CS107: Closure Conversion and Optimization

Guannan Wei
guannan.wei@tufts.edu

October 29, 2026

Fall 2026

Tufts University

A Simple Translation

For function definition:

- generate a "wrapper" function that transforms the function body
  - replace the free variables with accesses to the environment
- generate a sequence `block-alloc` / `block-set` to build the closure

For function application:

- extract the code pointer from the closure and pass the closure

Function Definition

```
⟦ def_f f_1(c_1, n_1,1, ...) = { e_1 }; def_f f_2...; e ⟧ =
  def_f w_1(c_1, env_1, n_1,1, ...) = {
    val_p v_1 = block-get(env_1, 1);
    ...
    ⟦e_1⟧[f_1 ↦ env_1, FV_1(0) ↦ v_1, ... ]
  };
  def_f w_2(...) =  ... ;
  val_p f_1 = block-alloc-202(|FV_1| + 1);
  val_p f_2 = ...;
  val_p t_1,1 = block-set(f_1, 0, w_1);
  val_p t_1,2 = block-set(f_1, 1, FV_1(0));
  ...;
  val_p t_2,1 = block-set(f_2, 0, w_2);
  ...;
  ⟦e⟧
```

where FV_i is an (arbitrary) ordering of the set $F[e_i] \setminus \{ f_i, n_{i,1}, \ldots \}$

Function Application

Function application has to be transformed in order to extract the code pointer from the closure and pass the closure as the first argument after the return continuation:

$$[\![\ \texttt{n(nc, } n_1\texttt{, ...)}\ ]\!] =$$
$$\texttt{val}_p\ \texttt{f = block-get(n, 0);}$$
$$\texttt{f(nc, n, } n_1\texttt{, ...)}$$

Exercise

How does the closure conversion phase translate the following MiniScala/CPS version of makeAdder? (Only the closure conversion):

```
val inc = makeAdder(1); inc(42) // MiniScala source code
```

CPS-translated code:

```
def_f makeAdder(rc1, x) = {
  def_f anon1(rc2, y) = { val_l v = x + y; rc2(v) };
  rc1(anon1)
};
val_l i1 = 1;
def_c rc4(r1) = {
  val_p inc = id(r1);
  val_l i2 = 42;
  def_c rc3(r2) = { val_l z = 0; z }; // initial context
  inc(rc3, i2)
};
makeAdder(rc4, i1)
```

Solution - Function Definition

CPS and closure converted code:

```
def_f wMakeAdder(rc1, env1, x) = { // CPS with closures converted
  def_f wAnon1(rc2, env2, y) = {
    val_l fv1 = block-get(env2, 1);
    val_l v = fv1 + y;
    rc2(v)
  };
  val_p anon1 = block-alloc-202(2);       // allocate closure (size 2)
  val_p t1 = block-set(anon1, 0, wAnon1); // set code pointer
  val_p t2 = block-set(anon1, 1, x);      // set free variable x
  rc1(anon1)
};
```

to be continue...

Solution - Function Application

```
val_p makeAdder = block-alloc-202(1);
val_p t3 = block-set(makeAdder, 0, wMakeAdder);
val_l i1 = 1;
def_c rc4(r1) = {
  val_p inc = id(r1);
  val_l i2 = 42;
  def_c rc3(r2) = { val_l z = 0; z }; // initial context
  val_p wInc = block-get(inc, 0);
  wInc(rc3, inc, i2)
};
val_p f = block-get(makeAdder, 0);
f(rc4, makeAdder, i1)
```

Improving CPS Closure Conversion

The translation just presented is suboptimal in two respects:

1) it always creates closures, even for functions that are never used as values (i.e. only applied in the defining scope),
2) it always performs calls through the closure, thereby making all calls indirect.

We can either (1) optimize the result of the translation, or (2) improve the translation itself to avoid these problems in the first place.

Translation Inefficiencies

The inefficiency of the simple translation can be observed on the `makeAdder` example:

```
def makeAdder(x: Int) = (y: Int) => x + y;
makeAdder(1)
```

The simple translation creates a closure for both functions. However, the outer one is **closed** and never used as a value, therefore no closure needs to be created for it.

Notice that even if the outer function escaped (i.e. used as a value) and needed a closure, the call to it could avoid fetching its code pointer from the closure, as here it is a known function explicitly defined with a name.

Improved Translation

The simple translation translates a source function into one target function and one closure.

The improved translation splits the target function in two:

1) the **wrapper**, which simply extracts the free variables from the environment and passes them as additional arguments to the worker,
2) the **worker**, which takes the free variables as additional arguments and does the real work.

The wrapper is stored in the closure.

If we can figure out the arguments of the source function, then we can directly call the worker, without going through the closure.

Improved Function Definition

```
⟦ def_f f_1(c_1, n_1, ...) = { e_1 }; def_f f_2...; e ⟧ =
  def_f w_1(c_1, n_1, ..., u_1, ...) = { // the worker function
    ⟦e_1⟧[FV_1(0) ↦ u_1, ... ]
  };
  def_f s_1(sc1, env_1, sn_1, ...) = { // the wrapper function
    val_p v_1 = block-get(env_1, 1);
    ...;
    w_1(sc_1, sn_1, ..., v_1, ...);
  };
  def_f w_2(...) = ...;
  def_f s_2(...) = ...;
  val_p f_1 = block-alloc-202(|FV_1| + 1);
  ...;
  val_p t_1,1 = block-set(f_1, 0, s_1);
  val_p t_1,2 = block-set(f_1, 1, FV_1(0));
  ...;
  ⟦e⟧
```

Improved Function Application

When translating function application, if the function being applied is known (i.e. is bound by an enclosing `LetRec`), the worker of that function can be used directly, without going through the closure:

```
⟦ n(n_c, n_1, ...) ⟧ =
  n_w(n_c, n_1, ..., FV_n(0), ...)
  if n is known, with worker n_w
```

otherwise, the closure has to be used, as in the simple translation:

```
⟦ n(nc, n_1, ...) ⟧ =
  val_p f = block-get(n, 0);
  f(nc, n, n_1, ...)
  otherwise
```

Exercise

How this new rules would translate the previous `makeAdder` example?

```
def_f makeAdder(rc1, x) = { // CPS code
  def_f anon1(rc2, y) = { val_l v = x + y; rc2(v) };
  rc1(anon1)
};
def_c rc4(r1) = {
  val_p inc = id(r1);
  def_c rc3(r2) = { val_l z = 0; z }; // initial context
  inc(rc3, 42)
};
makeAdder(rc4, 1)
```

(Note that the example inlines literal values, but the translation should not.)

Exercise Solution

```
def_f makeAdderWorker(rc1, x) = {
  val_p anon1 = block-alloc-202(2);
  val_p t7 = block-set(anon1, 0, anon1Wrapper);
  val_p t5 = block-set(anon1, 1, x);
  rc1(anon1)
};
def_f anon1Worker(rc2, y, x) = {
  val_p t1 = x + y;
  rc2(t1)
};
def_f anon1Wrapper(ret1, env1, y) = {
  val_p x = block-get(env1, 1);
  anon1Worker(ret1, y, x)
};
```

to be continue...

Exercise Solution

```
def_c rc4(r1) = {
  val_p inc = id(r_1);
  def_c rc3(r2) = {
    val_l z = 0;
    halt(z)
  };
  val_p t_12 = block-get(inc, 0);
  t_12(rc3, inc, 42)
};
makeAdderWorker(rc4, 1)
```

Note: (1) the translation doesn't take care of integer tagging, but the actual translation should; (2) we can directly call the worker of `makeAdder` because it is a known function; (3) but we need to go through the closure for `anon1`.

Free Variables

The improved translation makes the computation of free variables slightly more difficult.

That's because when a function `f` calls a known function `g` through its worker, it has to pass it its free variables as additional arguments.

The free variables of `g` now become free variables of `f`. These new free variables must be added to `f`'s arguments, which impacts its callers - which could include `g` if the two are mutually-recursive. And so on...

Implementation hint: need to compute the fixed point of the free variables of functions.

Function Hoisting

After closure conversion, all functions in the program are closed. Therefore, it is possible to hoist them all to a single outer lets.

Once this is done, the program has the following simple form:

```
def_f f_1(...) ...; ...; def_f f_n(...) ...; main
```

where the main program code does not contain any function definition (LetRec expression).

Hoisting functions to the top level simplifies the shape of the program and can make the job of later phases - e.g. assembly code generation - easier.

CPS Hoisting (1)

Hoisting: recursively pops up all local function definitions to the top level.

$$
\begin{aligned}
&[\![\ \texttt{val}_l\ \texttt{n = l; e}\ ]\!] = \\
&\quad \texttt{def}_f\ \texttt{f}_1\texttt{...; ...; def}_f\texttt{f}_n\texttt{...;} \\
&\quad \texttt{val}_l\ \texttt{n = l; e}_h \\
&\quad \texttt{where}\ [\![\texttt{e}]\!] = \texttt{def}_f\texttt{f}_1\texttt{...; ...; def}_f\texttt{f}_n\texttt{...; e}_h
\end{aligned}
$$

$$
\begin{aligned}
&[\![\ \texttt{val}_p\ \texttt{n = p(n}_1\texttt{, ...); e}\ ]\!] = \\
&\quad \texttt{def}_f\ \texttt{f}_1\texttt{...; ...; def}_f\texttt{f}_n\texttt{...;} \\
&\quad \texttt{val}_p\ \texttt{n = p(n}_1\texttt{, ...); e}_h \\
&\quad \texttt{where}\ [\![\texttt{e}]\!] = \texttt{def}_f\texttt{f}_1\texttt{...; ...; def}_f\texttt{f}_n\texttt{...; e}_h
\end{aligned}
$$

CPS Hoisting (2)

$[\![$ **def**$_c$ c1(n$_{1,1}$, ...) = { e1 }; **def**$_c$...; e $]\!]$
  def$_f$ fs$_{1,1}$...; **def**$_f$fs$_{1,m_1}$; **def**$_f$fs$_{2,1}$...; ...; **def**$_f$fs$_1$...; **def**$_f$fs$_n$...;
  def$_c$ c1(n$_{1,1}$, ...) = { e$_{h_1}$ }; **def**$_c$...; e$_h$
  where $[\![$e$_i]\!]$ = **def**$_f$fs$_{i,1}$...; ...; **def**$_f$fs$_{i,m_i}$...; e$_{h_i}$
    and $[\![$e$]\!]$ = **def**$_f$fs$_1$...; ...; **def**$_f$fs$_n$...; e$_h$

$[\![$ **def**$_f$ f$_1$(n$_{1,1}$, ...) = { e1 }; **def**$_f$...; e $]\!]$
  def$_f$ fs$_{1,1}$...; **def**$_f$fs$_{1,m_1}$; **def**$_f$fs$_{2,1}$...; ...; **def**$_f$fs$_1$...; **def**$_f$fs$_n$...;
  def$_f$ f$_1$(n$_{1,1}$, ...) = { e$_{h_1}$ }; **def**$_f$...; e$_h$
  where $[\![$e$_i]\!]$ = **def**$_f$fs$_{i,1}$...; ...; **def**$_f$fs$_{i,m_i}$...; e$_{h_i}$
    and $[\![$e$]\!]$ = **def**$_f$fs$_1$...; ...; **def**$_f$fs$_n$...; e$_h$

$[\![$e$]\!]$ when e is any other kind of expression =
  $\emptyset$; e

Optimization

Optimization

The goal of optimization is to rewrite a program being compiled to a new program that is simultaneously:

- behaviorally **equivalent** to the original one, and
- **better** in some respect - e.g., faster, smaller, more energy-efficient, etc.

Optimization

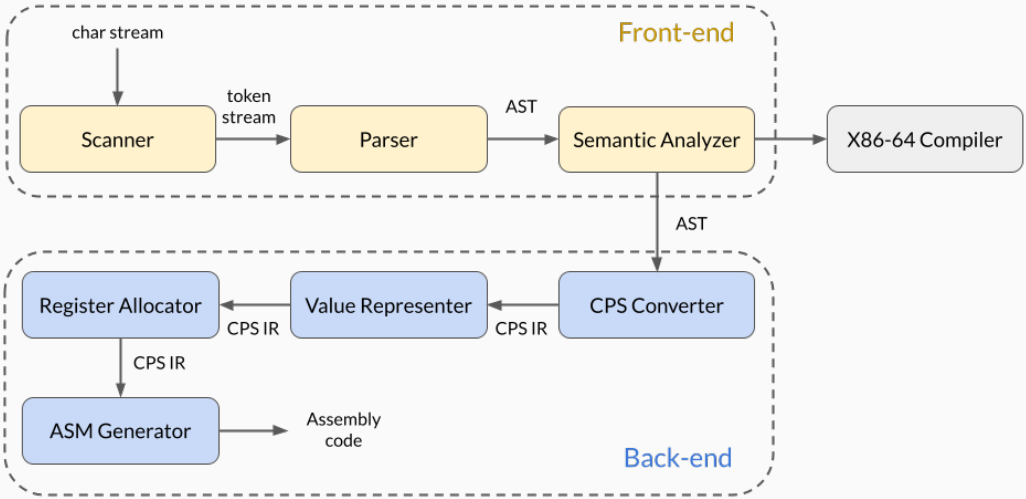
The goal of optimization is to rewrite a program being compiled to a new program that is simultaneously:

- behaviorally **equivalent** to the original one, and
- **better** in some respect - e.g., faster, smaller, more energy-efficient, etc.

Optimizations can be broadly split into two classes:

- **machine-independent optimizations**, e.g. constant folding or dead code elimination, which are high-level and do not depend on the target architecture,
- **machine-dependent optimizations**, e.g. register allocation or instruction scheduling, which are low-level and depend on the target architecture.

Compiler Architecture



Rewriting Optimizations

In this lecture, we will examine a simple set of machine independent **rewriting optimizations**.

Most of them are relatively simple rewrite rules that transform the source program to a shorter, equivalent one.

Rewriting Optimizations

- Constant propagation
- Constant folding
- Dead-code elimination
- Common subexpression elimination
- Eta reduction
- Algebraic laws
- Function inlining
- Contification

The Importance Of IRs

The intermediate representation (IR) on which rewriting optimizations are performed can have a dramatic impact on their ease of implementation.

A rewriting optimization generally works in two steps:

1) the program is analyzed to find rewrite opportunities,
2) the program is rewritten based on the opportunities identified in the first step.

A good IR should make both steps as easy as possible. The following examples illustrate the importance of using the right IR.

Case 1: Constant Propagation

To illustrate the impact of IR on the analysis step, consider the following program fragment in some imaginary IR:

$x \leftarrow 7$

Is it legal to perform constant propagation and blindly replace all later occurrences of x by 7?

Case 1: Constant Propagation

To illustrate the impact of IR on the analysis step, consider the following program fragment in some imaginary IR:

$x \leftarrow 7$

Is it legal to perform constant propagation and blindly replace all later occurrences of x by 7?

It depends on the IR:

- If multiple assignments to the same variable are allowed, then additional (data-flow) analyses are required to answer the question, as x might be re-assigned later.
- However, if multiple assignments to the same variable are prohibited, then yes, all occurrences of x can be unconditionally replaced by 7!

Other Simple Optimizations

Apart from constant propagation, many simple optimizations are made hard by the presence of multiple assignments to a single variable:

- common-subexpression elimination, which consists in avoiding the repeated evaluation of expressions,
- (simple) dead code elimination, which consists in removing assignments to variables whose value is not used later,
- etc.

In all cases, analyses are required to distinguish the various "versions" of a variable that appear in the program.

Conclusion: a good IR should not allow multiple assignments to a variable!

Case 2: Inlining

Inlining (or in-line expansion) consists in replacing a call to a function by a copy of the function body, with parameters replaced by the actual arguments. It is a very important compiler optimization, as it often opens the door to further optimizations.

Some aspects of the IR can have an important impact on the implementation of inlining. To illustrate this, let us examine some problems that can occur when performing inlining directly on the AST - a choice that might seem reasonable at first sight.

Naive Inlining: Problem #1

```
def printReturn(x: Int) = { printInt(x); x };
def twice(y: Int) = y + y;
def f(z: Int) = twice(printReturn(z));
```

Incorrect inlining of `twice` in `f`:

```
def f(z: Int) = printReturn(z) + printReturn(z)
```

Naive Inlining: Problem #1

```
def printReturn(x: Int) = { printInt(x); x };
def twice(y: Int) = y + y;
def f(z: Int) = twice(printReturn(z));
```

Incorrect inlining of `twice` in `f`:

```
def f(z: Int) = printReturn(z) + printReturn(z)
```

`z` is printed twice!

Possible solution: bind actual parameters to variables (using a `Let`) to ensure that they are evaluated at ***most*** once.

Naive Inlining: Problem #2

```
def first[T, U](t: T, u: U) = t;
def printReturn(z: Int) = first[Int, Unit](z, printInt(z));
```

Incorrect inlining of `first` in `printReturn`:

```
def printReturn(z: Int) = z
```

Naive Inlining: Problem #2

```
def first[T, U](t: T, u: U) = t;
def printReturn(z: Int) = first[Int, Unit](z, printInt(z));
```

Incorrect inlining of `first` in `printReturn`:

```
def printReturn(z: Int) = z
```

`z` is not printed at all!

Possible solution: bind actual parameters to variables (using a `Let`) to ensure that they are evaluated at ***least*** once.

Easy Inlining

The two pitfalls presented earlier can be avoided by bindings actual arguments to variables (using a `Let`) before using them in the body of the inlined function.

However, a properly-designed IR can also avoid the problems altogether by ensuring that actual parameters are always atoms, i.e. variables or constants.

Conclusion: a good IR should only allow atomic arguments to functions.

IR Comparison

Using the two very basic criterions we identified, we can evaluate the various classes of IRs we have seen:

- standard RTL/CFG is bad in that its variables are mutable; however, it allows only atoms as function arguments, which is good,
- RTL/CFG in SSA form, MiniScala/CPS and similar functional IRs are good in that their variables are immutable, and they only allow atoms as function arguments.