CS107: Value and Data Representation

Guannan Wei
guannan.wei@tufts.edu

February 17, 2026

Spring 2026

Tufts University

Project 4

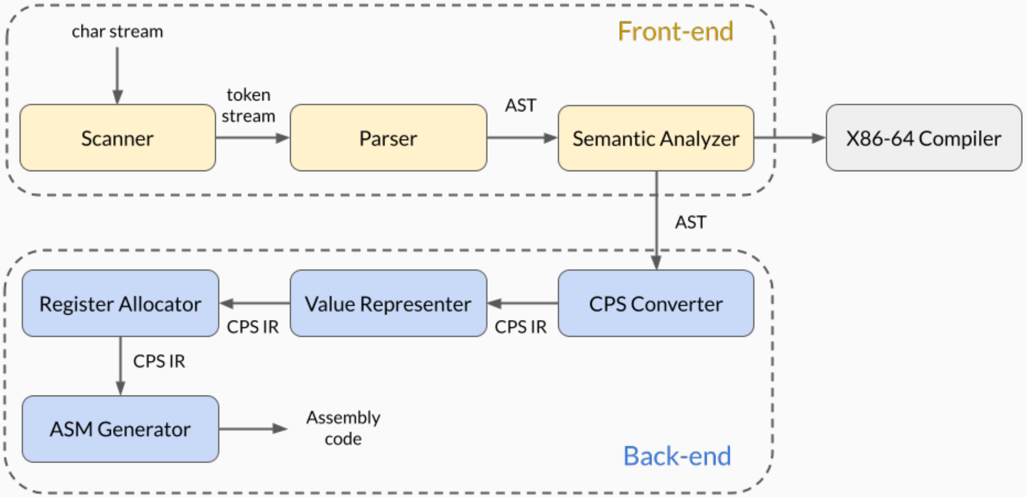
- Project 4 has been released. Due March 1, 11:59pm.
- The skeleton compiler handles a more realistic language.

```
->  proj4 git:(main) cloc .
      69 text files.
      58 unique files.
      11 files ignored.

-----------------------------------------------------
Language          files     blank     comment     code
-----------------------------------------------------
Scala                54       641         620     3572
-----------------------------------------------------
```

- Your task: implementing the CPS conversion we have been discussing in class.

Compiler Architecture



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	$\dots 1_2$
Block (pointer)	$\dots 00_2$
Character	$\dots 110_2$
Boolean	$\dots 1010_2$
Unit	$\dots 0010_2$

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
```

Tagging Integers

Deriving the encoded version of arithmetic primitives for tagged integers:

```
⟦n + m⟧ = 2[(⟦n⟧ - 1) / 2 + (⟦m⟧ - 1) / 2] + 1
        = (⟦n⟧ - 1) + (⟦m⟧ - 1) + 1
        = ⟦n⟧ + ⟦m⟧ - 1

⟦n - m⟧ = 2[(⟦n⟧ - 1) / 2 - (⟦m⟧ - 1) / 2] + 1
        = (⟦n⟧ - 1) - (⟦m⟧ - 1) + 1
        = ⟦n⟧ - ⟦m⟧ + 1

⟦n * m⟧ = 2[((⟦n⟧ - 1) / 2) * ((⟦m⟧ - 1) / 2)] + 1
        = (⟦n⟧ - 1) * ((⟦m⟧ - 1) / 2) + 1
        = (⟦n⟧ - 1) * (⟦m⟧ >> 1) + 1
```

Representing MiniScala Functions

MiniScala functions also need to be represented specially so that the operations permitted on them - e.g. save them to variables - can be implemented in the target language.

The phase that takes care of representing functions is commonly known as **closure conversion**. While it logically belongs to the values conversion phase, it will be presented separately later.

For now, we assume that functions are not translated specially:

$[\![$ $\texttt{def}_f$ $\texttt{f}_1\texttt{(c}_1\texttt{, n}_{1,1}\texttt{, ...) = \{ e}_1\texttt{ \}; def}_f\texttt{ ...; e}$ $]\!]$ =
  $\texttt{def}_f$ $\texttt{f}_1\texttt{(c}_1\texttt{, n}_{1,1}\texttt{, ...) = \{}$ $[\![\texttt{e}_1]\!]$ $\texttt{\}; def}_f\texttt{ ...;}$ $[\![\texttt{e}]\!]$

$[\![$ $\texttt{n(nc, n}_1\texttt{, ...)}$ $]\!]$ =
  $\texttt{n(nc, n}_1\texttt{, ...)}$

Representing MiniScala Continuations

Continuations, unlike functions, are limited enough that they do not need to be "closure converted" - i.e. they don't need special representation in the target 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 Integers (1)

$[\![\ \texttt{val}_l\ \texttt{n = i; e}\ ]\!]\ \texttt{=}$

$\quad \texttt{val}_l\ \texttt{n = 2 * i + 1;}\ [\![\texttt{e}]\!]$

$\quad$ *where i is an integer literal*

$[\![\ \texttt{if (int?(n)) ct else cf}\ ]\!]\ \texttt{=}$

$\quad \texttt{val}_l\ \underline{\texttt{c}_1}\ \texttt{= 1;}$

$\quad \texttt{val}_p\ \underline{\texttt{t}_1}\ \texttt{= n \& c}_1\texttt{;}$

$\quad \texttt{if (t}_1\ ==\ \texttt{c}_1\texttt{) ct else cf}$

Representing MiniScala Integers (2)

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

… other arithmetic primitives are similar.

$$
\begin{array}{l}
[\![\ \texttt{if}\ (\texttt{n}_1 < \texttt{n}_2)\ \texttt{ct}\ \texttt{else}\ \texttt{cf}\ ]\!] = \\
\quad \texttt{if}\ (\texttt{n}_1 < \texttt{n}_2)\ \texttt{ct}\ \texttt{else}\ \texttt{cf}
\end{array}
$$

… other integer comparison primitives are similar.

Representing MiniScala Integers (3)

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

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

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.

Representing MiniScala Integers (4)

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

$$
\begin{array}{l}
[\![\ \texttt{val}_p\ \texttt{n = byte-write(}\texttt{n}_1\texttt{); e}\ ]\!]\ \texttt{=} \\
\quad \texttt{left as an exercise}
\end{array}
$$

Representing MiniScala Characters

$[\![$ $\texttt{val}_l$ n = c; e $]\!]$ =
 $\texttt{val}_l$ n = (c << 3) | 110_2; $[\![$e$]\!]$
 where c is a character literal

$[\![$ $\texttt{val}_p$ n = char$\rightarrow$int($\texttt{n}_1$); e $]\!]$ =
 $\texttt{val}_l$ $\underline{\texttt{c}_2}$ = 2;
 $\texttt{val}_p$ n = $\texttt{n}_1$ >> $\texttt{c}_2$;
 $[\![$e$]\!]$

$[\![$ $\texttt{val}_p$ n = int$\rightarrow$char($\texttt{n}_1$); e $]\!]$ =
 $\texttt{val}_l$ $\underline{\texttt{c}_2}$ = 2;
 $\texttt{val}_p$ $\underline{\texttt{t}_1}$ = $\texttt{n}_1$ << $\texttt{c}_2$;
 $\texttt{val}_p$ n = $\texttt{t}_1$ + $\texttt{c}_2$;
 $[\![$e$]\!]$

Representing MiniScala Boolean

$$
\begin{array}{l}
[\![\ \texttt{val}_l\ \texttt{n = true; e}\ ]\!]\ \texttt{=} \\
\quad \texttt{val}_l\ \texttt{n = }\texttt{11010}_2\texttt{;}\ [\![\texttt{e}]\!] \\
\\
[\![\ \texttt{val}_l\ \texttt{n = false; e}\ ]\!]\ \texttt{=} \\
\quad \texttt{val}_l\ \texttt{n = }\texttt{01010}_2\texttt{;}\ [\![\texttt{e}]\!] \\
\\
[\![\ \texttt{if (bool?(n)) ct else cf}\ ]\!]\ \texttt{=} \\
\quad \texttt{val}_l\ \underline{\texttt{m}}\ \texttt{= }\texttt{1111}_2\texttt{;} \\
\quad \texttt{val}_l\ \underline{\texttt{t}}\ \texttt{= }\texttt{1010}_2\texttt{;} \\
\quad \texttt{val}_p\ \underline{\texttt{r}}\ \texttt{= n \& m;} \\
\quad \texttt{if (r == t) ct else cf}
\end{array}
$$

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
```

Exercise

Translating the following program:

```
def_f succ(c, x) = {
  val_l c_1 = 1;
  val_p t_1 = x + c_1;
  c(t_1)
};
succ
```

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 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 (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, 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:

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

Translated Adder

Translate to object-oriented code:

```
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".

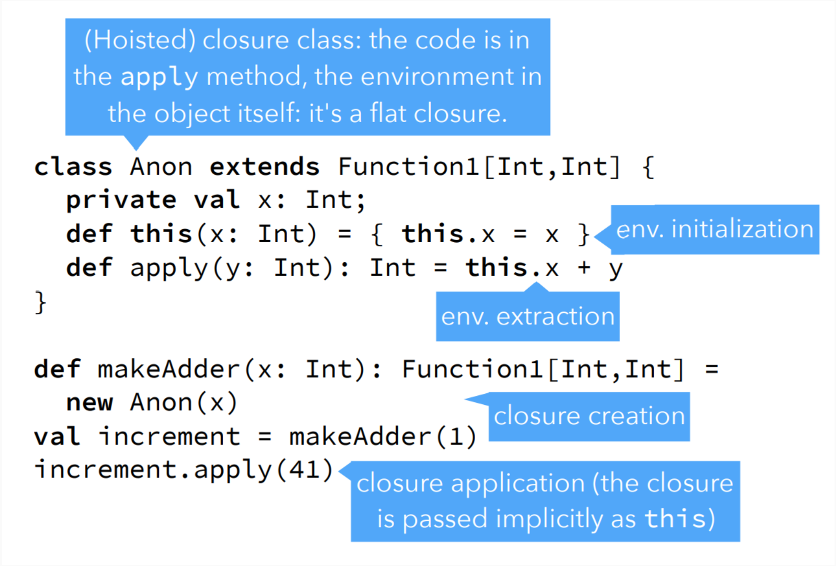
Closures and objects

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

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
}
```

env. initialization

env. extraction

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

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.

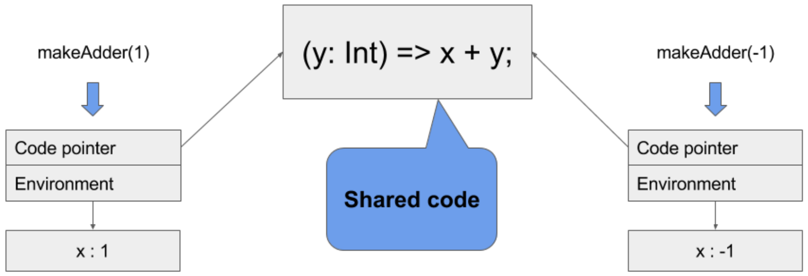
Closures

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

A pair of code pointers and an environments giving the values of the free variable(s) - here `x` - is called a closure.

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 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.

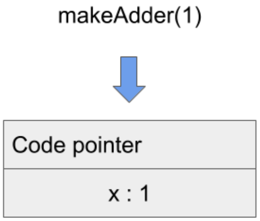
```
def hof(f: Int => Int) = f(42)
```

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 translation phase, called closure conversion.

- Closure conversion transforms functions that have free variables into equivalent closed functions.
- The output of closure conversion is therefore a program in which functions can be represented as pointers to code and first-order data.

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 low-level concepts of the target language
- 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

- Value representation for integers
- Introduce closures to represent higher-order functions

Next time: full rules for closure conversion!