CS107: SSA IR and Value Representation

Guannan Wei
guannan.wei@tufts.edu

February 12, 2026

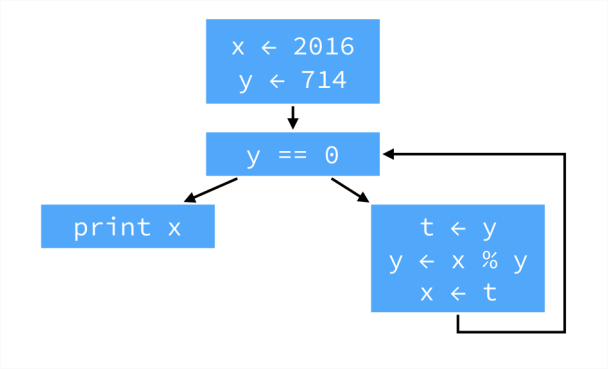
Spring 2026

Tufts University

Recap

- What did we learn last time?

RTL/CFG with Basic Blocks



- Register transfer language
- Control-flow graph
- Basic block: a list of sequentially executed instructions

RTL/CFG Issues

One problem of RTL/CFG is that even very simple optimizations (e.g. constant propagation, common-subexpression elimination) require *data-flow analyses*. This is because a single variable can be assigned multiple times.

RTL/CFG Issues

One problem of RTL/CFG is that even very simple optimizations (e.g. constant propagation, common-subexpression elimination) require *data-flow analyses*. This is because a single variable can be assigned multiple times.

Is it possible to improve RTL/CFG so that these optimizations can be performed without prior analysis?

Yes, by using a single-assignment variant of RTL/CFG!

IR #3: RTL/CFG In SSA Form

Static single-assignment (SSA): each variable has only one definition in the RTL/CFG representation.

- That single definition can be executed many times when the program is run (e.g., inside a loop), hence the qualifier static.
- SSA form is popular (e.g. LLVM) because it simplifies several optimizations and analyses.

Most (imperative) programs are not naturally in SSA form, and must therefore be transformed so that they are.

Transforming to SSA: Straight-line Code

Transforming a piece of straight-line code (i.e. without branches) to SSA is trivial: each definition of a given name gives rise to a new version of that name, identified by a subscript:

```
// non-SSA form
x <- 12
y <- 15
x <- x + y
y <- x + 4
z <- x + y
y <- y + 1
```

```
// corresponding SSA form
x_1 <- 12
y_1 <- 15
x_2 <- x_1 + y_1
y_2 <- x_2 + 4
z_1 <- x_2 + y_2
y_3 <- y_2 + 1
```

Φ-function

Join-points in the CFG (i.e., nodes with more than one predecessors) requires additional treatment, as each predecessor can bring its own version of a given name.

```
entry:
    x <- f()
    br cond left right
left:
    x_2 <- x_1 - 1
    jmp join
right:
    x_3 <- x_1 + 1
    jmp join
exit:
    print x // which x should we use?
```

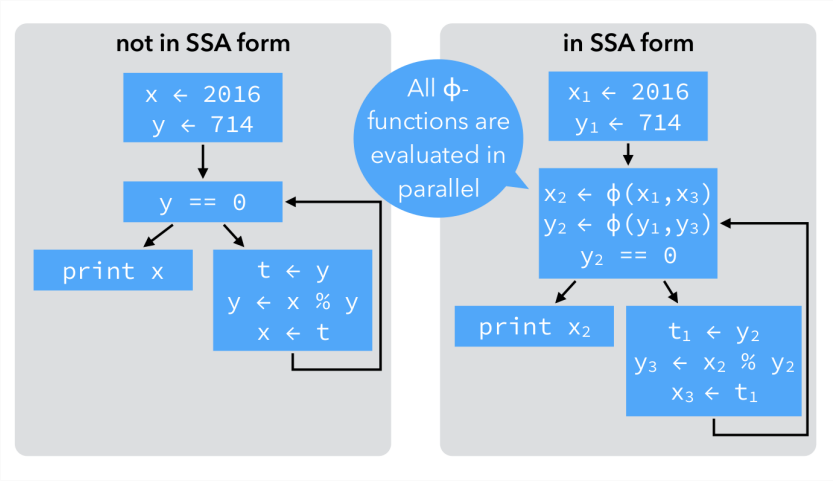
Φ-function

Join-points in the CFG (i.e., nodes with more than one predecessors) requires additional treatment, as each predecessor can bring its own version of a given name.

Solution: introduce a fictional Φ-function at the join point.

- A Φ-function takes as argument all the versions of the variable to reconcile, and
- automatically selects the right one depending on the incoming control flow.
- Φ-functions must be placed before any other non-Φ instructions.

Φ-functions Example



Evaluation of Φ-functions

It is crucial to understand that all Φ-functions of a block are evaluated in parallel, and not in sequence as the representation might suggest!

```
entry:
  i <- 5
l0:
  x_0 <- 0
  y_0 <- 0
  jmp l1
l1:
  x_1 <- Φ(l0 x_0, l1 y_1)
  y_1 <- Φ(l0 y_0, l1 x_1)
  print x1 y1
  cond <- (i > 0)
  i <- i - 1
  br cond l1 end
end:
```

Evaluation of Φ-functions

To make this clear, some authors write Φ-functions in matrix form, with one row per predecessor:

$$(x_1, y_2) \leftarrow \Phi \begin{pmatrix} x_1 & y_1 \\ x_3 & y_3 \end{pmatrix} \texttt{ instead of } x_1 \leftarrow \Phi(x_1, x_3); y_2 \leftarrow \Phi(y_1, y_3)$$

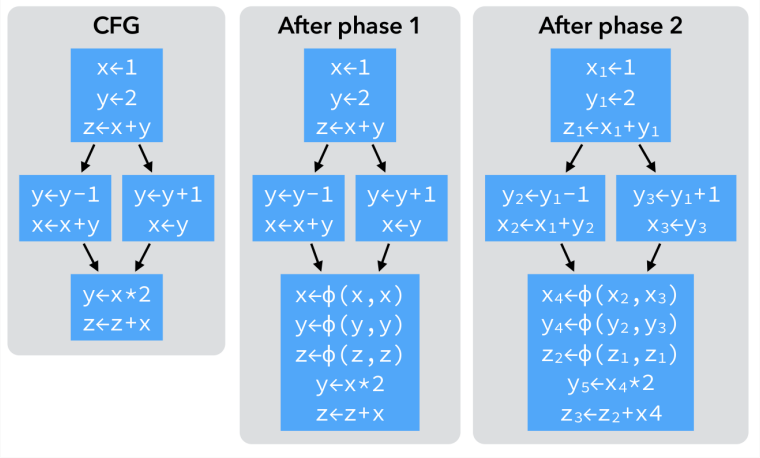
In our slides, we will usually stick to the common, linear representation, but keep the parallel nature of Φ-functions in mind.

(Naive) Building of SSA Form

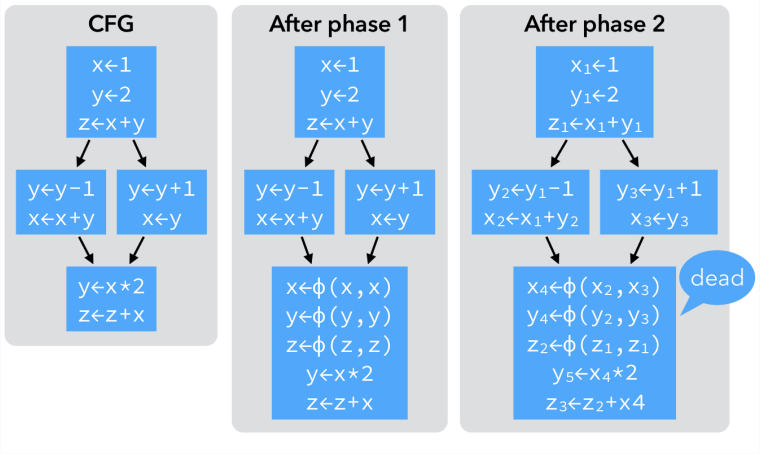
Naive technique to build SSA form:

- for each variable `x` of the CFG, at each join point `n`, insert a Φ-function of the form `x = Φ(x, ..., x)` with as many parameters as `n` has predecessors,
- compute reaching definitions, and use that information to rename any use of a variable according to the (now unique) definition reaching it.

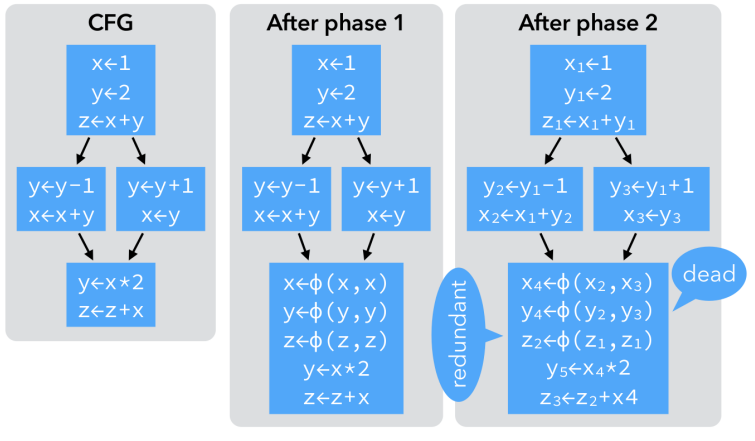
(Naive) Building of SSA Form



(Naive) Building of SSA Form



(Naive) Building of SSA Form



Better Building Techniques

The naive technique builds a program in SSA form and is equivalent to the original one.

However, it introduces too many Φ-functions: some are dead, and some are redundant. It in fact builds the **maximal** SSA form.

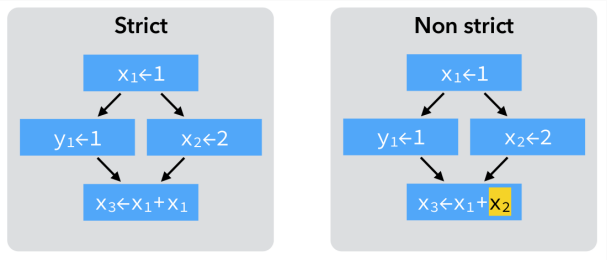
Better techniques exist to translate a program to SSA form.

Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. Cytron et al. ACM Transactions on Programming Languages and Systems, 1991

Strict SSA Form

Strict SSA form: an SSA form that all uses of a variable are dominated by the definition of that variable. In a CFG, a node n_1 dominates a node n_2 if all paths from the entry node to n_2 go through n_1 .

Strict SSA form guarantees that no variable is used before being defined.



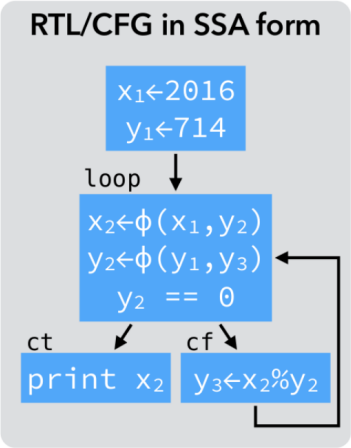
Comparing IRs: MiniScala/CPS vs RTL/CFG in SSA

Correspondence between SSA and CPS:

RTL/CFG in SSA	MiniScala/CPS
Named basic block	continuation
Φ-function	continuation argument
jump	continuation invocation
strict form	scoping rules

CPS is more powerful/flexible in that continuations can be argument, supporting more dynamic control flow structure.

MiniScala/CPS vs RTL/CFG in SSA




```
def_c loop(x2, y2) = {
  def_c ct() = { print(x2) };
  def_c cf() = {
    val_p y3 = x2 % y2;
    loop(y2, y3)
  };
  val_l z = 0;
  if (z == y2) ct else cf
};
val_l x1 = 2016; val_l y1 = 714;
loop(x1, y1)
```

Summary And References

Continuation-based, functional IRs like MiniScala/CPS are SSA done right, or even should replace it - or, at the very least, Φ-functions should be replaced by continuations arguments.

(This is fortunately starting to happen, e.g. the Swift Intermediate Language has basic-blocks with arguments.)

Reference: MiniScala/CPS is heavily based on the intermediate representation presented by Andrew Kennedy in *Compiling with Continuations, Continued*, in Proceedings of the International Conference on Functional Programming (ICFP) 2007.

Value Representation

Extended MiniScala Language

- Richer data types: polymorphic pairs, etc.

```
def makePair[T, U](t: T, u: U) = (t, u);

makePair[Int, Char](3, 'a')
```

Extended MiniScala Language

- Richer data types: polymorphic pairs, etc.

```
def makePair[T, U](t: T, u: U) = (t, u);

makePair[Int, Char](3, 'a')
```

- 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))
```

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...

The Problem

Most high-level languages have the ability to manipulate values whose exact type is unknown at compilation time.

This is trivially true of all "dynamically-typed" languages, but also of statically-typed languages that offer parametric polymorphism - e.g. Scala, Java 5+ and Haskell, etc.

Generating code to manipulate such values is problematic: how should values whose size is unknown be stored in variables, passed as arguments, etc. ?

Example

Consider the following MiniScala function:

```
def makePair[T,U](t: T, u: U) = (t, u);
```

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, copy them in their memory location, etc. given that their size is unknown?

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 can be used for them.

Tagging takes advantage of the fact that, on most target architectures, heap blocks 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.

```
0b110 -> 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.

```
0b110 -> 0b1101
```

Drawback: source integers have one bit less than the integers of the target architecture.

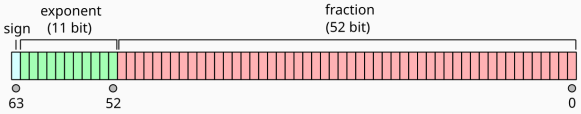
Some applications become very cumbersome to write in such conditions - e.g. the computation of a 32 bits checksum.

For that reason, some languages 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, it makes sense to use 64 bits words to represent values.

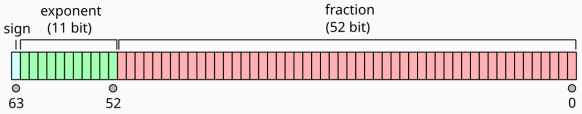
Although standard tagging techniques can be used in that case, it is also possible to take advantage of a characteristic of 64 bits IEEE 754 floating-point values (i.e. double in Java): so-called not-a-number values (NaN), which are used to indicate errors, are identified only by the 12 most-significant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



NaN Tagging

On modern 64-bits architectures, it makes sense to 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.

For example, when the type List[Int] appears in a program, the compiler produces a special class that represents lists of integers - and of nothing else.

(Full) Specialization Issues

Full specialization removes the need for a uniform representation of values in polymorphic code. However, this is achieved at 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, this is achieved at a considerable cost in terms of code size.

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

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

Solution - Partial Specialization

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

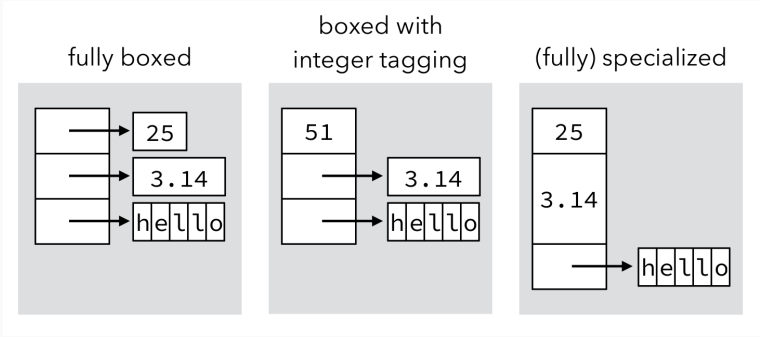
It is also possible to generate specialized code only for types that are deemed important - e.g. double in a numerical application - and fall back to non-specialized code (with a uniform representation) for the other types.

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

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

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
```

Tagged Integer Arithmetic

Deriving the encoded version of arithmetic primitives for tagged integers:

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

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

Tagged blocks are represented as pointers to themselves. Functions are currently represented as code pointers, although we will see later that this is incorrect!

Integers, characters, booleans and the unit value are represented as tagged values.

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	$...0010_2$

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
```

Representing MiniScala Integers

Appetizer:

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

Next time: Full rules of value representation!