continuation. The symbol represents a continuation in the translated program and is *applied* (in the target program) to the value of the term.
- 3) $\llbracket \cdot \rrbracket_C \text{ ct } \text{ cf}$, taking a term to translate and two parameterless continuations, ct and cf . The continuation ct is applied when the term evaluates to a true value, while the continuation cf is applied when it evaluates to a false value.

The Better Translation In Scala

In the compiler, the three translations are simply three mutually-recursive functions, with the following signatures:

```
def nonTail(t: MSTree)
    (c: Symbol => CPSTree): CPSTree
```

```
def tail(t: MSTree,
    c: Symbol): CPSTree
```

```
def cond(t: MSTree,
    ct: Symbol,
    cf: Symbol): CPSTree
```

The Non-tail Translation

$\llbracket \cdot \rrbracket_N$ is called the **non-tail** translation as it is used in non-tail contexts: once the term is evaluated, there are more steps to perform (rather than simply forwarding the value to an outer continuation).

For example, the arguments of a primitive are always in a non-tail context, since once they are evaluated, the primitive has to be applied on their value:

```
 $\llbracket p(e_1, e_2, \dots) \rrbracket_N C =$   
   $\llbracket e_1 \rrbracket_N (\lambda v_1 (\llbracket e_2 \rrbracket_N (\lambda v_2 \dots)))$ ;  
   $\text{val}_p \underline{n} = p(v_1, v_2, \dots)$ ;  
   $C[n]$   
  where  $p$  is not a logical primitive
```

The Tail Translation

The **tail translation** $\llbracket \cdot \rrbracket_T$ is used whenever the context passed to the simple translation has the form $\lambda v(c(v))$. We can directly pass the name of the continuation c to which the value of expression should be applied.

For example, the previous naive translation of function definition:

```
 $\llbracket \text{def } f_1(n_{1,1}: \_, \dots) = e_1; \dots ; e \rrbracket_C =$   
 $\text{def}_f f_1(\underline{c}, n_{1,1}, \dots) = \{ \llbracket e_1 \rrbracket(\lambda v(c(v))) \};$   
 $\text{def}_f \dots;$   
 $\llbracket e \rrbracket_C$ 
```

becomes:

```
 $\llbracket \text{def } f_1(n_{1,1}: \_, \dots) = e_1; \text{def } \dots ; e \rrbracket_N C =$   
 $\text{def}_f f_1(\underline{c}, n_{1,1}, \dots) = \{ \llbracket e_1 \rrbracket_T C \};$   
 $\text{def}_f \dots;$   
 $\llbracket e \rrbracket_N C$ 
```

The Cond Translation (1)

The cond translation $\llbracket \cdot \rrbracket_C$ is to translate condition term that decides how execution must proceed. It takes two continuations as arguments: the first is to be applied when the condition is true, while the second is to be applied when it is false.

This translation is used to handle the condition of an if expression:

```

$$\llbracket \text{if } (e_1) \text{ } e_2 \text{ else } e_3 \rrbracket_N C =$$

$$\begin{aligned} & \text{def}_c \underline{c}(r) = \{ C[r] \}; \\ & \text{def}_c \underline{ct}() = \{ \llbracket e_2 \rrbracket_T c \}; \\ & \text{def}_c \underline{cf}() = \{ \llbracket e_3 \rrbracket_T c \}; \\ & \llbracket e_1 \rrbracket_C \underline{ct} \underline{cf} \end{aligned}$$

```

The Cond Translation (2)

Having a separate translation for conditional expressions makes the efficient compilation of conditionals with literals in one of their branch possible:

$$\llbracket \text{if } (e_1) \text{ false else true} \rrbracket_C \text{ ct cf} = \\ \llbracket e_1 \rrbracket_C \text{ cf ct}$$

Intuition: `if` `[[ (if (a) false else true) ]]` `x` `else` `y` is equivalent to `if (a) y else x`.

The Cond Translation (2)

Having a separate translation for conditional expressions makes the efficient compilation of conditionals with literals in one of their branch possible:

$$\llbracket \text{if } (e_1) \text{ false else true} \rrbracket_C \text{ ct cf} = \\ \llbracket e_1 \rrbracket_C \text{ cf ct}$$

Intuition: **if** $\llbracket (\text{if } (a) \text{ false else true}) \rrbracket$ **x** **else** **y** is equivalent to **if** (a) **y** **else** **x**.

Similarly:

$$\llbracket \text{if } (e_1) e_2 \text{ else false} \rrbracket_C \text{ ct cf} = \\ \text{def}_c \underline{c}() = \{ \llbracket e_2 \rrbracket_C \text{ ct cf} \}; \\ \llbracket e_1 \rrbracket_C \underline{c} \text{ cf};$$

... and so on for all conditionals with at least one constant branch.

The Cond Translation (3)

```
[[ while (e1) e2; e3 ]]N C =  
  defc loop(d) = {  
    defc c() = { [[e3]]N C };  
    defc ct() = { [[e2]]T loop };  
    [[e1]]C ct c  
  };  
  valt d = ();  
  loop(d)
```

Popular in functional languages:

- Continuation-passing style (CPS)
- Administrative normal form (ANF)

Popular in imperative languages:

- Sea-of-nodes (graph-based) IRs: JVM, GraalVM, V8
- Register-transfer language (RTL) + control-flow graph (CFG): lower-level representation closer to machine code
- Static Single-Assignment (SSA) form: LLVM, GCC

IR #2: Standard RTL/CFG - Register-transfer Language

Register-transfer language (RTL):

- Most operations performs a computation using virtual registers (i.e. variables),
- Operations store the result in another virtual register.

Example: $x \leftarrow y + z$: adding variables y and z , storing the result in x could be written.

RTLs are very close to assembly languages, the main difference being that the number of virtual registers is usually not bounded.

Control-Flow Graph

A control-flow graph (CFG) is a directed graph whose nodes are the individual instructions of a function, and whose edges represent control-flow.

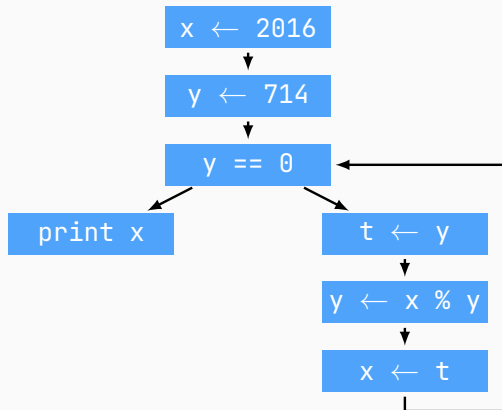
If the instruction of n_2 can be executed immediately after the instruction of n_1 , then there is an edge in the CFG from n_1 to n_2 .

RTL/CFG is an intermediate representation where each function of the program is represented as a control-flow graph whose nodes contain RTL instructions.

This kind of representation is very common in the later stages of compilers, especially those for imperative languages.

RTL/CFG Example

Computation of the GCD of 2016 and 714 in a typical RTL/CFG representation.

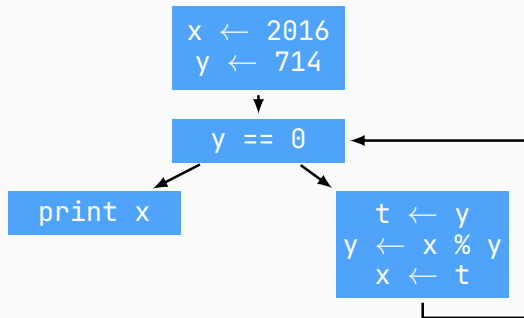


Basic block: a list of instructions that execute sequentially. Control can only enter through the first instruction of the block and leave through the last.

Basic blocks are often used as the nodes of the CFG, instead of individual instructions. This has the advantage of reducing the number of nodes in the CFG, but also complicates data-flow analyses.

RTL/CFG Example

The same examples as before, but with basic blocks instead of individual instructions.



One problem of RTL/CFG is that even very simple optimizations (e.g. constant propagation, common-subexpression elimination) require *data-flow analyses*. This is because a single variable can be assigned multiple times.

(We will cover data-flow analyses later in the course.)

One problem of RTL/CFG is that even very simple optimizations (e.g. constant propagation, common-subexpression elimination) require *data-flow analyses*. This is because a single variable can be assigned multiple times.

(We will cover data-flow analyses later in the course.)

Is it possible to improve RTL/CFG so that these optimizations can be performed without prior analysis?

Yes, by using a single-assignment variant of RTL/CFG!

Static single-assignment (SSA): each variable has only one definition in the RTL/CFG representation.

- That single definition can be executed many times when the program is run (e.g., inside a loop), hence the qualifier static.
- SSA form is popular (e.g. LLVM) because it simplifies several optimizations and analyses.

Most (imperative) programs are not naturally in SSA form, and must therefore be transformed so that they are.

Transforming to SSA: Straight-line Code

Transforming a piece of straight-line code (i.e. without branches) to SSA is trivial: each definition of a given name gives rise to a new version of that name, identified by a subscript:

// non-SSA form

```
x <- 12
y <- 15
x <- x + y
y <- x + 4
z <- x + y
y <- y + 1
```

// corresponding SSA form

```
x1 <- 12
y1 <- 15
x2 <- x1 + y1
y2 <- x2 + 4
z1 <- x2 + y2
y3 <- y2 + 1
```

Join-points in the CFG (i.e., nodes with more than one predecessors) requires additional treatment, as each predecessor can bring its own version of a given name.

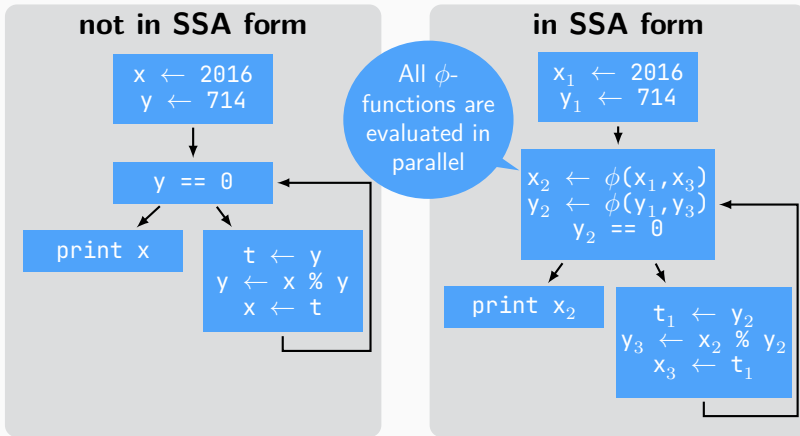
```
entry:
    x <- f()
    br cond left right
left:
    x2 <- x1 - 1
    jmp join
right:
    x3 <- x1 + 1
    jmp join
join:
    print x // which x should we use?
```

Join-points in the CFG (i.e., nodes with more than one predecessors) requires additional treatment, as each predecessor can bring its own version of a given name.

Solution: introduce a fictional Φ -function at the join point.

- A Φ -function takes as argument all the versions of the variable to reconcile, and
- automatically selects the right one depending on the incoming control flow.
- Φ -functions must be placed before any other non- Φ instructions.

Φ -functions Example



Evaluation of Φ -functions

It is crucial to understand that all Φ -functions of a block are evaluated in parallel, and not in sequence as the representation might suggest!

entry:

```
    i <- 5
```

l0:

```
    x0 <- 0
```

```
    y0 <- 0
```

```
    jmp l1
```

l1:

```
    x1 <-  $\Phi$ (l0 x0, l1 y1)
```

```
    y1 <-  $\Phi$ (l0 y0, l1 x1)
```

```
    print x1 y1
```

```
    cond <- (i > 0)
```

```
    i <- i - 1
```

```
    br cond l1 end
```

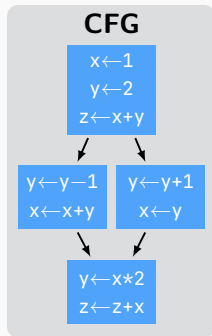
end:

(Naive) Building of SSA Form

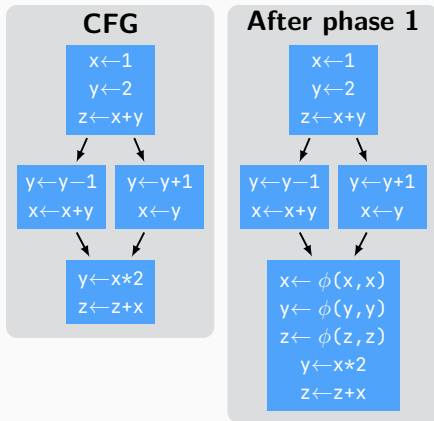
Naive technique to build SSA form:

- For each variable x of the CFG, at each join point n , insert a Φ -function of the form $x = \Phi(x, \dots, x)$ with as many parameters as n has predecessors,
- Follow each variable's assignments through the CFG, determine which assignment supplies the value at each use, and rename that use to match the assignment's unique name.

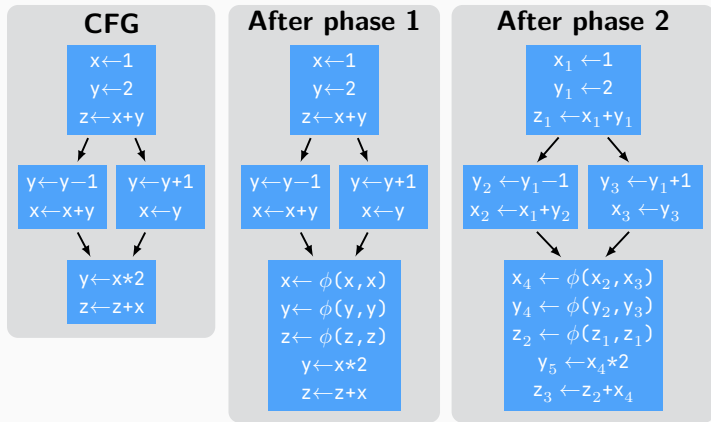
(Naive) Building of SSA Form



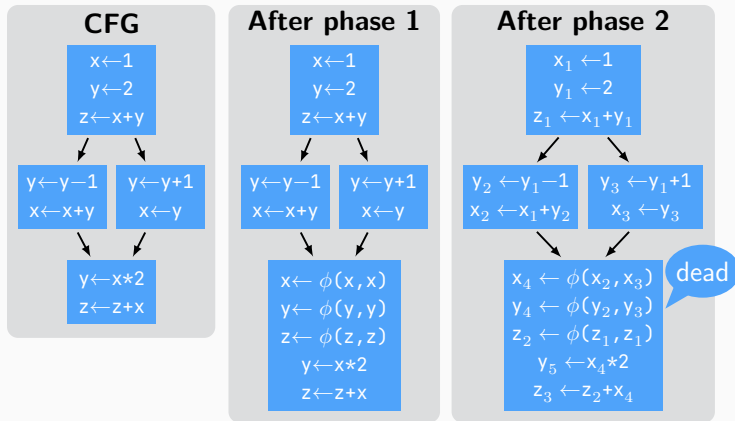
(Naive) Building of SSA Form



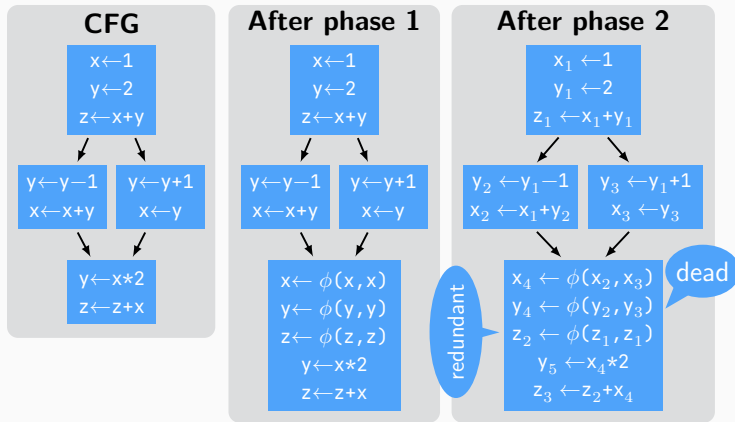
(Naive) Building of SSA Form



(Naive) Building of SSA Form



(Naive) Building of SSA Form



The naive technique builds a program in SSA form and is equivalent to the original one.

However, it introduces too many Φ -functions: some are dead, and some are redundant. It in fact builds the **maximal** SSA form.

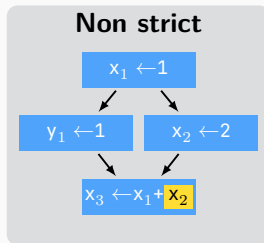
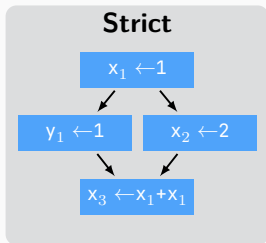
Better techniques exist to translate a program to SSA form.

Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. Cytron et al. ACM Transactions on Programming Languages and Systems, 1991

Strict SSA Form

Strict SSA form: an SSA form that all uses of a variable are dominated by the definition of that variable. In a CFG, a node n_1 dominates a node n_2 if all paths from the entry node to n_2 go through n_1 .

Strict SSA form guarantees that no variable is used before being defined.



Comparing IRs: MiniScala/CPS vs RTL/CFG in SSA

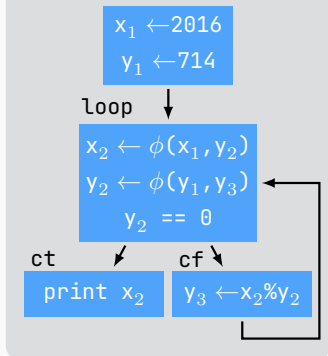
Correspondence between SSA and CPS:

RTL/CFG in SSA	MiniScala/CPS
Named basic block	continuation
Φ -function	continuation argument
jump	continuation invocation
strict form	scoping rules

CPS is more powerful/flexible in that continuations can be argument, supporting more dynamic control flow structure.

MiniScala/CPS vs RTL/CFG in SSA

RTL/CFG in SSA form



```
defc loop(x2, y2) = {  
  defc ct() = { print(x2) };  
  defc cf() = {  
    valp y3 = x2 % y2;  
    loop(y2, y3)  
  };  
  vall z = 0;  
  if (z == y2) ct else cf  
};  
vall x1 = 2016; vall y1 = 714;  
loop(x1, y1)
```

Continuation-based, functional IRs like MiniScala/CPS are SSA done right, or even should replace it - or, at the very least, Φ -functions should be replaced by continuations arguments.

(This is fortunately starting to happen, e.g. the Swift Intermediate Language has basic-blocks with arguments.)

Reference: MiniScala/CPS is heavily based on the intermediate representation presented by Andrew Kennedy in *Compiling with Continuations, Continued*, in Proceedings of the International Conference on Functional Programming (ICFP) 2007.