CS107: Closure Conversion

Guannan Wei
guannan.wei@tufts.edu

October 27, 2026

Fall 2026

Tufts University

Logistics

- Project 3 grade released
- Midterm grade released
- Project 5 released

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
```

Free Variables

The free variables of a function are the variables that are used but not defined in that function - i.e. they are defined in some enclosing scope.

The `makeAdder` example contains two functions:

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

The outer one does not have any free variable: it is a closed function. The inner one has a single free variable: `x`

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**, i.e., closure.

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

Closing Functions

Functions are closed by adding a parameter representing the environment, and using it in the function's body to access free variables.

Function abstraction/definition and application must of course be adapted accordingly:

- abstraction/definition must create and initialize the closure,
- application must pass the environment as an additional parameter.

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 if we pretend `((y: Int) => x + y)[x -> 42]` is not a substitution, but a data structure, then we have a closure tracking the value of `x`.

Closing Example

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

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

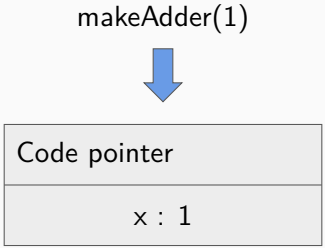
Functions are transformed to take an additional "environment/closure" argument:

```
val makeAdder =
  closureMake((env1: Env, x: Int) =>
    closureMake((env2: Env, y: Int) => closureGet(env2, 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 environment is the closure, represented as a block containing the code pointer (at index 0) and the values of the free variables.



- No sharing between multiple closures: each closure has its own copy of the environment.

Recursive Closures

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

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

With flat closures, the environment is the closure, and can be reused directly.

Recursive Closures

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

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

With flat closures, the environment is the closure, and can be reused directly.

Several other 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.

Mutually-recursive Closures

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

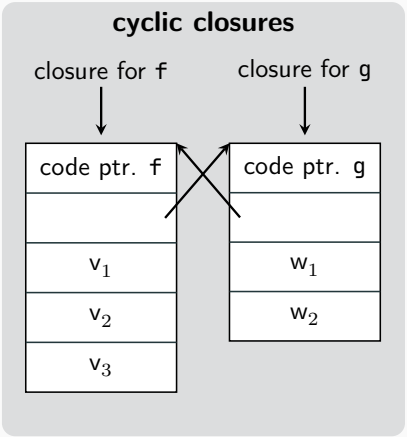
For example, in the following program, `f` needs access to the closure of `g`, and the other way around:

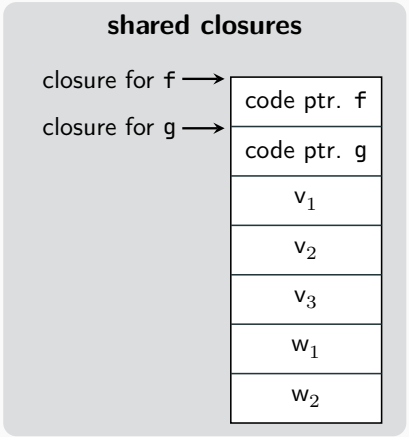
```
def f(l: List[Int]) = { ...; compose(f, g); ...};
def g(l: List[Int]) = { ...; compose(g, f); ...};
```

Possible 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 MiniScala/CPS compiler,

- Closure conversion is the part of the values representation phase that takes care of representing function values.
- 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).
- Mutually recursive functions are represented as cyclic closures.

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_1,1, ...) = { 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

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)
```