

CS107: CPS Translation

Guannan Wei

guannan.wei@tufts.edu

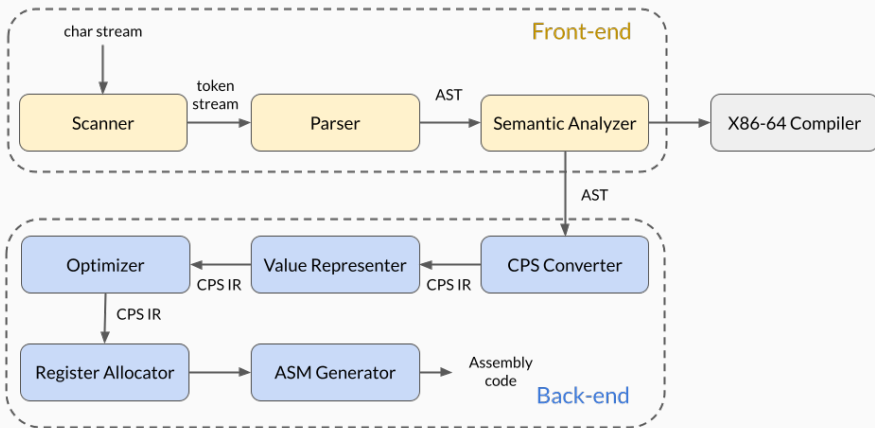
October 6, 2026

Fall 2026

Tufts University

- Project 3
- Upcoming schedule:
 - Oct 8: RTL/CFG/SSA IR (record video)
 - Oct 15: Industry guest lecture from Max Bernstein (Tufts alumnus!)

Recap



Recap

- Continuation: a function that represents the rest of the computation
- Continuation-passing style (CPS): a style of programming or IR where control flow is made explicit through continuations; all functions take an additional continuation argument.
- CPS translation: transforming a program into continuation-passing style

Why compiling to CPS?

- Continuation: a function that represents the rest of the computation
- Continuation-passing style (CPS): a style of programming or IR where control flow is made explicit through continuations; all functions take an additional continuation argument.
- CPS translation: transforming a program into continuation-passing style

Why compiling to CPS?

CPS provides a uniform representation of control flow, making it easier to reason about and optimize programs, especially for higher-order functional programs.

The translation function:

$\llbracket \cdot \rrbracket : \text{MSTree} \Rightarrow (\text{Symbol} \Rightarrow \text{CPSTree}) \Rightarrow \text{CPSTree}$

Example:

$\llbracket t \rrbracket c : \text{CPSTree}$
where $t : \text{MSTree}$ and $c : \text{Symbol} \Rightarrow \text{CPSTree}$

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

$\llbracket n \rrbracket C = C[n]$ where n is an identifier for immutable variable

$\llbracket l \rrbracket C = \text{val}_l \underline{n} = l; C[n]$ where l is a literal

$\llbracket \text{val } n_1 = e_1; e \rrbracket C =$
 $\llbracket e_1 \rrbracket (\lambda v (\text{val}_p n_1 = \text{id}(v); \llbracket e \rrbracket C))$

$\llbracket e(e_1, e_2, \dots) \rrbracket C =$
 $\llbracket e \rrbracket (\lambda v (\llbracket e_1 \rrbracket (\lambda v_1 (\llbracket e_2 \rrbracket (\lambda v_2 (\dots$
 $\text{def}_c \underline{c}(\underline{r}) = \{ C[r] \};$
 $v(\underline{c}, v_1, v_2 \dots))))))$

In which context should a complete program be translated?

The simplest answer is a context that halts execution with an exit code of 0 (no error), that is:

```
 $\lambda v(\text{val}_l\ z = 0; \text{halt}(z))$ 
```

An alternative would be to do something with the value v produced by the whole program, e.g. use it as the exit code instead of 0, print it, etc.

Example

Let k_0 be the initial context $\lambda v(\text{val}_l z = 0; \text{halt}(z))$.

```
[[ f(g(1, 2)) ]] k_0
{ f(g(1, 2)) is a function application }
= [[f]](\lambda v([g(1, 2)])(\lambda v_1
  def_c c_1(r_1) = { k_0[r_1] };
  v(c_1, v_1))))
= ???
```

Example

Let k_0 be the initial context $\lambda v(\text{val}_l z = 0; \text{halt}(z))$.

```
[[ f(g(1, 2)) ]] k0
{ f(g(1, 2)) is a function application }
= [[f]](λv([g(1, 2)](λv1
  defc c1(r1) = { k0[r1] };
  v(c1, v1))))
{ f is an immutable variable }
= [[g(1, 2)]](λv1
  defc 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]](λvg([1](λv1([2](λv2(
  defc c2(r2) = { k1[r2] };
  vg(c2, v1, v2))))))
{ g is an immutable variable }
```

Example

```
=  ⟦g⟧(λvg(⟦1⟧(λv1(⟦2⟧(λv2(  
    defc c2(r2) = { k1[r2] };  
    vg(c2, v1, v2))))))  
{ g is an immutable variable }  
=  ⟦1⟧(λv1(⟦2⟧(λv2(  
    defc c2(r2) = { k1[r2] };  
    g(c2, v1, v2))))  
{ 1 and 2 are literals }  
=  vall n1 = 1;  
    vall n2 = 2;  
    defc c2(r2) = { k1[r2] };  
    g(c2, n1, n2)  
{ inline and apply continuation remembered as k1 }  
=  vall n1 = 1;  
    vall n2 = 2;  
    defc c2(r2) = {  
        defc c1(r1) = { k0[r1] }; f(c1, r2)  
    };  
    g(c2, n1, n2)
```

Example

```
{ inline and apply continuation remembered as  $k_0$  }  
=  
  vall n1 = 1;  
  vall n2 = 2;  
  defc c2(r2) = {  
    defc c1(r1) = { vall z = 0; halt(z) }  
    f(c1, r2)  
  };  
  g(c2, n1, n2)  
{ we are done! }
```

Remaining CPS Translation: Function Definitions

```
[[ def f1(n1,1: -, ...) = e1; def ... ; e ] C =  
  deff f1(c, n1,1, ...) = {  
    [[e1]](λv(c(v)))  
  };  
  deff ...;  
  [[e]] C
```

Remaining CPS Translation: Function Definitions

Consider the empty context: $k_0 = \lambda v.v$:

```
[[ def f(g: () => Int) = g(); f ]] k0
= deff f(c, g) = {
  [[g()]](λv(c(v)))
};
[[f]] k0
= { intermediate steps ... }
= deff f(c, g) = {
  defc j(r) = { c(r) };
  g(j)
};
f
```

Remaining CPS Translation: Mutable Variables and Assignments

```
[[ var n1 = e1; e ]] C =  
  vall s = 1;  
  valp n1 = block-alloc-var(s);  
  vall z = 0;  
  [[e1]](λv (valp d = block-set(n1, z, v); [[e]] C))
```

```
[[n]] C =  
  vall z = 0;  
  valp v = block-get(n, z); C[v]  
where n is an identifier for mutable variable
```

```
[[ n1 = e1 ]] C =  
  vall z = 0;  
  [[e1]](λv (valp d = block-set(n1, z, v); C[v] ))
```

Remaining CPS Translation: If

```
[[ if (p(e1, ...)) e2 else e3 ] C =  
  defc c(r) = { C[r] };  
  defc ct() = { [[e2]](λv2(c(v2))) };  
  defc cf() = { [[e3]](λv3(c(v3))) };  
  [[e1]](λv1(... (if (p(v1 ...)) ct else cf)))  
  where p is a logical primitive
```

```
[[ if (e1) e2 else e3 ] C =  
  defc c(r) = { C[r] };  
  defc ct() = { [[e2]](λv2(c(v2))) };  
  defc cf() = { [[e3]](λv3(c(v3))) };  
  vall f = false;  
  [[e1]](λv1(if (v1 ≠ f) ct else cf))
```

Remaining CPS Translation: While and Logical Primitives

```
[[ while (e1) e2; e3 ] C =  
  defc loop() = {  
    defc c() = { [[e3] C ];  
    defc ct() = { [[e2](λv(loop())) ];  
    vall f = false;  
    [[e1](λv(if (v ≠ f) ct else c))  
  };  
  loop()
```

```
[[ p(e1, e2, ...) ] C =  
  [[ if (p(e1, e2, ...)) true else false ] C  
  where p is a logical primitive
```

```
[[ p(e1, e2, ...) ] C =  
  left as an exercise  
  where p is not a logical primitive
```

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.

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