CS107: Improved CPS Translation and Other IRs

Guannan Wei
guannan.wei@tufts.edu
February 10, 2026

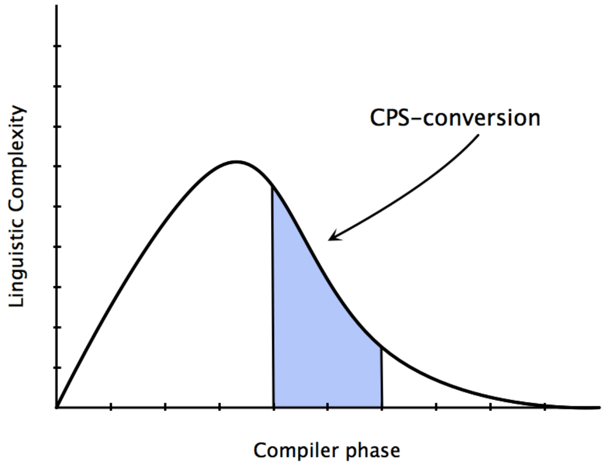
Spring 2026

Tufts University

Today

- Review the CPS translation
- Improve the CPS translation
- SSA-based IRs

Recap



[1]

[1]Image credit: https://matt.might.net/articles/cps-conversion/

Selected Rules for CPS Translation

The translation function:

$[\![\cdot]\!]$: `MSTree=> (Symbol => CPSTree) => CPSTree`

Example:

$[\![t]\!]\ c$: `CPSTree`
where t: `MSTree` and c: `Symbol => CPSTree`

Key ideas:

- Translation under a context, the "result" will be plugged into the context
- Context (aka continuation) is represented as a function in the meta-language
- Translation follows the evaluation order

Selected Rules for CPS Translation

$[\![\mathtt{n}]\!]\ \mathtt{C} = \mathtt{C[n]}$ *where n is an identifier for immutable variable*

$[\![\mathtt{l}]\!]\ \mathtt{C} = \mathtt{val}_l\ \underline{\mathtt{n}} = \mathtt{l};\ \mathtt{C[n]}$ *where l is a literal*

$[\![\ \mathtt{val}\ \mathtt{n}_1 = \mathtt{e}_1;\ \mathtt{e}\ ]\!]\ \mathtt{C} =$
$\quad [\![\mathtt{e}_1]\!](\lambda \mathtt{v}\ (\mathtt{val}_p\ \mathtt{n}_1 = \mathtt{id(v)};\ [\![\mathtt{e}]\!]\ \mathtt{C}))$

$[\![\ \mathtt{var}\ \mathtt{n}_1 = \mathtt{e}_1;\ \mathtt{e}\ ]\!]\ \mathtt{C} =$
$\quad \mathtt{val}_l\ \underline{\mathtt{s}} = \mathtt{1};$
$\quad \mathtt{val}_p\ \mathtt{n}_1 = \mathtt{block\text{-}alloc\text{-}var(s)};$
$\quad \mathtt{val}_l\ \underline{\mathtt{z}} = \mathtt{0};$
$\quad [\![\mathtt{e}_1]\!](\lambda \mathtt{v}\ (\mathtt{val}_p\ \underline{\mathtt{d}} = \mathtt{block\text{-}set}(\mathtt{n}_1,\ \mathtt{z},\ \mathtt{v});\ [\![\mathtt{e}]\!]\ \mathtt{C}))$

$[\![\ \mathtt{n}_1 = \mathtt{e}_1\ ]\!]\ \mathtt{C} =$
$\quad \mathtt{val}_l\ \underline{\mathtt{z}} = \mathtt{0};$
$\quad [\![\mathtt{e}_1]\!](\lambda \mathtt{v}\ (\mathtt{val}_p\ \underline{\mathtt{d}} = \mathtt{block\text{-}set}(\mathtt{n}_1,\ \mathtt{z},\ \mathtt{v});\ \mathtt{C[v]}\ ))$

Selected Rules for CPS Translation

```
⟦n⟧ C =
  val_l z = 0;
  val_p v = block-get(n, z); C[v]
  where n is an identifier for mutable variable

⟦ def f_1(n_{1,1}: _, ...) = e_1; def ... ; e⟧ C =
  def_f f_1(c, n_{1,1}, ...) = {
    ⟦e_1⟧(λv(c(v)))
  };
  def_f ...;
  ⟦e⟧ C

⟦ e(e_1, e_2, ...) ⟧ C =
  ⟦e⟧(λv(⟦e_1⟧(λv_1(⟦e_2⟧(λv_2(...
  def_c c(r) = { C[r] };
  v(c, v_1, v_2 ...)))))))
```

Example

Let k_0 be the initial context λv(val$_l$ z $=$ 0; halt(z)).

```
  ⟦ f(g(1, 2)) ⟧ k₀
{ f(g(1, 2)) is a function application }
 =  ⟦f⟧(λv(⟦g(1, 2)⟧(λv₁
    def_c c₁(r₁) = { k₀[r₁] };
    v(c₁, v₁))))
 = ???
```

Example

Let k_0 be the initial context λv(val$_l$ z $=$ 0; halt(z)).

```
 ⟦ f(g(1, 2)) ⟧ k₀
{ f(g(1, 2)) is a function application }
=  ⟦f⟧(λv(⟦g(1, 2)⟧(λv₁
   def_c c₁(r₁) = { k₀[r₁] };
   v(c₁, v₁))))
{ f is an immutable variable }
=  ⟦g(1, 2)⟧(λv₁
   def_c c₁(r₁) = { k₀[r₁] };
   f(c₁, v₁))
{ let's remember the continuation in the meta-lang as k₁ }
=  ⟦g(1, 2)⟧ k₁
{ g(1, 2) is a function application }
=  ⟦g⟧(λv_g(⟦1⟧(λv₁(⟦2⟧(λv₂(
   def_c c₂(r₂) = { k₁[r₂] };
   v_g(c₂, v₁, v₂))))))
{ g is an immutable variable }
```

Example

```
=  ⟦g⟧(λv_g(⟦1⟧(λv_1(⟦2⟧(λv_2(
   def_c c_2(r_2) = { k_1[r_2] };
   v_g(c_2, v_1, v_2))))))
{ g is an immutable variable }
=  ⟦1⟧(λv_1(⟦2⟧(λv_2(
   def_c c_2(r_2) = { k_1[r_2] };
   g(c_2, v_1, v_2))))
{ 1 and 2 are literals }
=  val_l n_1 = 1;
   val_l n_2 = 2;
   def_c c_2(r_2) = { k_1[r_2] };
   g(c_2, n_1, n_2)
{ inline and apply continuation remembered as k_1 }
=  val_l n_1 = 1;
   val_l n_2 = 2;
   def_c c_2(r_2) = {
     def_c c_1(r_1) = { k_0[r_1] }; f(c_1, r_2)
   };
   g(c_2, n_1, n_2)
```

Example

```
{ inline and apply continuation remembered as k₀ }
=  val_l n₁ = 1;
   val_l n₂ = 2;
   def_c c₂(r₂) = {
     def_c c₁(r₁) = { val_l z = 0; halt(z) }
     f(c₁, r₂)
   };
   g(c₂, n₁, n₂)
{ we are done! }
```

Improving The Translation

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.

One solution: define 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
```

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:

```
def_f f(c, g) = {
  def_c 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:

```
def_f f(c, g) = {
  def_c j(r) = { c(r) };
  g(j)
};
f
```

instead of the equivalent and more compact:

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

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:

```
def_c ci_1(v_1) = { v_1 };
def_c ct_1() = { ci_1(x) };
def_c cf_1() = { ci_1(y) };
val_l f_1 = false;
def_c ci_2(v_2) = {
  if (v_2 != f_1) ct_1 else cf_1 };
def_c ct_2() = { ci_2(b) };
def_c cf_2() = {
  val_l i_1 = false; ci_2(i_1) };
val_l f_2 = false;
if (a != f_2) ct_2 else cf_2
```

Suboptimal Conditionals (2)

A much better translation for:

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

would be:

```
def_c ci_1(v_1) = { v_1 };
def_c ct_1() = { ci_1(x) };
def_c cf_1() = { ci_1(y) };
def_c ca_1() = {
  val_l i_1 = false;
  if (b != i_1) ct_1 else cf_1 };
val_l i_2 = false;
if (a != i_2) ca_1 else cf_1
```

which immediately applies continuation $\texttt{cf}_1$ if a is false.

Source Contexts

Common to the two problems: the translation could be better if we translate differently under different source 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) $[\![\cdot]\!]_N$ `C`, taking as before a term to translate and a context `C`, whose 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) $[\![\cdot]\!]_N$ `C`, taking as before a term to translate and a context `C`, whose hole must be plugged with a *name/symbol* bound to the value of the term.

2) $[\![\cdot]\!]_T$ `C`, taking a term to translate and a symbol `C` for the continuation. This continuation is to be applied to the value of the term.

A Better Translation

We split the single translation function into three separate ones:

1) $[\![\cdot]\!]_N$ `C`, taking as before a term to translate and a context `C`, whose hole must be plugged with a *name/symbol* bound to the value of the term.

2) $[\![\cdot]\!]_T$ `C`, taking a term to translate and a symbol `C` for the continuation. This continuation is to be applied to the value of the term.

3) $[\![\cdot]\!]_C$ `ct cf`, taking a term to translate and two parameterless continuations, `ct` and `cf`. The continuation `ct` is to be applied when the term evaluates to a true value, while the continuation `cf` is to be applied when it evaluates to a false value.

The Non-tail Translation

$[\![\cdot]\!]_N$ is called the **non-tail** translation as it is used in non-tail contexts. That is, when the work that has to be done once the term is evaluated is more complex than simply applying a continuation to the term's value.

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:

```
⟦ p(e₁, e₂, ...) ⟧_N C =
  ⟦e₁⟧_N(λv₁(⟦e₂⟧_N(λv₂...)));
  val_p n = p(v₁, v₂, ...);
  C[n]
  where p is not a logical primitive
```

The Tail Translation

The tail translation $[\![\cdot]\!]_T$ is used whenever the context passed to the simple translation has the form λv(c(v)). It gets as argument the name of the continuation c to which the value of expression should be applied.

For example, the previous translation of function definition:

$$[\![\ \texttt{def f}_1\texttt{(n}_{1,1}\texttt{: \_, ...) = e}_1\texttt{; ... ; e}\ ]\!]\ \texttt{C} =$$
$$\texttt{def}_f\ \texttt{f}_1\texttt{(}\underline{\texttt{c}}\texttt{, n}_{1,1}\texttt{, ...) = \{ }[\![\texttt{e}_1]\!](\lambda\texttt{v(c(v))) \};}$$
$$\texttt{def}_f\ \texttt{...;}$$
$$[\![\texttt{e}]\!]\ \texttt{C}$$

becomes:

$$[\![\ \texttt{def f}_1\texttt{(n}_{1,1}\texttt{: \_, ...) = e}_1\texttt{; def ... ; e}\ ]\!]_N\ \texttt{C} =$$
$$\texttt{def}_f\ \texttt{f}_1\texttt{(}\underline{\texttt{c}}\texttt{, n}_{1,1}\texttt{, ...) = \{ }[\![\texttt{e}_1]\!]_T\texttt{ c \};}$$
$$\texttt{def}_f\ \texttt{...;}$$
$$[\![\texttt{e}]\!]_N\ \texttt{C}$$

The Cond Translation (1)

The cond translation $[\![.]\!]_C$ is used whenever the term to translate is a condition to decide 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:

$$
\begin{array}{l}
[\![\ \texttt{if (e}_1\texttt{) e}_2\ \texttt{else e}_3\ ]\!]_N\ \texttt{C =} \\
\quad \texttt{def}_c\ \underline{\texttt{c}}\texttt{(}\underline{\texttt{r}}\texttt{) = \{ C[r] \};} \\
\quad \texttt{def}_c\ \underline{\texttt{ct}}\texttt{() = \{ } [\![\texttt{e}_2]\!]_T\ \texttt{c \};} \\
\quad \texttt{def}_c\ \underline{\texttt{cf}}\texttt{() = \{ } [\![\texttt{e}_3]\!]_T\ \texttt{c \};} \\
\quad [\![\texttt{e}_1]\!]_C\ \texttt{ct cf}
\end{array}
$$

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:

$[\![$ `if (e`$_1$`) false else true` $]\!]_C$ `ct cf =`
  $[\![$`e`$_1]\!]_C$ `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:

```
⟦ if (e₁) false else true ⟧_C ct cf =
  ⟦e₁⟧_C cf ct
```

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

```
⟦ if (e₁) e₂ else false ⟧_C ct cf =
  def_c c() = { ⟦e₂⟧_C ct cf };
  ⟦e₁⟧_C c cf;
```

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

The Cond Translation (3)

```
⟦ while (e₁) e₂; e₃ ⟧_N C =
  def_c loop(d) = {
    def_c c() = { ⟦e₃⟧_N C };
    def_c ct() = { ⟦e₂⟧_T loop };
    ⟦e₁⟧_C ct c
  };
  val_l d = ();
  loop(d)
```

Exercise

Translating the following program:

```scala
def f(g: () => Int) = 42 + g(); f
```

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
```

Other IRs

- Continuation-passing style (CPS)
- Administrative normal form (ANF)
- Register-transfer language (RTL) and control-flow graph (CFG)
- Static Single-Assignment (SSA) form
- Sea-of-nodes (graph-based) IRs

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

Register-transfer language (RTL):

- Most operations/instructions compute a function using virtual registers (i.e. variables),
- Operations store the result in another virtual register.

Example: `x ← y + z`: adding variables y and z, storing the result in x could be written.

Such instructions are sometimes called quadruples, because they typically have four components: the three variables (x, y and z here) and the operation (+ here).

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.

More precisely, there is an edge in the CFG from a node n_1 to a node n_2 if and only if the instruction of n_2 can be executed immediately after the instruction of n_1.

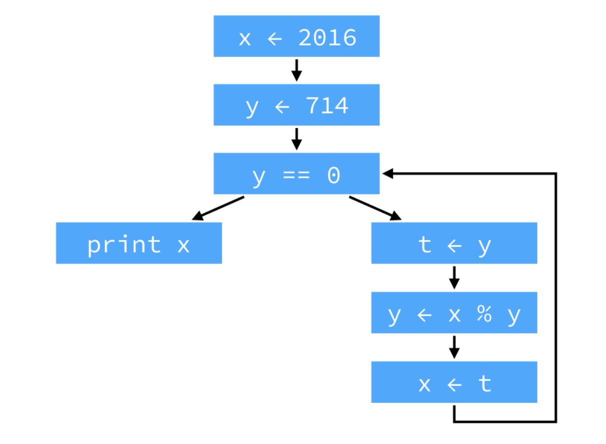
RTL/CFG

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.



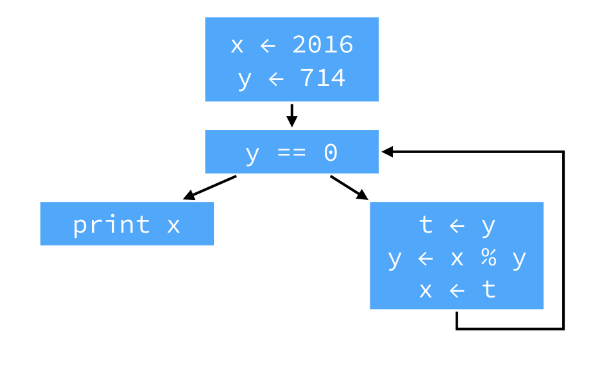
Basic Blocks

Basic block: a list of instructions that 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.



Summary

Today's lecture:

- Improved CPS translation
- RTL/CFG IR

Next time:

- More on SSA IR
- Value representation