Intermediate Representations

Compilers: Principles And Practice

Tiark Rompf

Where Were We?

What did we learn in the last class?

A Taste Of MiniScala

A MiniScala function to compute x to the power of y:

```
def pow(x: Int,  y: Int) =
  if (y == 0) 1                     // pow(x, 0) == 1
  else if (even(y)) {
    val t = pow(x, y /2)            // pow(x, 2z) = pow(pow(x, z), 2)
    t * t
  } else {
    x * pow(x, y - 1)               // pow(x, z + 1) = x * pow(x, z)
  }
```

A Taste Of MiniScala

Say “Hello”:

```
val arr = new Array[Int](5);
arr(0) = ’H’; arr(1) = ’e’; arr(2) = ’l’; arr(3) = ’l’; arr(4) = ’o’;

var i = 0;
while (i < 5) {
  putchar(arr(i));
  i = i + 1
}
```

A Taste Of MiniScala

Our implementation of MiniScala is already quite powerful.

- We can do essentially everything we can do in C
- Missing features (structs, strings): can be implemented as arrays

Are we done yet?

A Taste Of MiniScala

Our implementation of MiniScala is already quite powerful.

- We can do essentially everything we can do in C
- Missing features (structs, strings): can be implemented as arrays

Are we done yet?

- Higher level features: nested first-class functions, objects
- Code quality: optimizations

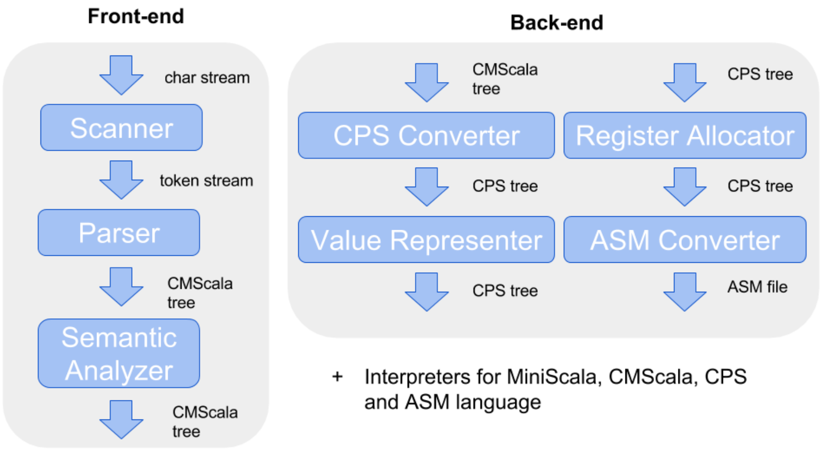
Intermediate Languages

The term intermediate representation (IR) or intermediate language designates the data-structure(s) used by the compiler to represent the program being compiled.

Choosing a good IR is crucial, as many analyses and transformations (e.g. optimizations) are substantially easier to perform on some IRs than on others.

Most non-trivial compilers actually use several IRs during the compilation process, and they tend to become more low-level as the code approaches its final form.

Compiler Architecture



+ Interpreters for MiniScala, CMScala, CPS and ASM language

Intermediate Languages

The revised MiniScala compiler manipulates a total of four languages:

1. MiniScala is the source language that is parsed, but never exists as a tree — it is desugared to CMScala immediately,
2. CMScala - a.k.a. CoreMiniScala - is the desugared version of MiniScala,
3. CPS is the main intermediate language, on which optimizations are performed,
4. ASM is the assembly language of the target (virtual) machine.

The compiler contains interpreters for the last three languages, which is useful to check that a program behaves in the same way as it is undergoes transformation. These interpreters also serve as semantics for their language.

Code Example

To illustrate the differences between the various intermediate representations, we will use a program fragment to compute and print the greatest common divisor (GCD) of 2016 and 714.

The MiniScala version of that fragment could be:

```
def gcd(x: Int, y: Int) =
  if (y == 0)
    printInt(x)
  else
    gcd(y, y % x)

gcd(2016, 714)
```

IR #1: CPS/MiniScala - A Functional IR

A functional IR is an intermediate representation that is close to a (very) simple functional programing language. Typical functional IRs have the following characteristics:

- all primitive operations (e.g. arithmetic operations) are performed on atomic values (variables or constants), and the result of these operations is always named,
- variables cannot be re-assigned.

As we will see later, some of these characteristics are shared with more mainstream IRs, like SSA.

CPS/MiniScala is the functional IR used by the MiniScala compiler.

Local Continuations

A crucial notion in CPS/MiniScala is that of local continuation.

A local continuation is similar to a (local) function but with the following restrictions:

- continuations are not "first class citizens": they cannot be stored in variables or passed as arguments. The only exception being the return continuation (described later),
- continuations never return, and must therefore be invoked in tail position only.

These restrictions enable continuations to be compiled much more efficiently than normal functions. This is the only reason why continuations exist as a separate construct.

Uses Of Continuations

Continuations are used for two purposes in CPS/MiniScala:

1. To represent code blocks which can be “jumped to” from several locations, by invoking the continuation.
2. To represent the code to execute after a function call.

For that purpose, every function gets a continuation as argument, which it must invoke with its return value.

GCD in CPS/MiniScala

The CPS/MiniScala version of the GCD program fragment looks as follows:

```
def gcd(x: Int, y: Int) =
  if (y == 0)
    printInt(x)
  else
    gcd(y, y % x)

gcd(2016, 714)
```

```
def_f gcd(c, x, y) = {
  def_c ct() = { printInt(c, x) };
  def_c cf() = {
    val_p t = x % y;
    gcd(c, y, t);
  }
  val_l z = 0;
  if (y == z) ct else cf
}
val_l x = 2016; val_l y = 714;
gcd(c, x, y)
```

CPS/MiniScala Grammar

```
T ::= val_l N = L; T
    | val_p N = P(N, ...); T
    | def_c N(N, ...) = { T }; T
    | def_f N(N, ...) = { T }; T
    | N(N, ...)
    | if (N C N) N() else N()
    | halt(N)
N ::= name
L ::= integer, character, boolean or unit literal
P ::= '+' | '-' | '*' | '/' | '%' | ...
C ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
```

CPS/MiniScala Local Bindings

val$_l$ `n = l; e`

Binds the name `n` to the literal value `l` in expression `e`. The literal value can be an integer, a character, a boolean or the unit value.

val$_p$ `n = p(n`$_1$`, ...); e`

Binds the name `n` to the result of the application of primitive `p` to the value of `n`$_1$`, ...` in expression `e`. The primitive `p` cannot be a logical (i.e. boolean) primitive, as such primitives are only meant to be used in conditional expressions - see later.

CPS/MiniScala Functions

$\texttt{def}_f$ $\texttt{f}_1\texttt{(c}_1\texttt{, n}_{1,1}\texttt{, ...) = \{ b}_1\texttt{ \}; }$ $\texttt{def}_f$ $\texttt{f}_2\texttt{ ...; e}$

Binds the names $\texttt{f}_1$, ... to functions with arguments $\texttt{n}_{1,1}$, ... and return continuation $\texttt{c}_1$, ... in expression e. The functions can be mutually recursive. The return continuation takes a single argument: the return value. Applying it is interpreted as returning from the function.

$\texttt{f(c, n}_1\texttt{, ...)}$

Applies the function bound to `f` to return continuation `c` and arguments $\texttt{n}_1$, ... The name `c` must either be bound by an enclosing **$\texttt{def}_c$** or be the name of the return continuation of the current function.

CPS/MiniScala Local Continuations

def$_c$ `c`$_1$`(n`$_1$`, ...) = { b`$_1$` }; e`

Binds the names `c`$_1$,... to local continuations with arguments `n`$_1$, ... and body `b`$_1$,... in expression `e`.

Interpretation: like a local function that never returns.

`c(n`$_1$`, ...)`

Applies the continuation bound to `c` to the value of `n`$_1$, ... The name `c` must either be bound by an enclosing **def**$_c$ or be the name of the return continuation of the current function.

Interpretation: if `c` designates a local continuation, a continuation invocation can be seen as a jump with arguments. If `c` designates the current return continuation, a continuation invocation can be seen as a return from the current function, with the given return value.

CPS/MiniScala Control Constructs

```
if (n₁ p n₂) ct else cf
```

Tests whether the condition p is true for the value of n_1 and n_2, then applies continuation ct if it is, or cf if it isn't. Both ct and cf must be parameterless continuations.

The primitive p must be a logical primitive.

Note: if is a branching form of continuation invocation for parameterless continuations. It is therefore a conditional version of `c()`.

```
halt(n)
```

Halts program execution, exiting with the value bound to n (which must be an integer).

CPS/MiniScala Syntactic Sugar

To make CPS/MiniScala programs easier to read and write, we allow to use the postfix and infix notations for the primitive operations.

$\texttt{val}_l$ `t = 1;`
$\texttt{val}_p$ `x = -t + t`

is equivalent to:

$\texttt{val}_l$ `t = 1;`
$\texttt{val}_p$ `x = +(-(t), t)`

Continuation Scopes

The scoping rules of CPS/MiniScala are mostly the “obvious ones”. The only exception is the rule for continuation variables, which are not visible in nested functions!

For example, in the following code:

```
def_c c_0(r) = printInt(r);
def_f f(c_1, x) = {
  val_p t = x + x;
  c_1(t)
}
```

c_0 is not visible in the body of `f`!

This guarantees that continuations are truly local to the function that defines them, and can therefore be compiled efficiently.

GCD in CPS/MiniScala

The CPS/MiniScala version of the GCD program fragment looks as follows:

```
def gcd(x: Int, y: Int) =
  if (y == 0)
    printInt(x)
  else
    gcd(y, y % x)

gcd(2016, 714)
```

```
def_f gcd(c, x, y) = {
  def_c ct() = { printInt(c, x) };
  def_c cf() = {
    val_p t = x % y;
    gcd(c, y, t);
  }
  val_l z = 0;
  if (y == z) ct else cf
}
val_l x = 2016; val_l y = 714;
gcd(c, x, y)
```

Translating CMScala to CPS/MiniScala

The translation from CMScala to CPS/MiniScala is specified as a function denoted by $[\![.]\!]$ and taking two arguments:

1. `T`, the CMScala term to be translated,
2. `C`, the context, a CPS/MiniScala term containing a hole into which a name bound to the value of the translated term has to be plugged.

This function is written in a "mixfix" notation, as follows:

$[\![\texttt{T}]\!]\ \texttt{C}$

The translation function must return a CPS/MiniScala term including both the translation of the CMScala term `T` and the context `C` with its hole plugged by a name bound to the value of (the translation of) `T`.

Translation Context

The translation context is a CPS/MiniScala term representing the partial translation of the CMScala expression surrounding the one being translated.

This term contains a single hole, written $\square$, representing the currently unknown name that will be bound to the value of the expression being translated.

The hole of a context `C` must eventually be plugged with some name `n`, written as `C[n]`.

For example, the context `f(`$\square$`)` plugged with name `m` results in the term `f(m)`.

Context Representation

Translation rules often build contexts that include other contexts. One (fictional) example could be:

$$[\![e_1]\!]\ \texttt{(}[\![e_2]\!]\ \Box\texttt{(}\Box\texttt{))}$$

In such situations, using the anonymous hole ($\Box$) is ambiguous. To lift the ambiguity, we represent contexts as meta-functions taking a single (named) argument. The above is therefore written as

$$[\![e_1]\!]\ \lambda v_1\texttt{(}[\![e_2]\!]\ \lambda v_2\texttt{(}v_1\texttt{(}v_2\texttt{)))}$$

lifting the ambiguity.

With such a representation of contexts, filling the hole of some context `C` with some name `n` is done by meta-function application - still written `C[n]`.

The Translation In Scala

In Scala - the meta-language in the MiniScala project - the translation function $[\![.]\!]$ is defined as a function with the following profile:

```scala
def CMScalaToCPS(t: CMScalaTree,
  c: Symbol => CPSTree): CPSTree
```

In the body of that function, plugging the context `c` with a name (i.e. a Symbol) bound to a Scala value named `n` is done using Scala function application:

```scala
c(n)
```

To clarify the presentation, CMScala terms appear in green, CPS/MiniScala terms in orange, and meta-terms in black.

CMScala to CPS/MiniScala Translation (1)

Note: in the following expressions, all underlined names are fresh.

```
$[\![n]\!]$ C = where n is an indentifier for immutable variable
   C[n]

$[\![l]\!]$ C = where l is a literal
  $val_l$ $\underline{n}$ = l; C[n]

$[\![$ val $n_1$ = $e_1$; e $]\!]$ C =
  $[\![e_1]\!]$($\lambda$v ($val_p$ $n_1$ = id(v); $[\![e]\!]$ C))

$[\![$ var $n_1$ = $e_1$; e $]\!]$ C =
  $val_l$ $\underline{s}$ = 1;
  $val_p$ $n_1$ = block-alloc-242(s);
  $val_l$ $\underline{z}$ = 0;
  $[\![e_1]\!]$($\lambda$v ($val_p$ $\underline{d}$ = block-set($n_1$, z, v); $[\![e]\!]$ C))
```

CMScala to CPS/MiniScala Translation (2)

```
⟦ n₁ = e₁ ⟧ C
  val_l z = 0;
  ⟦e₁⟧(λv (val_p d = block-set(n₁, z, v); C[v] ))

⟦n⟧ C = where n is an indentifier for mutable variable
  val_l z = 0;
  val_p v = block-get(n, z); C[v]

⟦ def f₁(n₁,₁: _, ...) = e₁; def ...  ; e⟧ C =
  def_f f₁(c, n₁,₁, ...) = {
    ⟦e₁⟧(λv(c(v)))
  }; def_f ...;
  ⟦e⟧ C
```

CMScala to CPS/MiniScala Translation (3)

```
$\llbracket$ e(e$_1$, e$_2$, ...) $\rrbracket$ C =
  $\llbracket$e$\rrbracket$($\lambda$v($\llbracket$e$_1\rrbracket$($\lambda$v$_1$($\llbracket$e$_2\rrbracket$($\lambda$v$_2$ ...)))));
  def$_c$ $\underline{\text{c}}$($\underline{\text{r}}$) = { C[r] };
  v(c, v$_1$, v$_2$ ...)
```

```
$\llbracket$ if (p(e$_1$, ...)) e$_2$ else e$_3$ $\rrbracket$ C where p is a logical primitive =
  def$_c$ $\underline{\text{c}}$($\underline{\text{r}}$) = { C[r] };
  def$_c$ $\underline{\text{ct}}$() = { $\llbracket$e$_2\rrbracket$($\lambda$v$_2$(c(v$_2$))) };
  def$_c$ $\underline{\text{cf}}$() = { $\llbracket$e$_3\rrbracket$($\lambda$v$_3$(c(v$_3$))) };
  $\llbracket$e$_1\rrbracket$($\lambda$v$_1$(... (if (p(v$_1$ ...)) ct else cf)))
```

CMScala to CPS/MiniScala Translation (4)

```
⟦ if (e1) e2 else e3 ⟧ C =
  def_c c(r) = { C[r] };
  def_c ct() = { ⟦e2⟧(λv2(c(v2))) };
  def_c cf() = { ⟦e3⟧(λv3(c(v3))) };
  val_l f = false;
  ⟦e1⟧(λv1(if (v1 != f) ct else cf))

⟦ while (e1) e2; e3 ⟧ C =
  def_c loop() = {
    def_c c() = { ⟦e3⟧ C };
    def_c ct() = { ⟦e2⟧(λv(loop())) };
    val_l f = false;
    ⟦e1⟧(λv(if (v != f) ct else c))
  };
  loop()
```

CMScala to CPS/MiniScala Translation (5)

$[\![$ `p( e`$_1$`, e`$_2$`, ...)` $]\!]$ `C where p is a logical primitive =`

  $[\![$ `if (p(e`$_1$`, e`$_2$`, ...)) true else false` $]\!]$ `C`

$[\![$ `p( e`$_1$`, e`$_2$`, ...)` $]\!]$ `C where p is not a logical primitive =`

  `left as an exercise`

Initial Context

In which context should a complete program be translated?

The simplest answer is a context that halts execution with an exit code of 0 (no error), that is:

λv(`val`$_l$ `z = 0; halt(z)`)

An alternative would be to do something with the value v produced by the whole program, e.g. use it as the exit code instead of 0, print it, etc.

Improving The Translation

The translation presented before has two shortcomings:

1. it produces terms containing useless continuations, and
2. it produces suboptimal CPS/MiniScala code for some conditionals.

One solution to improve the translation is to define several different translations depending on the source (i.e. MiniScala) context in which the expression to translate appears.

Useless Continuations

The first problem can be illustrated with the MiniScala term:

```
def f(g: () => Int) = g(); f
```

which - in the empty context - gets translated to:

```
def_f f(c, g) = {
  def_c j(r) = { c(r) };
  g(j)
};
f
```

instead of the equivalent and more compact:

```
def_f f(c, g) = { g(c) };
f
```

Suboptimal Conditionals (1)

The second problem can be illustrated with the MiniScala term:

```
if (if (a) b else false) x else y
```

which, in the empty context, gets translated to:

```
def_c ci_1(v_1) = { v_1 };
def_c ct_1() = { ci_1(x) };
def_c cf_1() = { ci_1(y) };
val_l f_1 = false;
def_c ci_2(v_2) = {
  if (v_2 != f_1) ct_1 else cf_1 };
def_c ct_2() = { ci_2(b) };
def_c cf_2() = {
  val_l i_1 = false; ci_2(i_1) };
val_l f_2 = false;
if (a != f_2) ct_2 else cf_2
```

Suboptimal Conditionals (2)

A much better translation for:

```
if (if (a) b else false) x else y
```

would be:

```
def_c ci_1(v_1) = { v_1 };
def_c ct_1() = { ci_1(x) };
def_c cf_1() = { ci_1(y) };
def_c ca_1() = {
  val_l i_1 = false;
  if (b != i_1) ct_1 else cf_1 };
val_l i_2 = false;
if (a != i_2) ca_1 else cf_1
```

which immediately applies continuation $\texttt{cf}_1$ if a is false.

Source Contexts

These two problems have in common the fact that the translation could be better if it depended on the source context in which the expression to translate appears.

- In the first example, the function call could be translated more efficiently because it appears as the last expression of the function (i.e. it is in **tail position**).
- For the second example, the nested if expression could be translated more efficiently because it appears in the condition of another if expression and one of its branches is a simple boolean literal (here `false`).

Therefore, instead of having one translation function, we should have several: one per source context worth considering!

A Better Translation

To solve the two problems, we split the single translation function into three separate ones:

1. $[\![.]\!]_N$ `C`, taking as before a term to translate and a context `C`, whose hole must be plugged with a name bound to the value of the term.
2. $[\![.]\!]_T$ `c`, taking a term to translate and a one-parameter continuation `c`. This continuation is to be applied to the value of the term.
3. $[\![.]\!]_C$ `ct cf`, taking a term to translate and two parameterless continuations, `ct` and `cf`. The continuation `ct` is to be applied when the term evaluates to a true value, while the continuation `cf` is to be applied when it evaluates to a false value.

The Non-tail Translation

$[\![.]\!]_N$ is called the **non-tail** translation as it is used in non-tail contexts. That is, when the work that has to be done once the term is evaluated is more complex than simply applying a continuation to the term's value.

For example, the arguments of a primitive are always in a non-tail context, since once they are evaluated, the primitive has to be applied on their value:

```
$[\![$ p(e$_1$, e$_2$, ...) $]\!]_N$ C where p is not a logical primitive =
  $[\![$e$_1$$]\!]_N$($\lambda$v$_1$($[\![$e$_2$$]\!]_N$($\lambda$v$_2$...)));
  val$_p$ $\underline{\text{n}}$ = p(v$_1$, v$_2$, ...);
  C[n]
```

The Tail Translation

The tail translation $[\![.]\!]_T$ is used whenever the context passed to the simple translation has the form $\lambda\texttt{v(c(v))}$. It gets as argument the name of the continuation `c` to which the value of expression should be applied.

For example, the previous translation of function definition:

$$\begin{array}{l}
[\![\ \texttt{def}\ \texttt{f}_1\texttt{(n}_{1,1}\texttt{: \_, ...) = e}_1\texttt{; ...\ \ ; e}\ ]\!]\ \texttt{C} = \\
\quad \texttt{def}_f\ \texttt{f}_1\texttt{(}\underline{\texttt{c}}\texttt{, n}_{1,1}\texttt{, ...) = \{ } [\![\texttt{e}_1]\!]\texttt{(}\lambda\texttt{v(c(v))) \};} \\
\quad \texttt{def}_f\ \texttt{...;} \\
\quad [\![\texttt{e}]\!]\ \texttt{C}
\end{array}$$

becomes:

$$\begin{array}{l}
[\![\ \texttt{def}\ \texttt{f}_1\texttt{(n}_{1,1}\texttt{: \_, ...) = e}_1\texttt{; def ...\ \ ; e}\ ]\!]_N\ \texttt{C} = \\
\quad \texttt{def}_f\ \texttt{f}_1\texttt{(}\underline{\texttt{c}}\texttt{, n}_{1,1}\texttt{, ...) = \{ } [\![\texttt{e}_1]\!]_T\ \texttt{c \};} \\
\quad \texttt{def}_f\ \texttt{...;} \\
\quad [\![\texttt{e}]\!]_N\ \texttt{C}
\end{array}$$

The Cond Translation (1)

The cond translation $[\![.]\!]_C$ is used whenever the term to translate is a condition to be tested to decide how execution must proceed. It gets two continuations as arguments: the first is to be applied when the condition is true, while the second is to be applied when it is false.

This translation is used to handle the condition of an if expression:

```
$[\![$ if (e$_1$) e$_2$ else e$_3$ $]\!]_N$ C =
  def$_c$ $\underline{\texttt{c}}$($\underline{\texttt{r}}$) = { C[r] };
  def$_c$ $\underline{\texttt{ct}}$() = { $[\![$e$_2]\!]_T$ c };
  def$_c$ $\underline{\texttt{cf}}$() = { $[\![$e$_3]\!]_T$ c };
  $[\![$e$_1]\!]_C$ ct cf
```

The Cond Translation (2)

Having a separate translation for conditional expressions makes the efficient compilation of conditionals with literals in one of their branch possible:

$$
\begin{array}{l}
\llbracket\ \texttt{if (}\mathtt{e}_1\texttt{) }\mathtt{e}_2\texttt{ else false}\ \rrbracket_C\ \texttt{ct cf =} \\
\quad \texttt{def}_c\ \underline{\texttt{ac}}\texttt{() = \{ }\llbracket \mathtt{e}_2 \rrbracket_C\ \texttt{ct cf \};} \\
\quad \llbracket \mathtt{e}_1 \rrbracket_C\ \texttt{ac cf;} \\
\\
\llbracket\ \texttt{if (}\mathtt{e}_1\texttt{) false else true}\ \rrbracket_C\ \texttt{ct cf =} \\
\quad \llbracket \mathtt{e}_1 \rrbracket_C\ \texttt{cf ct}
\end{array}
$$

. . . and so on for all conditionals with at least one constant branch.

The Cond Translation (3)

```
$[\![$ while ($e_1$) $e_2$; $e_3$ $]\!]_N$ C =
  $\text{def}_c$ $\underline{\text{loop}}$($\underline{\text{d}}$) = {
    $\text{def}_c$ $\underline{\text{c}}$() = { $[\![e_3]\!]_N$ C };
    $\text{def}_c$ $\underline{\text{ct}}$() = { $[\![e_2]\!]_T$ loop };
    $[\![e_1]\!]_C$ ct c
  };
  $\text{val}_l$ $\underline{\text{d}}$ = ();
  loop(d)
```

The Better Translation In Scala

In the compiler, the three translations are simply three mutually-recursive functions, with the following profiles:

```
def nonTail(t: CMScalaTree)
           (c: Symbol => CPSTree): CPSTree

def tail(t: CMScalaTree,
         c: Symbol):CPSTree

def cond(t: CMScalaTree,
         ct: Symbol,
         cf: Symbol): CPSTree
```

IR #2: Standard RTL/CFG - Register-transfer Language

A register-transfer language (RTL) is a kind of intermediate representation in which most operations compute a function of several virtual registers (i.e. variables) and store the result in another virtual register.

For example, the instruction adding variables y and z, storing the result in x could be written $x \leftarrow y + z$. Such instructions are sometimes called quadruples, because they typically have four components: the three variables (x, y and z here) and the operation (+ here).

RTLs are very close to assembly languages, the main difference being that the number of virtual registers is usually not bounded.

Control-Flow Graph

A control-flow graph (CFG) is a directed graph whose nodes are the individual instructions of a function, and whose edges represent control-flow.

More precisely, there is an edge in the CFG from a node n_1 to a node n_2 if and only if the instruction of n_2 can be executed immediately after the instruction of n_1.

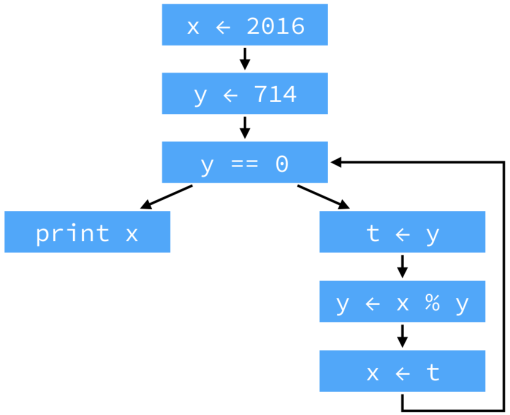
RTL/CFG

RTL/CFG is the name given to intermediate representations where each function of the program is represented as a control-flow graph whose node contain RTL instructions.

This kind of representation is very common in the later stages of compilers, especially those for imperative languages.

RTL/CFG Example

Computation of the GCD of 2016 and 714 in a typical RTL/CFG representation.



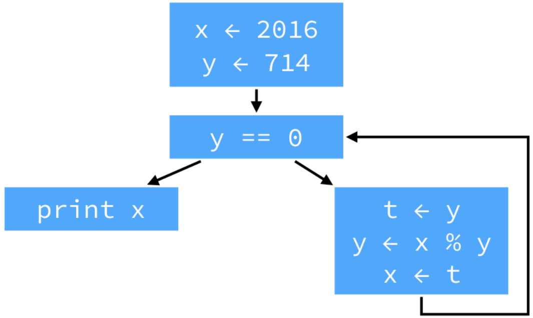
Basic Blocks

A basic block is a maximal sequence of instruction for which control can only enter through the first instruction of the block and leave through the last.

Basic blocks are sometimes used as the nodes of the CFG, instead of individual instructions. This has the advantage of reducing the number of nodes in the CFG, but also complicates data-flow analyses. It is therefore far from being clear that basic blocks are still useful today.

RTL/CFG Example

The same examples as before, but with basic blocks instead of individual 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.

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

An RTL/CFG program is said to be in static single-assignment (SSA) form if each variable has only one definition in the program.

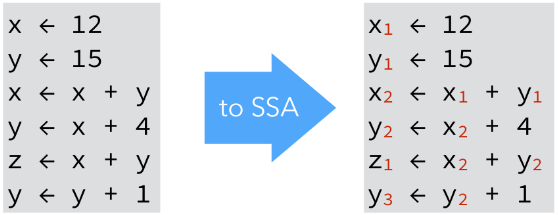
That single definition can be executed many times when the program is run - if it is inside a loop - hence the qualifier static.

SSA form is popular because it simplifies several optimizations and analysis, as we will see.

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

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:



```
x ← 12
y ← 15
x ← x + y
y ← x + 4
z ← x + y
y ← y + 1
```

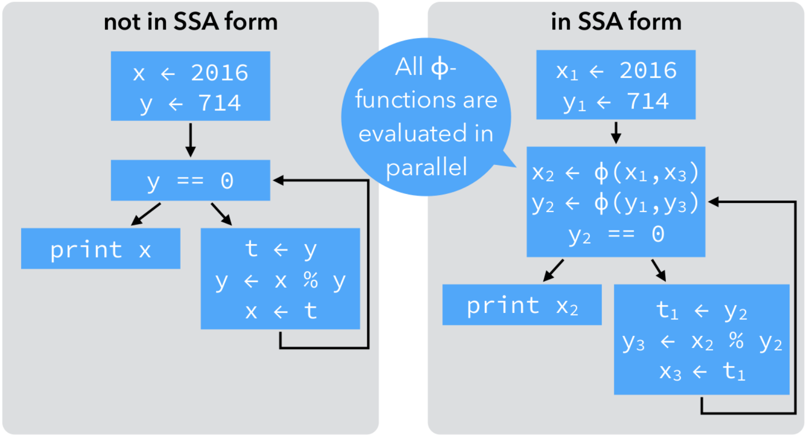
$x_1 \leftarrow 12$
$y_1 \leftarrow 15$
$x_2 \leftarrow x_1 + y_1$
$y_2 \leftarrow x_2 + 4$
$z_1 \leftarrow x_2 + y_2$
$y_3 \leftarrow y_2 + 1$

Φ-function

Join-points in the CFG - nodes with more than one predecessors - are more problematic, as each predecessor can bring its own version of a given name.

To reconcile those different versions, a fictional Φ-function is introduced at the join point. That function takes as argument all the versions of the variable to reconcile, and automatically selects the right one depending on the flow of control.

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!

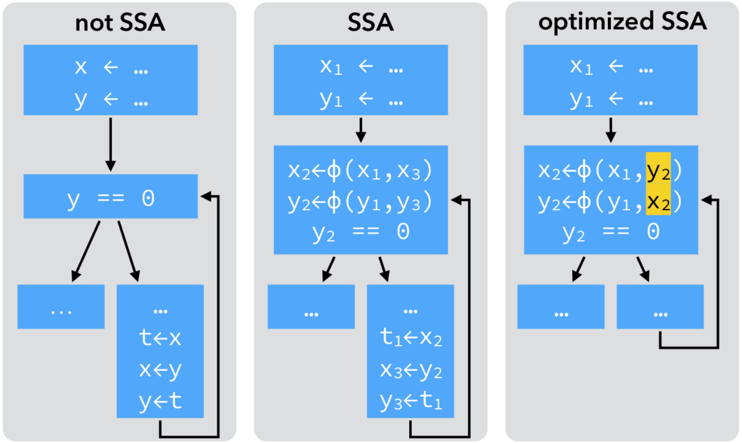
To make this clear, some authors write Φ-functions in matrix form, with one row per predecessor:

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

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

Evaluation of Φ-functions

The following loop extract illustrates why Φ-functions must be evaluated in parallel.

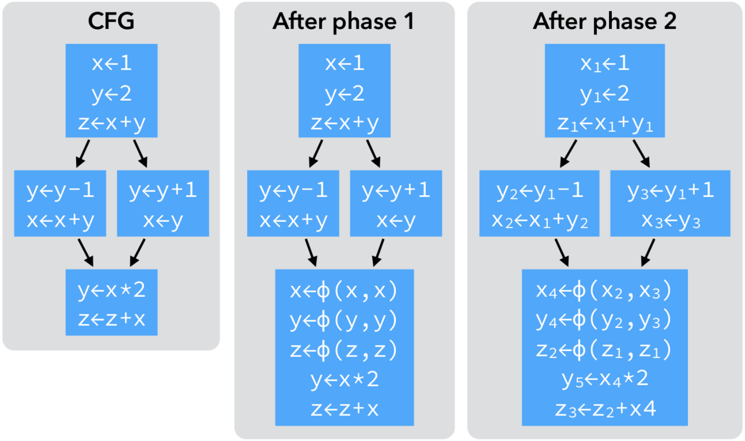


(Naive) Building of SSA Form

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



CFG

$x \leftarrow 1$, $y \leftarrow 2$, $z \leftarrow x+y$

$y \leftarrow y-1$, $x \leftarrow x+y$ | $y \leftarrow y+1$, $x \leftarrow y$

$y \leftarrow x*2$, $z \leftarrow z+x$

After phase 1

$x \leftarrow 1$, $y \leftarrow 2$, $z \leftarrow x+y$

$y \leftarrow y-1$, $x \leftarrow x+y$ | $y \leftarrow y+1$, $x \leftarrow y$

$x \leftarrow \phi(x,x)$, $y \leftarrow \phi(y,y)$, $z \leftarrow \phi(z,z)$, $y \leftarrow x*2$, $z \leftarrow z+x$

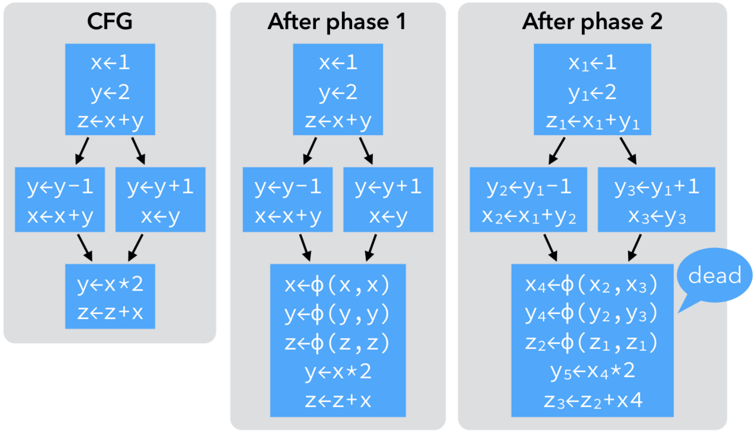
After phase 2

$x_1 \leftarrow 1$, $y_1 \leftarrow 2$, $z_1 \leftarrow x_1+y_1$

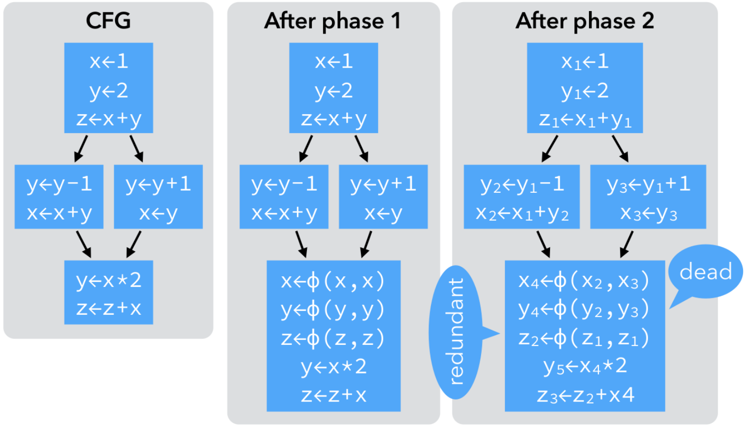
$y_2 \leftarrow y_1-1$, $x_2 \leftarrow x_1+y_2$ | $y_3 \leftarrow y_1+1$, $x_3 \leftarrow y_3$

$x_4 \leftarrow \phi(x_2,x_3)$, $y_4 \leftarrow \phi(y_2,y_3)$, $z_2 \leftarrow \phi(z_1,z_1)$, $y_5 \leftarrow x_4*2$, $z_3 \leftarrow z_2+x4$

(Naive) Building of SSA Form



(Naive) Building of SSA Form



Better Building Techniques

The naive technique just presented works, in the sense that the resulting program is in SSA form and is equivalent to the original one.

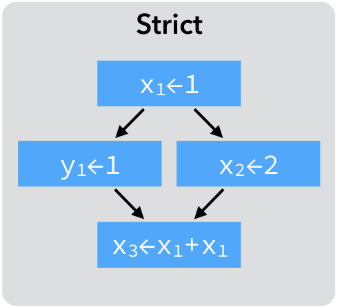
However, it introduces too many Φ-functions - some dead, some redundant - to be useful in practice. It builds the **maximal** SSA form.

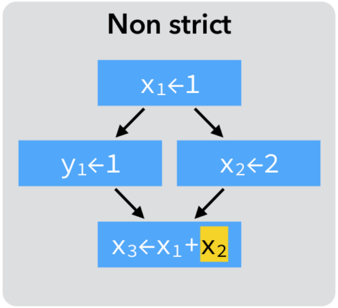
Better techniques exist to translate a program to SSA form.

Strict SSA Form

A program is said to be in strict SSA form if it is in SSA form and 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 .)

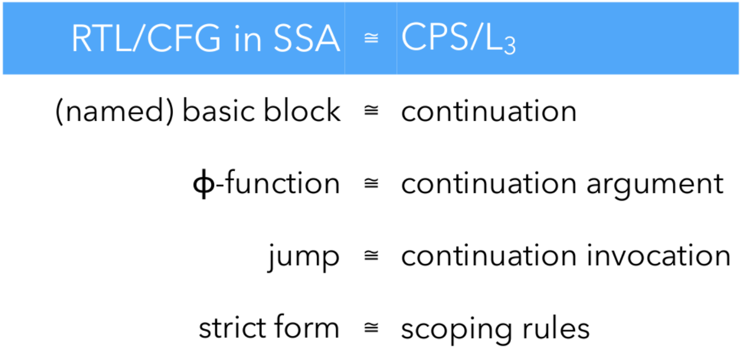
Strict SSA form guarantees that no variable is used before being defined.





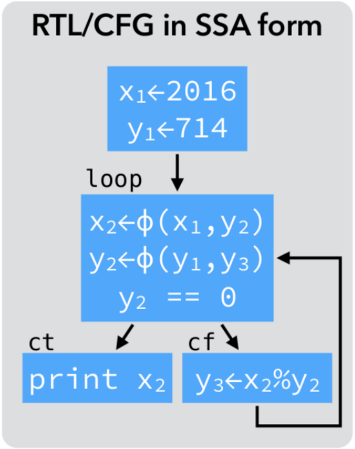
Comparing IRs: CPS/MiniScala vs RTL/CFG in SSA

As the correspondences in the table below illustrate, CPS/MiniScala is very close to RTL/CFG in SSA form.



RTL/CFG in SSA	≅	CPS/L_3
(named) basic block	≅	continuation
ϕ-function	≅	continuation argument
jump	≅	continuation invocation
strict form	≅	scoping rules

CPS/MiniScala vs RTL/CFG in SSA



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

Summary And References

Claim: continuation-based, functional IRs like CPS/MiniScala are SSA done right, and 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.)

* * *

CPS/MiniScala 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.