PURDUE
UNIVERSITY

Closure Conversion

Compilers: Principles And Practice

Tiark Rompf

What did we learn in the last class?

High-order Function

A higher-order function (HOF) is a function that either:

- takes another function as argument, or
- returns a function.

Many languages offer higher-order functions, but not all provide the same power...

HOFs in C

In C, it is possible to pass a function as an argument, and to return a function as a result.

However, C functions cannot be nested: they must all appear at the top level. This severely restricts their usefulness, but greatly simplifies their implementation - they can be represented as simple code pointers.

HOFs in Functional Languages

In functional languages - Scala, OCaml, Haskell, etc. - functions can be nested, and they can survive the scope that defined them.

This is very powerful as it permits the definition of functions that return "new" functions - e.g. functional composition.

However, as we will see, it also complicates the representation of functions, as simple code pointers are no longer sufficient.

HOF Example

To illustrate the issues related to the representation of functions in a functional language, we will use the following MiniScala example:

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

val increment = makeAdder(1);
increment(41); // 42

val decrement = makeAdder(-1);
decrement(42); // 41
```

Adder Maker In Scala

To see how closures are handled in Scala, let's look at how the translation of the Scala equivalent of the `makeAdder` function:

```
def makeAdder(x: Int): Int => Int =
  (y: Int) => x + y
val increment = makeAdder(1)
increment(41)
```

Translated Adder

```
class Anon extends Function1[Int,Int] {
  private val x: Int;
  def this(x: Int) = { this.x = x }
  def apply(y: Int): Int = this.x + y
}

def makeAdder(x: Int): Function1[Int,Int] = new Anon(x)
val increment = makeAdder(1)
increment.apply(41)
```

Translated Adder

(Hoisted) closure class: the code is in the `apply` method, the environment in the object itself: it's a flat closure.

```
class Anon extends Function1[Int,Int] {
  private val x: Int;
  def this(x: Int) = { this.x = x }
  def apply(y: Int): Int = this.x + y
}

def makeAdder(x: Int): Function1[Int,Int] =
  new Anon(x)
val increment = makeAdder(1)
increment.apply(41)
```

env. initialization

env. extraction

closure creation

closure application (the closure is passed implicitly as `this`)

Representing Adders

To represent the functions returned by `makeAdder`, there are basically two choices:

1. Use simple code pointers. Unfortunately, this implies run-time code generation, as each function returned by `makeAdder` is different!
2. Find another representation for functions, which does not depend on run-time code generation.

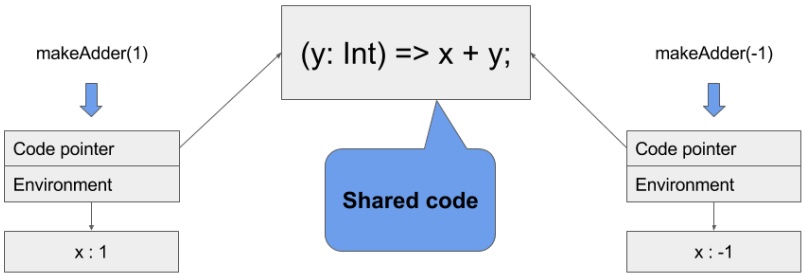
Closures

To adequately represent the functions returned by `makeAdder`, their code pointer must be augmented with the value of `x`.

Such a combination of a code pointer and an environment giving the values of the free variable(s) - here `x` - is called a closure.

The name refers to the fact that the pair composed of the code pointer and the environment is closed, i.e. selfcontained.

Closures



The code of the closure must be evaluated in its environment, so that x is “known”

Introducing Closures

Using closures instead of function pointers changes the way functions are manipulated at run time:

- function abstraction builds and returns a closure instead of a simple code pointer,
- function application extracts the code pointer from the closure, and invokes it with the environment as an additional argument.

Representing Closures

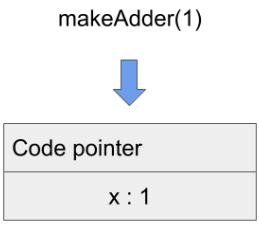
During function application, nothing is known about the closure being called - it can be any closure in the program.

The code pointer must therefore be at a known and constant location so that it can be extracted.

The values contained in the environment, however, are not used during application itself: they will only be accessed by the function body. This provides some freedom to place them.

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.



Exercise

Given the following MiniScala composition function:

```
def compose(f: Int => Int, g: Int => Int) =
  (x: Int) => f(g(x))
```

draw the flat closure returned by the application `compose(succ, twice)` assuming that succ and twice are two functions defined in an enclosing scope.

Closure Conversion

In a compiler, closures can be implemented by a simplification phase, called closure conversion.

Closure conversion transforms a program in which functions can have free variables into an equivalent one containing only closed functions.

The output of closure conversion is therefore a program in which functions can be represented as code pointers.

Closure Conversion

Closure conversion is nothing more than values representation for functions: it encodes the high-level notion of functions of the source language using the lowlevel concepts of the target language - in this case heapallocated blocks and code pointers.

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`

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 and application must of course be adapted accordingly:

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

Closing Example

Assuming the existence of abstract `closureMake` and `closureGet` functions, a closure conversion phase could transform the `makeAdder` example:

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

into:

```
val makeAdder = {
  closureMake((env1: Env, x: Int) =>
    closureMake((env2: Env, y: Int) => closureGet(env2, 1) + y,
    List(x)), // Value to be added into the environment
    Nil
  )
};
closureGet(makeAdder, 0)(makeAdder, 42)
```

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.

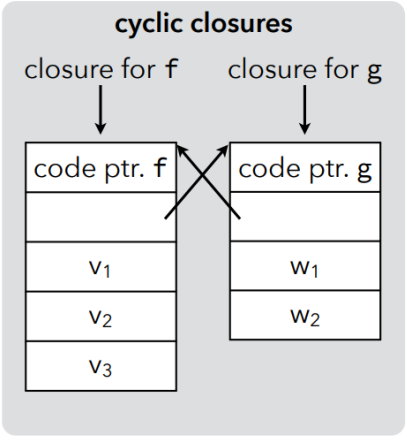
For example, in the following program, `f` needs access to the closure of `g`, and the other way around:

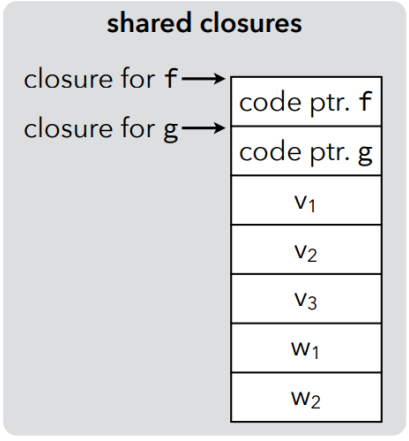
```
def f(l: List[Int]) = { ...; compose(f, g); ...};
def g(l: List[Int]) = { ...; compose(g, f); ...};
```

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





CPS/MiniScala - Closure Conversion

In the L3 compiler, we represent L3 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 - if any - contain the environment of the closure.

CPS/MiniScala - Closure Conversion

In the L3 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.

CPS/MiniScala - Free Variables

The F function computes the free variables of a CPS term:

$$
\begin{aligned}
&F[\![\ \texttt{val}_l\ n = l\ ;\ e\ ]\!] = F\ [\![e]\!] \setminus \{\ n\ \} \\
&F[\![\ \texttt{val}_p\ n = p(n_1, \ldots);\ e\ ]\!] = \\
&\quad (F[\![e]\!] \setminus \{\ n\ \}) \cup \{\ n_1, \ldots\ \} \\
&F[\![\ \texttt{def}_c\ n(n_1, \ldots) = \{\ e_1\ \};\ \ldots;\ e\ ]\!] = \\
&\quad F[\![e]\!] \cup (F[\![e_1]\!] \setminus \{\ n_1, \ldots\ \}) \cup \ldots \\
&F[\![\ \texttt{def}_f\ f_1(n_1, \ldots) = \{\ e_1\ \};\ \ldots;\ e\ ]\!] = \\
&\quad (F[\![e]\!] \cup (F[\![e_1]\!] \setminus \{\ n_{1,1}, \ldots\ \}) \cup \ldots) \setminus \{\ f_1, \ldots\ \} \\
&F[\![\ c(n_1, \ldots)\ ]\!] = \{\ n_1, \ldots\ \} \\
&F[\![\ f(c, n_1, \ldots)\ ]\!] = \{\ f, n_1, \ldots\ \} \\
&F[\![\ \texttt{if}\ (p(n_1, \ldots))\ \texttt{ct else cf}\ ]\!] = \{\ n_1, \ldots\ \}
\end{aligned}
$$

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

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

Function Definition

$$
\begin{array}{l}
[\![\ \mathtt{def}_f\ \mathtt{f}_1(\mathtt{c}_1,\ \mathtt{n}_{1,1},\ \ldots) = \{\ \mathtt{e}_1\ \};\ \mathtt{def}_f\ \mathtt{f}_2 \ldots;\ \mathtt{e}\ ]\!] = \\
\quad \mathtt{def}_f\ \mathtt{w}_1(\mathtt{c}_1,\ \mathtt{env}_1,\ \mathtt{n}_{1,1},\ \ldots) = \{ \\
\quad\quad \mathtt{val}_p\ \mathtt{v}_1 = \mathtt{block\text{-}get}(\mathtt{env}_1,\ 1); \\
\quad\quad \ldots \\
\quad\quad [\![\mathtt{e}_1]\!][\mathtt{f}_1 \mapsto \mathtt{env}_1,\ \mathtt{FV}_1(0) \mapsto \mathtt{v}_1,\ \ldots\ ]\ \}; \\
\quad \mathtt{def}_f\ \mathtt{w}_2(\ldots; \\
\quad \mathtt{val}_p\ \mathtt{f}_1 = \mathtt{block\text{-}alloc\text{-}202}(|\mathtt{FV}_1| + 1); \\
\quad \mathtt{val}_p\ \mathtt{f}_2 = \ldots; \\
\quad \mathtt{val}_p\ \mathtt{t}_{1,1} = \mathtt{block\text{-}set}(\mathtt{f}_1,\ 0,\ \mathtt{w}_1); \\
\quad \mathtt{val}_p\ \mathtt{t}_{1,2} = \mathtt{block\text{-}set}(\mathtt{f}_1,\ 1,\ \mathtt{FV}_1(0)); \\
\quad \ldots; \\
\quad \mathtt{val}_p\ \mathtt{t}_{2,1} = \mathtt{block\text{-}set}(\mathtt{f}_2,\ 0,\ \mathtt{w}_2); \\
\quad \ldots; \\
\quad [\![\mathtt{e}]\!]
\end{array}
$$

where $\mathtt{FV}_i$ is an (arbitrary) ordering of the set $\mathtt{F}[\mathtt{e}_i] \setminus \{\ \mathtt{f}_i,\ \mathtt{n}_{i,1},\ \ldots\ \}$

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:

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

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 CPS/MiniScala version of makeAdder? (Only the closure conversion):

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

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)
```

Solution - Function Definition

```
def_f wMakeAdder(rc1, env1, x) = { // CPS with closures converted
  def_f wAnon1(rc2, env2, y) = {
    val_l fv1 = blok-get(env2, 1);
    val_l v = fv1 + y; rc2(v)
  };
  val_p anon1 = block-alloc-202(2);
  val_p t1 = block-set(anon1, 0, wAnon1);
  val_p t2 = block-set(anon1, 1, x);
  rc1(anon1)
};
val_p makeAdder = block-alloc-202(1);
val_p t3 = block-set(makeAdder, 0, wMakeAdder);
```

to continue. . .

Solution - Function Application

```
val_I i1 = 1;
def_C rc4(r1) = {
  val_P inc = id(r1);
  val_I i2 = 42;
  def_C rc3(r2) = { val_I 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.

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 put 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

$$
\begin{array}{l}
[\![\ \mathtt{def}_f\ \mathtt{f}_1(\mathtt{c}_1,\ \mathtt{n}_{1,1},\ \ldots)\ =\ \{\ \mathtt{e}_1\ \};\ \mathtt{def}_f\ \mathtt{f}_2\ldots;\ \mathtt{e}\ ]\!] = \\
\quad \mathtt{def}_f\ \mathtt{w}_1(\mathtt{c}_1,\ \mathtt{n}_{1,1},\ \ldots,\ \mathtt{u}_1,\ \ldots)\ \ =\ \{ \\
\quad\quad [\![\mathtt{e}_1]\!][\mathtt{FV}_1(0) \mapsto \mathtt{u}_1,\ \ldots\ ]\ \}; \\
\quad \mathtt{def}_f\ \mathtt{s}_1(\mathtt{sc1},\ \mathtt{env}_1,\ \mathtt{sn}_{1,1},\ \ldots)\ \ =\ \{ \\
\quad\quad \mathtt{val}_p\ \mathtt{v}_1\ =\ \mathtt{block\text{-}get}(\mathtt{env}_1,\ 1); \\
\quad\quad \ldots; \\
\quad\quad \mathtt{w}_1(\mathtt{sc}_1,\ \mathtt{sn}_{1,1},\ \ldots,\ \mathtt{v}_1,\ \ldots);\ \}; \\
\quad \mathtt{def}_f\ \mathtt{w}_2(\ldots; \\
\quad \mathtt{def}_f\ \mathtt{s}_2(\ldots; \\
\quad \mathtt{val}_p\ \mathtt{f}_1\ =\ \mathtt{block\text{-}alloc\text{-}202}(|\mathtt{FV}_1|\ +\ 1); \\
\quad \ldots; \\
\quad \mathtt{val}_p\ \mathtt{t}_{1,1}\ =\ \mathtt{block\text{-}set}(\mathtt{f}_1,\ 0,\ \mathtt{s}_1); \\
\quad \mathtt{val}_p\ \mathtt{t}_{1,2}\ =\ \mathtt{block\text{-}set}(\mathtt{f}_1,\ 1,\ \mathtt{FV}_1(0)); \\
\quad \ldots; \\
\quad [\![\mathtt{e}]\!]
\end{array}
$$

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

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

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

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 letf.

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)

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

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

CPS Hoisting (2)

$$
\begin{aligned}
&[\![\ \texttt{def}_c\ \texttt{c1}(\texttt{n}_{1,1}, \ldots) = \{\ \texttt{e1}\ \};\ \texttt{def}_c \ldots;\ \texttt{e}\ ]\!] \\
&\quad \texttt{def}_f\ \texttt{fs}_{1,1} \ldots;\ \texttt{def}_f\ \texttt{fs}_{1,m_1};\ \texttt{def}_f\ \texttt{fs}_{2,1} \ldots;\ \ldots;\ \texttt{def}_f\ \texttt{fs}_1 \ldots;\ \texttt{def}_f\ \texttt{fs}_n \ldots; \\
&\quad \texttt{def}_c\ \texttt{c1}(\texttt{n}_{1,1}, \ldots) = \{\ \texttt{e}_{h_1}\ \};\ \texttt{def}_c \ldots;\ \texttt{e}_h \\
&\text{where } [\![\texttt{e}_i]\!] = \texttt{def}_f\ \texttt{fs}_{i,1} \ldots;\ \ldots;\ \texttt{def}_f\ \texttt{fs}_{i,m_i} \ldots;\ \texttt{e}_{h_i} \\
&\quad \text{and } [\![\texttt{e}]\!] = \texttt{def}_f\ \texttt{fs}_1 \ldots;\ \ldots;\ \texttt{def}_f\ \texttt{fs}_n \ldots;\ \texttt{e}_h
\end{aligned}
$$

$$
\begin{aligned}
&[\![\ \texttt{def}_f\ \texttt{f}_1(\texttt{n}_{1,1}, \ldots) = \{\ \texttt{e1}\ \};\ \texttt{def}_f \ldots;\ \texttt{e}\ ]\!] \\
&\quad \texttt{def}_f\ \texttt{fs}_{1,1} \ldots;\ \texttt{def}_f\ \texttt{fs}_{1,m_1};\ \texttt{def}_f\ \texttt{fs}_{2,1} \ldots;\ \ldots;\ \texttt{def}_f\ \texttt{fs}_1 \ldots;\ \texttt{def}_f\ \texttt{fs}_n \ldots; \\
&\quad \texttt{def}_f\ \texttt{f}_1(\texttt{n}_{1,1}, \ldots) = \{\ \texttt{e}_{h_1}\ \};\ \texttt{def}_f \ldots;\ \texttt{e}_h \\
&\text{where } [\![\texttt{e}_i]\!] = \texttt{def}_f\ \texttt{fs}_{i,1} \ldots;\ \ldots;\ \texttt{def}_f\ \texttt{fs}_{i,m_i} \ldots;\ \texttt{e}_{h_i} \\
&\quad \text{and } [\![\texttt{e}]\!] = \texttt{def}_f\ \texttt{fs}_1 \ldots;\ \ldots;\ \texttt{def}_f\ \texttt{fs}_n \ldots;\ \texttt{e}_h
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{e}]\!] \text{ when e is any other kinf of expression} = \\
&\quad \emptyset;\ \texttt{e}
\end{aligned}
$$

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 - 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.

In languages like Scala or Java 8, a special syntax exists for anonymous functions, which are translated to nested classes.

Adder Maker In Scala

To see how closures are handled in Scala, let's look at how the translation of the Scala equivalent of the `makeAdder` function:

```
def makeAdder(x: Int): Int => Int =
  (y: Int) => x + y
val increment = makeAdder(1)
increment(41)
```

Translated Adder

```
class Anon extends Function1[Int,Int] {
  private val x: Int;
  def this(x: Int) = { this.x = x }
  def apply(y: Int): Int = this.x + y
}

def makeAdder(x: Int): Function1[Int,Int] = new Anon(x)
val increment = makeAdder(1)
increment.apply(41)
```

Translated Adder

(Hoisted) closure class: the code is in the `apply` method, the environment in the object itself: it's a flat closure.

```
class Anon extends Function1[Int,Int] {
  private val x: Int;
  def this(x: Int) = { this.x = x }
  def apply(y: Int): Int = this.x + y
}

def makeAdder(x: Int): Function1[Int,Int] =
  new Anon(x)
val increment = makeAdder(1)
increment.apply(41)
```

env. initialization

env. extraction

closure creation

closure application (the closure is passed implicitly as `this`)