CS107: Closure Conversion

Guannan Wei
guannan.wei@tufts.edu

February 24, 2026

Spring 2026

Tufts University

Logistics

- Midterm exam: March 5
- Problem set for practice will be released later
- In-class review next Tuesday (March 3)

Value Representation for Functions

- Simply using code pointers is not sufficient to represent higher-order functions. Why?

Value Representation for Functions

- Simply using code pointers is not sufficient to represent higher-order functions. Why?
- Functions can be nested and escape from the defining scope. They can have free variables, which are variables defined in an enclosing scope. We need to know the value for these variables when the function is called.

```
def makeAdder(x: Int) = (y: Int) => x + y;
                       |------------------|
                        anonymous function w/ free variable x
|-----------------------------------------|
  makeAdder is a closed function
```

Value Representation for Functions

Closure conversion: transforms a program in which functions can have free variables into an equivalent one that is closed (i.e. no free variables).

After the conversion, a function is represented as a pair of **a code pointer** and **an environment**.

An environment is (again) a data structure containing the values of the free variables of the function.

A Bit of History

The notion of closure was introduced by Peter Landin in his seminal paper “The Mechanical Evaluation of Expressions”, 1964.

Scheme implemented lexical scope for Lisp using closures in 1975.

Closing Example

Source program:

```
def makeAdder(x: Int) = (y: Int) => x + y;
makeAdder(42)
```

In the λ-calculus, "execution'' is expressed by substitution (i.e., β-reduction):

```
makeAdder(42) = ((y: Int) => x + y)[x -> 42] = (y: Int) => 42 + y
```

Closing Example

Source program:

```
def makeAdder(x: Int) = (y: Int) => x + y;
makeAdder(42)
```

In the λ-calculus, "execution'' is expressed by substitution (i.e., β-reduction):

```
makeAdder(42) = ((y: Int) => x + y)[x -> 42] = (y: Int) => 42 + y
```

In reality, we cannot rely on substitution for efficient execution. But we still need a way to remember the mapping from `x` to its value 42 when producing the function value `(y: Int) => x + y` from `makeAdder (42)`.

Closing Example

Assuming the existence of abstract functions `closureMake` and `closureGet`.

```
def makeAdder(x: Int) = (y: Int) => x + y;
makeAdder(42)
```

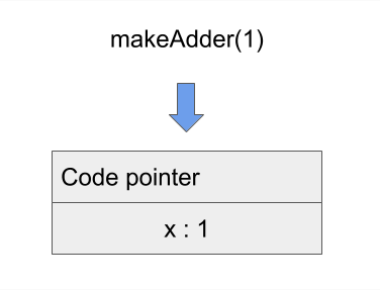
Functions are transformed to take an additional "environment/closure" argument:

```
val makeAdder =
  closureMake((env$_1$: Env, x: Int) =>
    closureMake((env$_2$: Env, y: Int) => closureGet(env$_2$, 1) + y, List(x)),
    Nil
  )
closureGet(makeAdder, 0)(makeAdder, 42)
```

- Free variables are accessed by looking them up in the environment.
- Function call needs to extract the code pointer from the closure and pass the closure as an additional argument.

Flat Closures

In flat (or one-block) closures, the environment is “inlined” into the closure itself, instead of being referred from it. The closure itself plays the role of the environment.



In other words, the environment is the closure, represented as a block containing the code pointer (at index 0) and the values of the free variables.

Recursive Closures

Recursive functions need access to their own closure. For example:

```
def f(l: List[Int]) = { ...; map(f, l); ... }; ...
```

Several techniques can be used to give a closure access to itself:

- the closure (here f) can be treated as a free variable, and put in its own environment, leading to a cyclic closure,
- the closure can be rebuilt from scratch,
- with flat closures, the environment is the closure, and can be reused directly.

Mutually-recursive Closures

Mutually-recursive functions all need access to the closures of all the functions in the definition.

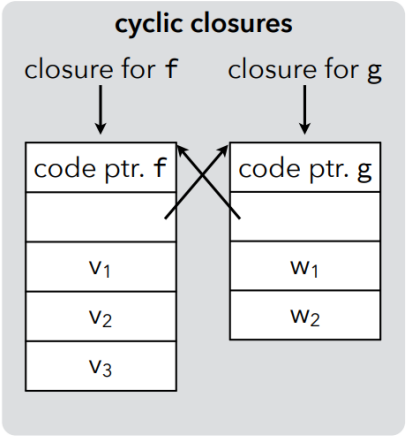
For example, in the following program, `f` needs access to the closure of `g`, and the other way around:

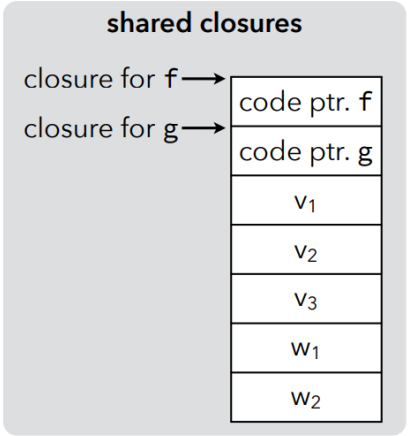
```
def f(l: List[Int]) = { ...; compose(f, g); ...};
def g(l: List[Int]) = { ...; compose(g, f); ...};
```

Solutions:

- use cyclic closures, or
- share a single closure with interior pointers - but note that the resulting interior pointers make the job of the garbage collector harder.

Mutually-recursive Closures





MiniScala/CPS - Closure Conversion

In our compiler, we represent functions using flat closures.

Flat closures are simply blocks tagged with a tag reserved for functions - we choose 202.

The first element of the block contains the code pointer while the other elements contain the environment of the closure (which could be empty).

MiniScala/CPS - Closure Conversion

In our MiniScala compiler, closure conversion is not a separate phase. Rather, it is the part of the values representation phase that takes care of representing function values.

Closure conversion is therefore specified exactly like the values representation phase.

MiniScala/CPS - Free Variables

The F⟦·⟧ function computes the *set of free variables* of a CPS term:

```
F⟦ val_l n = l; e ⟧ = F⟦e⟧ \ { n }
F⟦ val_p n = p(n_1, ...); e ⟧ =
  (F⟦e⟧ \ { n }) ∪ { n_1, ... }
F⟦ def_c n(n_1, ...) = { e_1 }; ...; e ⟧ =
  F⟦e⟧ ∪ (F⟦e_1⟧ \ { n_1, ... }) ∪ ...
F⟦ def_f f_1(n_1, ...) = { e_1 }; ...; e ⟧ =
  (F⟦e⟧ ∪ (F⟦e_1⟧ \ { n_{1,1}, ... }) ∪ ...) \ { f_1, ... }
F⟦ c(n_1, ...) ⟧ = { n_1, ... }
F⟦ f(c, n_1, ...) ⟧ = { f, n_1, ... }
F⟦ if (p(n_1, ...)) ct else cf ⟧ = { n_1, ... }
```

Note: CPS scoping ensure that continuation variables are never free in a function, so we ignore them.

Notation

To simplify some of the following slides, we assume that integer literals can be used as arguments of primitives. For example, we write:

```
val_p n = block-get(b, 1); ...
```

instead of

```
val_l c_1 = 1;
val_p n = block-get(b, c_1); ...
```

But be careful that the actual code still requires intermediate bindings!

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 $\mathsf{F}[\mathsf{e}_i] \setminus \{ \mathsf{f}_i, \mathsf{n}_{i,1}, \ldots \}$

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:

```
⟦ n(nc, n1, ...) ⟧ =
  valp f = block-get(n, 0);
  f(nc, n, n1, ...)
```

Exercise

We have seen two techniques to represent the closures of mutually-recursive functions: cyclic closures and shared closures.

Which of these two techniques does our transformation use (explain)?

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 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),
2) it always performs calls through the closure, thereby making all calls indirect.

These problems could be solved by later optimizations or by a better version of the translation sketched below.

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)
```

Applied to the CPS version of that program, 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. was 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.

The worker is used directly whenever the source function is applied to arguments instead of being used as a value.

Improved Function Definition

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

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`, 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...

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 (letf expression). Does that remind you of something?

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)

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

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

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

Closures And Objects

There is a strong similarity between closures and objects: closures can be seen as objects with a single method, containing the code of the closure, and a set of fields (i.e. the environment).

Anonymous nested classes can therefore be used to simulate closures, but the syntax for them is often too heavyweight to be used often.

Object-oriented languages like Java, C#, etc. now support lambda expressions, which are translated to nested classes.

Summary

- Value representation for functions
- Closure conversion
  - Flat closures
  - A simple translation and an improved one

You will be implementing value representations and closure conversion in Project 5!