CS107: Value Representation and Closure Conversion

Guannan Wei
guannan.wei@tufts.edu

October 20, 2026

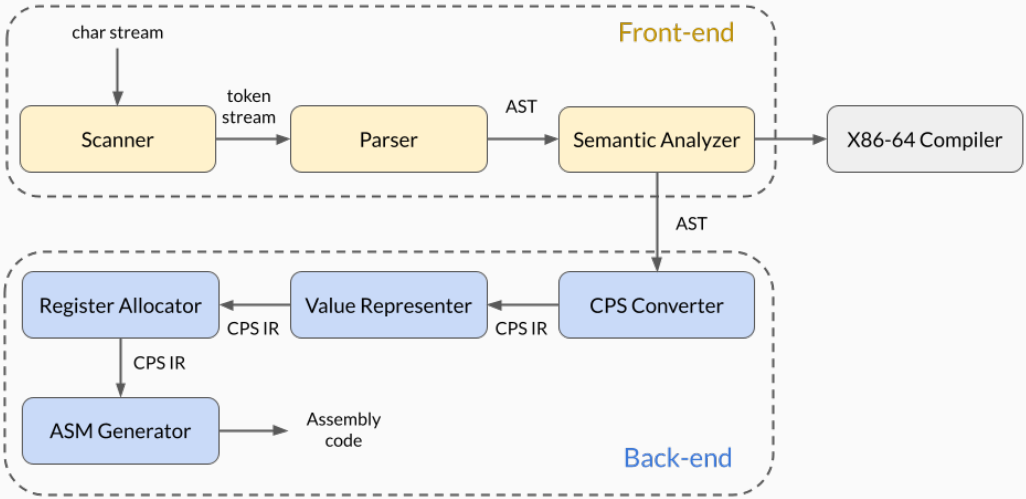
Fall 2026

Tufts University

Logistics

- Midterm exam: Oct 22, 12:00-1:15pm, in class
- Final exam: Dec 17, 3:30pm to 5:30pm, JCC 610

Values Representation Phase



Values Representation Phase

- **Input**: CPS program where all values and primitives are "high-level", in that they have the semantics of the MiniScala language gives them.
- **Output**: a "low-level" version of that program as output, in which all values are either integers or pointers and primitives correspond directly to instructions of a typical microprocessor.

As usual, we will specify this phase as a transformation function called $[\![.]\!]$, which maps high-level CPS terms to their low-level equivalent.

Tagging Scheme

For MiniScala, we require that the two least-significant bits (LSBs) of pointers are always 0. This enables us to represent integers, booleans and unit as tagged values.

The tagging scheme for MiniScala is given by the table below.

Kind of value	LSBs
Integer	$...1_2$
Block (pointer)	$...00_2$
Character	$...110_2$
Boolean	$...1010_2$
Unit	$...0010_2$

Tagging Integers

We have seen this encoding last time:

```
⟦ val_l n = i; e ⟧ =
  val_l n = 2 * i + 1; ⟦e⟧
  where i is an integer literal

⟦ if (int?(n)) ct else cf ⟧ =
  val_l c_1 = 1;
  val_p t_1 = n & c_1;
  if (t_1 == c_1) ct else cf

⟦ val_p n = n_1 + n_2; e ⟧ =
  val_l c_1 = 1;
  val_p t_1 = n_1 + n_2;
  val_p n = t_1 - c_1;
  ⟦e⟧
```

Representing MiniScala Integers - More

Integer comparison primitives:

$$[\![\ \texttt{if (}n_1\ \texttt{<}\ n_2\texttt{) ct else cf}\ ]\!] =$$
$$\texttt{if (}n_1\ \texttt{<}\ n_2\texttt{) ct else cf}$$

Other arithmetic primitives and integer primitives are similar.

Block Operations

$$
\begin{array}{l}
[\![\ \texttt{val}_p\ \texttt{n = block-alloc-k(n}_1\texttt{); e}\ ]\!] = \\
\quad \texttt{val}_l\ \underline{\texttt{c}_1}\ \texttt{= 1;} \\
\quad \texttt{val}_p\ \underline{\texttt{t}_1}\ \texttt{= n}_1\ \texttt{>> c}_1\texttt{;} \\
\quad \texttt{val}_p\ \texttt{n = block-alloc-k(t}_1\texttt{);} \\
\quad [\![\texttt{e}]\!]
\end{array}
$$

$$
\begin{array}{l}
[\![\ \texttt{val}_p\ \texttt{n = block-tag(n}_1\texttt{); e}\ ]\!] = \\
\quad \texttt{val}_l\ \underline{\texttt{c}_1}\ \texttt{= 1;} \\
\quad \texttt{val}_p\ \underline{\texttt{t}_1}\ \texttt{= block-tag(n}_1\texttt{);} \\
\quad \texttt{val}_p\ \underline{\texttt{t}_2}\ \texttt{= t}_1\ \texttt{<< c}_1\texttt{;} \\
\quad \texttt{val}_p\ \texttt{n = t}_2\ \texttt{+ c}_1\texttt{;} \\
\quad [\![\texttt{e}]\!]
\end{array}
$$

Note: allocation is parameterized over tags `k`, which can be retrieved by `block-tag`; see `project4/BlockTag.scala` and `project4/CPSPrimitives.scala`. Other block primitives are similar.

Character Operations

Read and write a char from console:

```
⟦ val_p n = byte-read(); e ⟧ =
   val_p t_1 = byte-read();
   val_l c_1 = 1;
   val_p t_2 = t_1 << c_1;
   val_p n = t_2 + c_1;
   ⟦e⟧

⟦ val_p n = byte-write(n_1); e ⟧ =
  left as an exercise
```

Character Operations

$[\![$ val_l n = c; e $]\!]$ =
  val_l n = (c << 3) | 110_2; $[\![$e$]\!]$
  where c is a character literal

$[\![$ val_p n = char→int(n_1); e $]\!]$ =
  val_l c_2 = 2;
  val_p n = n_1 >> c_2;
  $[\![$e$]\!]$

$[\![$ val_p n = int→char(n_1); e $]\!]$ =
  val_l c_2 = 2;
  val_p t_1 = n_1 << c_2;
  val_p n = t_1 + c_2;
  $[\![$e$]\!]$

Boolean Operations

```
⟦ val_l n = true; e ⟧ =
  val_l n = 11010_2; ⟦e⟧

⟦ val_l n = false; e ⟧ =
  val_l n = 01010_2; ⟦e⟧

⟦ if (bool?(n)) ct else cf ⟧ =
  val_l m = 1111_2;
  val_l t = 1010_2;
  val_p r = n & m;
  if (r == t) ct else cf
```

Representation MiniScala Unit, etc.

$[\![$ `val`$_l$ `n = (); e` $]\!]$ =
  `val`$_l$ `n =` 0010_2`; e`

$[\![$ `if (unit?(n)) ct else cf` $]\!]$ =
  left as an exercise

$[\![$ `halt(n)` $]\!]$ =
  left as an exercise

Names are left untouched by value representation transformation:

$[\![$`n`$]\!]$ =
  `n`

Representation MiniScala Unit, etc.

$[\![$ `val`$_l$ `n = (); e` $]\!]$ =
  `val`$_l$ `n =` 0010_2`; e`

$[\![$ `if (unit?(n)) ct else cf` $]\!]$ =
  left as an exercise

$[\![$ `halt(n)` $]\!]$ =
  left as an exercise

Names are left untouched by value representation transformation:

$[\![$`n`$]\!]$ =
  `n`

That's all for the primitive values! Then what about functions and continuations?

Representing MiniScala Continuations

Continuations are limited enough (they are not first-class) that they do not need to special representation in the language.

Their body must still be transformed recursively:

```
⟦ def_c c_1(n_1, ...) = { e_1 }; def_c ...; e ⟧ =
  def_c c_1(n_1, ...) = { ⟦e_1⟧ }; def_c ...; ⟦e⟧

⟦ nc(n_1, ...) ⟧ =
  nc(n_1, ...)
```

Representing MiniScala Functions

For functions, so far we only have top-level functions. If we keep it that way, we do not special representation for functions either:

```
⟦ def_f f_1(c_1, n_1,1, ...) = { e_1 }; def_f ...; e ⟧ =
  def_f f_1(c_1, n_1,1, ...) = { ⟦e_1⟧ }; def_f ...; ⟦e⟧

⟦ n(nc, n_1, ...) ⟧ =
  n(nc, n_1, ...)
```

Representing MiniScala Functions

For functions, so far we only have top-level functions. If we keep it that way, we do not special representation for functions either:

```
⟦ def_f f_1(c_1, n_1,1, ...) = { e_1 }; def_f ...; e ⟧ =
  def_f f_1(c_1, n_1,1, ...) = { ⟦e_1⟧ }; def_f ...; ⟦e⟧

⟦ n(nc, n_1, ...) ⟧ =
  n(nc, n_1, ...)
```

However, if we are about to support first-class functions (defining nested functions, saving them to variables, returning them from other functions, etc.), we need a representation for functions known as **closures**. The phase taking care of representing functions is commonly known as **closure conversion**.

Higher-Order Functions

Higher-order Functions

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 (pointer) as an argument, and to return a function (pointer) 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 (e.g. 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, 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
```

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
```

- We cannot simply use code pointers to represent `increment` and `decrement`, as they are different functions.
- Observe that they do share the same code (the body of the function returned by `makeAdder`), but accessing a different value of `x`.

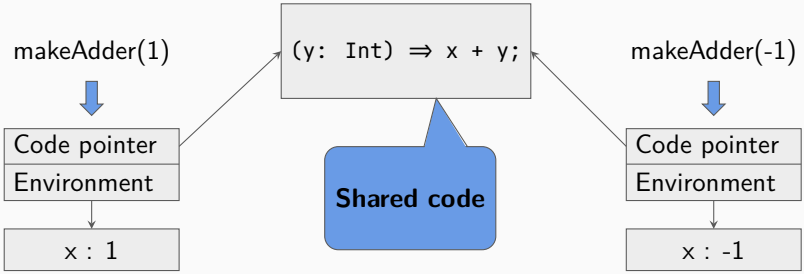
Closures

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

Closure = code pointer + environment

- The environment gives the values of the free variable(s) - here `x` for the returned function `(y: Int) => x + y`.
- The pair of code pointers and environment is **closed**, i.e. self-contained, hence the name "closure".

Closures



The code of the closure must be evaluated under its environment, so that x is "known".

Simulating Closures with Objects and Classes

How Scala (and object-oriented languages) handle closure? Let's look at how the translation of the Scala equivalent of the `makeAdder` function:

```
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)
```

Guy Steele: A closure is an object that supports exactly one method: "apply".

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)
```

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; create closure

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; create closure

env. extraction

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; create closure

env. extraction

closure creation

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; create closure

env. extraction

closure creation

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

Closures and objects

Closures are a poor man's object, and objects are a poor man's closures.

RE: What's so cool about Scheme?
https://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html

Representing 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

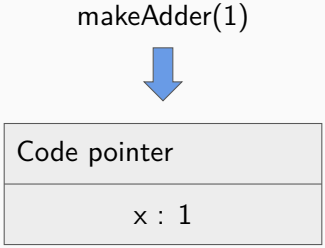
During function application, nothing is known about the closure being called - it can be any closure in the program.

```
def hof(f: Int => Int) = f(42)
```

- The code pointer must be at a known and constant location, so that it can be extracted and jumped to.
- 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.



- No sharing between multiple closures: each closure has its own copy 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 value representation phase, called closure conversion.

- It encodes the high-level notion of functions of the source language using the low-level concepts of the target language: functions with free variables translated into equivalent **closed functions**.
- The output of closure conversion is a program in which functions can be represented as pointers to code and first-order data.
- At runtime, closures are heap-allocated blocks and code pointers.

Closure Conversion - 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((env$_1$: Env, x: Int) =>
    closureMake((env$_2$: Env, y: Int) => closureGet(env$_2$, 1) + y,
      List(x)), // Value to be added into the environment
    Nil
  );
closureGet(makeAdder, 0)(makeAdder, 42)
```

Summary

- We have discussed value representation for integers
- Introduce closures to represent higher-order functions
- This Thursday: Midterm
- Next week: rules for closure conversion