CS107: Value and Data Representation

Guannan Wei
guannan.wei@tufts.edu

October 13, 2026

Fall 2026

Tufts University

Recap & Logistics

- This Thursday (Oct 15): Review for Midterm lead by Jonah
- Next Thursday (Oct 22): Midterm Exam

Improved CPS Translation

- Last (recorded) lecture:
  - Improving the CPS translation
  - CFG/RTF/SSA IRs

Improved CPS Translation

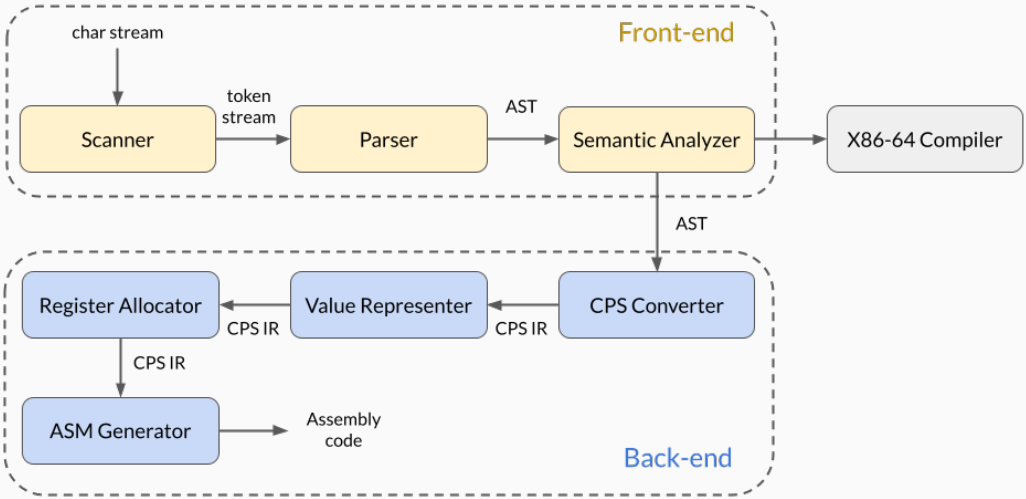
- Last (recorded) lecture:
  - Improving the CPS translation
  - CFG/RTF/SSA IRs

Key ideas of the improved CPS translation:

- Identify translation under certain contexts (e.g. tail position, condition of `if`/`while`) to avoid unnecessary continuations.
- Specify three mutually recursive translation function for the non-tail case, the tail case, and the condition case.

Project 4 (CPS translation): due Oct 25 11:59pm

Compiler Architecture



The Problem

A compiler must have a way of representing the values of the source language using values of the target language.

This representation must be as **efficient** as possible, in terms of both memory consumption and execution speed.

For simple languages like C, the representation is trivial as most source values (e.g. C's int, float, double, etc.) map directly to the corresponding values of the target machine.

For more complex languages, things are not so easy...

Example: Polymorphic Functions

Consider the following MiniScala function:

```
def makePair[T,U](t: T, u: U) = (t, u);
makePair[Int, Char](3, 'a')
```

Obviously, nothing is known about the type of `t` and `u` at compilation time.

Example: Polymorphic Functions

Consider the following MiniScala function:

```
def makePair[T,U](t: T, u: U) = (t, u);
makePair[Int, Char](3, 'a')
```

Obviously, nothing is known about the type of `t` and `u` at compilation time.

How can the compiler generate code to pass `t` and `u` around as arguments, copy them in their memory location, etc. given that their size is unknown?

For "dynamically-typed" languages it is easy, but for statically-typed languages that offer parametric polymorphism - e.g. Scala, Java 5+ and Haskell, etc, generating code to manipulate such values is problematic.

Example: Higher-Order Functions

First-class functions:

```
def adder(x: Int): Int => Int = {
  def inner(y: Int) = x + y
  inner
};
val inc = adder(1);
println(inc(2));
println(inc(3))
```

Function `adder` returns another function that depends on the argument of `adder`. How should we generate code representing such returned functions depending on the argument of `adder`?

Solutions

We will first examine value representations for polymorphic code:

- Boxed representation
- Tagging
  - NaN tagging
- On-demand boxing
- Specialization

Then next time we will talk about higher-order functions.

The Solutions - Boxed Representation

The simplest solution to the values representation problem is to use a **(fully) boxed values representation**.

The idea is that all source values are represented by **a pointer to a tagged block** (a.k.a. a box), allocated in the heap. The block contains the representation of the source value, and the tag identifies the type of that value.

The Solutions - Boxed Representation

The simplest solution to the values representation problem is to use a **(fully) boxed values representation**.

The idea is that all source values are represented by **a pointer to a tagged block** (a.k.a. a box), allocated in the heap. The block contains the representation of the source value, and the tag identifies the type of that value.

Simple but costly: even small values like integers or booleans are stored in the heap. A simple operation like addition requires fetching the integers to add from their box, adding them and storing the result in a newly-allocated box...

The Solutions - Tagging

To avoid paying the hefty price of fully boxed representation for small values like integers, booleans, etc., **tagging** (bit stealing) can be used for them.

Tagging takes advantage of the fact that, on most target architectures, heap pointers are aligned on multiple of 2, 4 or more addressable units. Therefore, some of their least significant bits (LSBs) are always zero.

150 in binary:

```
0b10010110
         ^
         |
```

As a consequence, it is possible to use values with non-zero LSBs to represent small values.

Integer Tagging Example

A popular technique to represent a source integer `n` as a tagged target value is to encode it as `2n + 1`. This ensures that the LSB of the encoded integer is always 1, which makes it distinguishable from pointers at run time.

$[\![\texttt{0b110}]\!] = \texttt{0b1101}$

Integer Tagging Example

A popular technique to represent a source integer `n` as a tagged target value is to encode it as `2n + 1`. This ensures that the LSB of the encoded integer is always 1, which makes it distinguishable from pointers at run time.

$[\![\texttt{0b110}]\!] = \texttt{0b1101}$

Drawback:

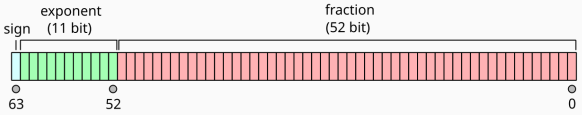
- Source integers have one bit less than the integers of the target architecture.
- Some applications become very cumbersome to write in such conditions - e.g. the computation of a 32 bits checksum.

For that reason, some languages (e.g. Java) offer several kinds of integer-like types: fast, tagged integers with reduced range, and slower, boxed integers with the full range of the target architecture.

NaN Tagging

On modern 64-bits architectures, we could use 64-bits words to represent values.

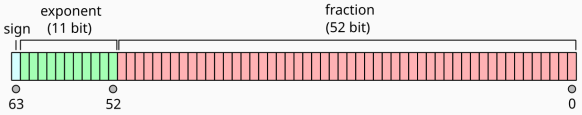
Although standard tagging techniques can be used in that case, it is also possible to take advantage of a characteristic of 64 bits IEEE 754 floating-point values (i.e. double in Java): so-called not-a-number values (NaN), which are used to indicate errors, are identified only by the 12 most-significant bits, which must be 1.



NaN Tagging

On modern 64-bits architectures, we could use 64-bits words to represent values.

Although standard tagging techniques can be used in that case, it is also possible to take advantage of a characteristic of 64 bits IEEE 754 floating-point values (i.e. double in Java): so-called not-a-number values (NaN), which are used to indicate errors, are identified only by the 12 most-significant bits, which must be 1.



The 52 least-significant bits can be arbitrary. Therefore, floating-point values can be represented naturally, and other values (pointers, integers, booleans, etc.) as specific NaN values.

Solutions - On-Demand Boxing

Hybrid solution:

Statically-typed languages have the option of using **unboxed/untagged values for monomorphic code**, and switching to (and from) **boxed/tagged representation only for polymorphic code**.

Dynamically-typed language implementations can try to infer types to obtain the same result.

Solutions - (Full) Specialization

For statically-typed, polymorphic languages, specialization (or monomorphization) is another way to represent values.

Specialization consists in translating polymorphism away by producing several specialized versions of polymorphic code.

Example:

```
def makePair[T,U](t: T, u: U) = (t, u);
makePair[Int, Char](3, 'a')
// specialized to:
def makePair_Int_Char(t: Int, u: Char) = (t, u);
makePair_Int_Char(3, 'a')
```

(Full) Specialization Issues

Full specialization removes the need for a uniform representation of values in polymorphic code. However, it may introduce a considerable cost in terms of **code size**.

(Full) Specialization Issues

Full specialization removes the need for a uniform representation of values in polymorphic code. However, it may introduce a considerable cost in terms of **code size**.

Moreover, the specialization process can loop forever in pathological cases like:

```
class C[T]
class D[T] extends C[D[D[T]]]
```

- `D[Int]` extends `C[D[D[Int]]]`, so the compiler encounters `D[D[Int]]`
- `D[D[Int]]` extends `C[D[D[D[Int]]]]`, introducing `D[D[D[Int]]]`.
- ...

Solution - Partial Specialization

To reap some of the benefits of specialization without paying its full cost, it is possible to perform partial specialization.

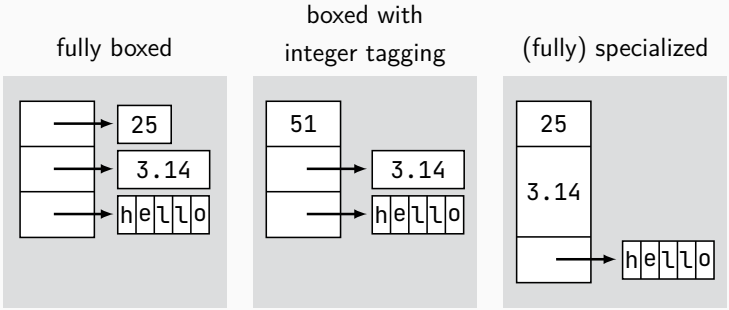
It is also possible to generate specialized code only for types that are deemed important - e.g. double in a numerical application - and fall back to non-specialized code (with a uniform representation) for the other types.

For example, Scala offers the @specialized annotation.

```
class SpecializedTo[@specialized(Int, Boolean) T] { ... }
```

Comparing Solutions

The figures below show how an object containing the integer 25, the real 3.14 and the string `"hello"` could be represented using the three techniques previously described.



Translation of operations

Translation of operations

When a source value is encoded to a target value, the operations on the source values have to be compiled according to the encoding.

Translation of operations

When a source value is encoded to a target value, the operations on the source values have to be compiled according to the encoding.

For example, if integers are boxed, addition of two source integers has to be compiled as the following sequence:

- the two boxed integers are unboxed,
- the sum of these unboxed values is computed, and
- finally a new box is allocated and filled with that result. This new box is the result of the addition.

Translation of operations

When a source value is encoded to a target value, the operations on the source values have to be compiled according to the encoding.

For example, if integers are boxed, addition of two source integers has to be compiled as the following sequence:

- the two boxed integers are unboxed,
- the sum of these unboxed values is computed, and
- finally a new box is allocated and filled with that result. This new box is the result of the addition.

Similarly, if integers are tagged, the tags must be removed before the addition is performed, and added back afterwards - at least in principle. In the case of tagging, it is however possible to do better for several operations...

Tagged Integer Arithmetic

Assuming integers are represented with the LSB is 1.

Deriving the encoded version of arithmetic primitives for tagged integers:

$$
\begin{aligned}
[\![n + m]\!] &= 2[([\![n]\!] - 1) / 2 + ([\![m]\!] - 1) / 2] + 1 \\
&= ([\![n]\!] - 1) + ([\![m]\!] - 1) + 1 \\
&= [\![n]\!] + [\![m]\!] - 1
\end{aligned}
$$

Tagged Integer Arithmetic

Assuming integers are represented with the LSB is 1.

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
```

What about -?

```
⟦n - m⟧ = ???
```

Tagged Integer Arithmetic

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
```

Similar derivations can be made for other operations (division, remainder, bitwise operations, etc.).

MiniScala Values Representation

MiniScala Values Representation

MiniScala has the following kinds of values: tagged blocks, functions, integers, characters, booleans, unit.

- Tagged blocks are represented as pointers to themselves.
- Integers, characters, booleans and the unit value are represented as tagged values.
- Functions are currently represented as code pointers, although we will see later that this is incorrect!

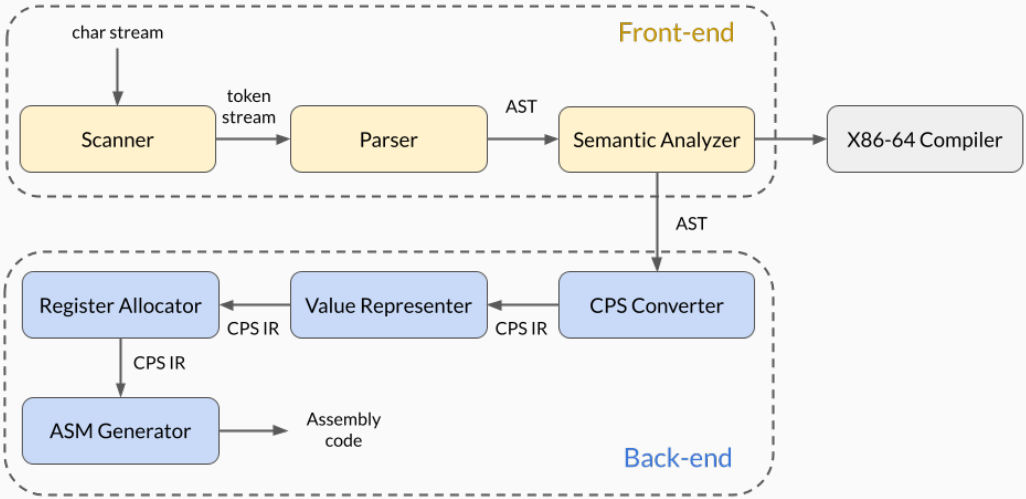
MiniScala Tagging Scheme

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	$...0010_2$

Compiler Architecture



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.

Representing MiniScala Integers

Appetizer:

```
⟦ 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

Appetizer:

$[\![$ $\mathtt{val}_l$ n = i; e $]\!]$ =
  $\mathtt{val}_l$ n = 2 * i + 1; $[\![$e$]\!]$
  where i is an integer literal

$[\![$ if (int?(n)) ct else cf $]\!]$ =
  $\mathtt{val}_l$ $\underline{\mathtt{c}_1}$ = 1;
  $\mathtt{val}_p$ $\underline{\mathtt{t}_1}$ = n & $\mathtt{c}_1$;
  if ($\mathtt{t}_1$ == $\mathtt{c}_1$) ct else cf

$[\![$ $\mathtt{val}_p$ n = $\mathtt{n}_1$ + $\mathtt{n}_2$; e $]\!]$ =
  $\mathtt{val}_l$ $\underline{\mathtt{c}_1}$ = 1;
  $\mathtt{val}_p$ $\underline{\mathtt{t}_1}$ = $\mathtt{n}_1$ + $\mathtt{n}_2$;
  $\mathtt{val}_p$ n = $\mathtt{t}_1$ - $\mathtt{c}_1$;
  $[\![$e$]\!]$

Next time: Full rules of value representation!