Intermediate representations

Advanced Compiler Construction
Michel Schinz – 2013-02-28

Intermediate representations

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.

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 L_3 version of that fragment could be:

```
(rec loop ((x 2016)
           (y 714))
    (if (= 0 y)
        (int-print x)
        (loop y (% x y))))
```

IR #1: CPS/L_3

CPS/L$_3$: 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/L$_3$ is the functional IR used by the L$_3$ compiler.

Local continuations

A crucial notion in CPS/L$_3$ 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/L$_3$:

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.

CPS/L$_3$ grammar

```
T ::= (letₗ ((N L)) T)
    | (letₚ ((N (P N …))) T)
    | (letₖ ((N (cont (N …) T))) T)
    | (letf ((N (fun (N N …) T)) …) T)
    | (appₖ N N …)
    | (appf N N N …)
    | (if (C N N) N N)
    | (halt)
N ::= name
L ::= integer, character, boolean or unit literal
P ::= + | - | * | / | % | …
C ::= < | <= | = | != | >= | >
```

CPS/L3 local bindings

($\mathbf{let_l}$ ((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.

($\mathbf{let_p}$ ((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/L3 functions

($\mathbf{let_f}$ ((f_1 (**fun** (k_1 $n_{1,1}$…) b_1)) …) e)

Binds the names f_1, … to functions with arguments $n_{1,1}$, … and return continuation k_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.

($\mathbf{app_f}$ f k n_1 …)

Applies the function bound to f to return continuation k and arguments n_1, … The name k must either be bound by an enclosing $\mathbf{let_k}$ or be the name of the return continuation of the current function.

CPS/L3 local continuations

($\mathbf{let_k}$ ((k (**cont** (n_1 …) b))) e)

Binds the name k to a local continuation with arguments n_1, … and body b in expression e. Interpretation: similar to a local function that never returns.

($\mathbf{app_k}$ k n_1 …)

Applies the continuation bound to k to the value of n_1, … The name k must either be bound by an enclosing $\mathbf{let_k}$ or be the name of the return continuation of the current function. Interpretation: if k designates a local continuation, $\mathbf{app_k}$ can be seen as a jump with arguments. If k designates the current return continuation, $\mathbf{app_k}$ can be seen as a return from the current function, with the given return value.

CPS/L3 control constructs

(**if** (p n_1 n_2) k_t k_f)

Tests whether the condition p is true for the value of n_1 and n_2, then applies continuation k_t if it is, or k_f if it isn't. Both k_t and k_f must be parameterless continuations. The primitive p must be a logical primitive.

(**halt**)

Halts program execution.

Continuation visibility

The scoping rules of CPS/L_3 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:

```
(letk ((k0 (cont (r) (appf print r))))
  (letf ((f (fun (k1 x)
               (letp ((t (+ x x)))
                 (k1 t)))))))
```

k0 is **not visible** in the body of f!

This guarantees that continuations are truly local and can therefore be compiled efficiently.

CPS/L_3 syntactic sugar

To make CPS/L_3 programs easier to read and write, we allow the collapsing of nested occurrences of letl, letp and letk expressions to a single let* expression.
We also allow the elision of appf and appk in applications.
Example:

```
(letl ((c1 1))
  (letl ((c2 2))
    (letp ((c3 (+ c1 c2)))
      … (appf f k))))

              ⇕

(let* ((c1 1)
       (c2 2)
       (c3 (+ c1 c2)))
  … (f k))
```

GCD in CPS/L_3

The CPS/L3 version of the GCD program fragment looks as follows:

```
(letk ((loop
        (cont (x y)
          (let* ((kt (cont ()
                        (appf print x)))
                 (kf (cont ()
                        (letp ((t (% x y)))
                          (appk loop y t))))
                 (z 0))
            (if (= y z) kt kf)))))
  (let* ((x 2016) (y 714))
    (appk loop x y)))
```
(To simplify, no return continuation is passed to print).

Translation of CL$_3$ to CPS/L_3

Translating CL_3 to CPS/L_3

The translation from CL_3 to CPS/L_3 is performed by a function denoted by $[\![\cdot]\!]$ and taking two arguments:

1. T, the CL_3 term to be translated,
2. C, the CPS/L_3 context – i.e. a CPS/L_3 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:

$[\![T]\!]$ C

Translation context

The translation rules often nest contexts. Using the anonymous hole $\square$ is ambiguous in this case, e.g.:

$(\text{app}_f\ \square\ \square)$

For that reason, we represent contexts as metafunctions taking a single argument, the value with which the hole has to be plugged. The above context can be represented as the following metafunction:

$\lambda k\, \lambda f\, (\text{app}_f\ f\ k)$

Plugging the hole of this context with the name n is done using metafunction application, which we denote as C{n}.

The translation in Scala

In Scala, the translation function $[\![\cdot]\!]$ is defined as a function with the following profile:

```
def CL3ToCPS(t: CL3Tree,
             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 n is done using Scala function application:

```
c(n)
```

CL_3 to CPS/L_3 translation (1)

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

$[\![n]\!]$ C *where* n *is a name* =
    C{n}

$[\![l]\!]$ C *where* l *is a literal value* =
    $(\text{let}_l\ ((\underline{n}\ l))\ C\{n\})$

$[\![(\textbf{let}\ ((n_1\ e_1)\ (n_2\ e_2)\ ...)\ e)]\!]$ C =
    $[\![(\textbf{let}\ ((n_1\ e_1))\ (\textbf{let}\ ((n_2\ e_2)\ ...)\ e))]\!]$ C

$[\![(\textbf{let}\ ((n_1\ e_1))\ e)]\!]$ C =
    $(\text{let}_k\ ((\underline{k}\ (\textbf{cont}\ (n_1)\ [\![e]\!]\ C)))$
        $[\![e_1]\!](\lambda v\ (\textbf{app}_k\ k\ v)))$

Only valid when names are globally unique (as symbols are in the compiler)

CL₃ to CPS/L₃ translation (2)

⟦(**letrec** ((f₁ (**fun** (n₁,₁ n₁,₂ …) e₁)) …) e)⟧ C =
  (**let_f** ((f₁ (**fun** (k̲ n₁,₁ n₁,₂ …)
               ⟦e₁⟧(λv (**app_k** k v)))) …)
    ⟦e⟧ C)
⟦(e e₁ e₂ …)⟧ C =
  ⟦e⟧(λv ⟦e₁⟧(λv₁ ⟦e₂⟧(λv₂ …
    (**let_k** ((k̲ (**cont** (r̲) C{r})))
      (**app_f** v k v₁ v₂ …)))))

CL₃ to CPS/L₃ translation (3)

⟦(if (p e₁ …) e₂ e₃)⟧ C *where* p *is a logical primitive* =
  (let_k ((k̲ (cont (r̲) C{r})))
    (let_k ((k̲t̲ (cont () ⟦e₂⟧(λv₂ (app_k k v₂)))))
      (let_k ((k̲f̲ (cont () ⟦e₃⟧(λv₃ (app_k k v₃)))))
        ⟦e₁⟧(λv₁ … (if (p v₁ …) kt kf))))))
⟦(if e₁ e₂ e₃)⟧ C =
  (let_k ((k̲ (cont (r̲) C{r})))
    (let_k ((k̲t̲ (cont () ⟦e₂⟧(λv₂ (app_k k v₂)))))
      (let_k ((k̲f̲ (cont () ⟦e₃⟧(λv₃ (app_k k v₃)))))
        (let_l ((f̲ #f))
          ⟦e₁⟧(λv₁ (if (!= v₁ f) kt kf)))))))

CL₃ to CPS/L₃ translation (4)

⟦(@ p e₁ e₂ …)⟧ C *where* p *is a logical primitive* =
  ⟦(if (p e₁ e₂ …) #t #f)⟧ C
⟦(@ p e₁ e₂ …)⟧ 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, that is:
  λv (halt)
An alternative would be to do something with the value v produced by the whole program, e.g. print it or use it as a return code.

Exercise

What is the translation of the following L_3 expression:
```
(letrec ((apply-to-succ (fun (f x)
                             (f (@ + x 1)))))
  apply-to-succ)
```
in the empty context □ (i.e. λx x)?

Better translation of CL$_3$ to CPS/L$_3$

Improving the translation

The translation presented before has two shortcomings:
1. it produces terms containing useless continuations, and
2. it produces suboptimal CPS/L$_3$ code for some conditionals.

One solution to improve the translation is to define several different translations depending on the source (i.e. L_3) context in which the expression to translate appears.

Useless continuations

The first problem can be illustrated with the L_3 term:
```
(letrec ((f (fun (g) (g)))) f)
```
which, in the empty context, gets translated to:
```
(letf ((f (fun (k g)
            (letk ((j (cont (r)
                          (appk k r))))
              (appf g j)))))
  f)
```
instead of the equivalent and more compact:
```
(letf ((f (fun (k g) (appf g k))))
  f)
```

Suboptimal conditionals (1)

The second problem can be illustrated with the L_3 term:
```
(if (if a b #f) x y)
```
which, in the empty context, gets translated to:
```
(let* ((ki1 (cont (v1) v1))
       (kt1 (cont () (appₖ ik1 x)))
       (kf1 (cont () (appₖ ik1 y)))
       (f1 #f)
       (ki2 (cont (v2)
               (if (!= v2 f1) kt1 kf1)))
       (kt2 (cont () (appₖ ki2 b)))
       (kf2 (cont ()
               (let ((i1 #f)) (appₖ ki2 i1))))
       (f2 #f))
  (if (!= a f2) kt2 kf2))
```

Suboptimal conditionals (2)

A much better translation for:
```
(if (if a b #f) x y)
```
would be:
```
(let* ((ki1 (cont (v1) v1))
       (kt1 (cont () (appₖ ki1 x)))
       (kf1 (cont () (appₖ ki1 y)))
       (ka1 (cont ()
                (let ((i1 #f))
                  (if (!= b i1) kt1 kf1))))
       (i2 #f))
  (if (!= a i2) ka1 kf1))
```
which immediately applies continuation **kf1** 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 **#f**).

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. $[\![\cdot]\!]_N$ 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. $[\![\cdot]\!]_T$ k, taking a term to translate and a one-parameter continuation k. This continuation is to be applied to the value of the term.
3. $[\![\cdot]\!]_C$ k_t k_f, taking a term to translate and two parameterless continuations, k_t and k_f. The continuation k_t is to be applied when the term evaluates to a true value, while the continuation k_f is to be applied when it evaluates to a false value.

The *non-tail* transformation

$[\![\cdot]\!]_N$ 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:

$[\![(@\ p\ e_1\ e_2\ \dots)]\!]_N\ C$ *where* p *is not a logical primitive* =
$\quad [\![e_1]\!]_N(\lambda v_1\ [\![e_2]\!]_N(\lambda v_2 \dots$
$\qquad (let_p\ ((\underline{n}\ (p\ v_1\ v_2\ \dots)))$
$\qquad\quad C\{\underline{n}\})))$

The *tail* transformation

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

For example, the previous translation of function definition:

$[\![(letrec\ ((f_1\ (fun\ (n_{1,1}\ n_{1,2}\ \dots)\ e_1))\ \dots)\ e)]\!]\ C =$
$\quad (let_f\ ((f_1\ (fun\ (k\ n_{1,1}\ n_{1,2}\ \dots)$
$\qquad\qquad [\![e_1]\!](\lambda v\ (app_k\ k\ v))))\ \dots)$
$\qquad [\![e]\!]\ C)$

becomes:

$[\![(letrec\ ((f_1\ (fun\ (n_{1,1}\ n_{1,2}\ \dots)\ e_1))\ \dots)\ e)]\!]_N\ C =$
$\quad (let_f\ ((f_1\ (fun\ (k\ n_{1,1}\ n_{1,2}\ \dots)$
$\qquad\qquad [\![e_1]\!]_T\ k))\ \dots)$
$\qquad [\![e]\!]_N\ C)$

The *cond* translation (1)

The **cond** translation $[\![\cdot]\!]_C$ 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:

$[\![(if\ e_1\ e_2\ e_3)]\!]_N\ C =$
$\quad (let_k\ ((\underline{k}\ (cont\ (\underline{r})\ C\{\underline{r}\})))$
$\qquad (let_k\ ((\underline{kt}\ (cont\ ()\ [\![e_2]\!]_T\ k)))$
$\qquad\quad (let_k\ ((\underline{kf}\ (cont\ ()\ [\![e_3]\!]_T\ k)))$
$\qquad [\![e_1]\!]_C\ kt\ kf))))$

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:

$[\![(if\ e_1\ e_2\ \#f)]\!]_C\ k_t\ k_f =$
$\quad (let_k\ ((\underline{ak}\ (cont\ ()\ [\![e_2]\!]_C\ k_t\ k_f)))$
$\qquad [\![e_1]\!]_C\ ak\ k_f)$
$[\![(if\ e_1\ \#f\ \#t)]\!]_C\ k_t\ k_f =$
$\quad [\![e_1]\!]_C\ k_f\ k_t$

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

The new translation in Scala

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

```scala
def nonTail(t: CL3Tree,
            c: Symbol⇒CPSTree): CPSTree

def tail(t: CL3Tree, k: Symbol): CPSTree
def cond(t: CL3Tree,
         kt: Symbol,
         kf: 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.
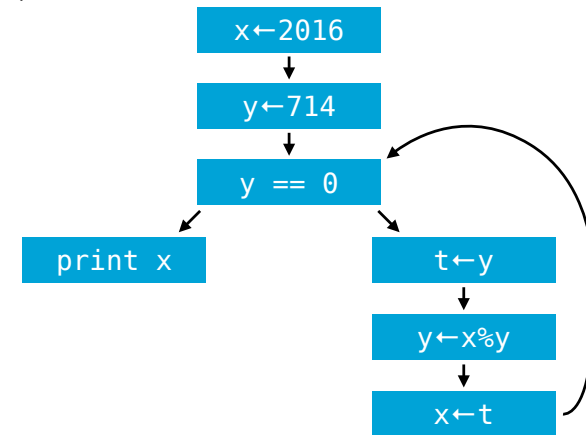
RTL/CFG

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.

```
x←2016
y←714
y == 0
print x      t←y
             y←x%y
             x←t
```
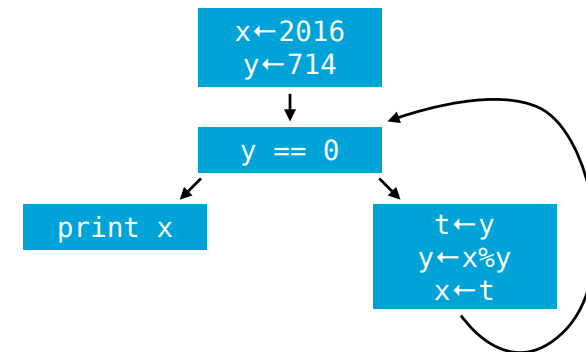
Basic blocks

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.

```
x←2016
y←714
y == 0
print x      t←y
             y←x%y
             x←t
```

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

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:

$$x \leftarrow 12$$
$$y \leftarrow 15$$
$$x \leftarrow x+y$$
$$y \leftarrow x+4$$
$$z \leftarrow x+y$$
$$y \leftarrow y+1$$

to SSA

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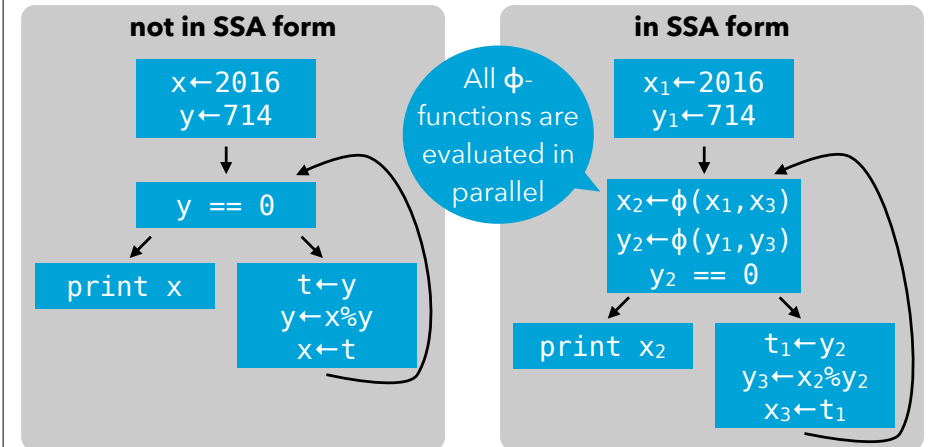$$y_3 \leftarrow y_2+1$$

ɸ-functions

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 **ɸ-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.

ɸ-functions example

not in SSA form

```
x←2016
y←714
```

```
y == 0
```

```
print x
```

```
t←y
y←x%y
x←t
```

All ɸ-functions are evaluated in parallel

in SSA form

```
x₁←2016
y₁←714
```

$$x_2 \leftarrow \phi(x_1, x_3)$$
$$y_2 \leftarrow \phi(y_1, y_3)$$
$$y_2 == 0$$

```
print x₂
```

$$t_1 \leftarrow y_2$$
$$y_3 \leftarrow x_2 \% y_2$$
$$x_3 \leftarrow t_1$$

Evaluation of ɸ-functions

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

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

$$(x_2, y_2) \leftarrow \phi \begin{pmatrix} x_1 & y_1 \\ x_3 & y_3 \end{pmatrix} \quad \text{instead of} \quad \begin{array}{l} x_2 \leftarrow \phi(x_1, x_3) \\ y_2 \leftarrow \phi(y_1, y_3) \end{array}$$

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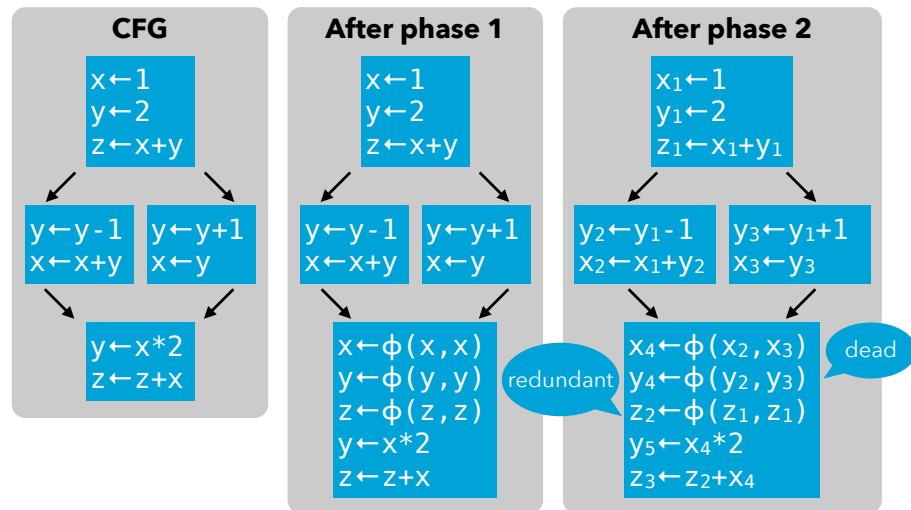

(Naïve) building of SSA form

Naïve technique to build SSA form:
  – for each variable x of the CFG, at each join point n, insert a **ɸ**-function of the form x=ɸ(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.

(Naïve) building of SSA form

CFG

```
x←1
y←2
z←x+y
```

```
y←y-1      y←y+1
x←x+y      x←y
```

```
y←x*2
z←z+x
```

After phase 1

```
x←1
y←2
z←x+y
```

```
y←y-1      y←y+1
x←x+y      x←y
```

```
x←φ(x,x)
y←φ(y,y)      redundant
z←φ(z,z)
y←x*2
z←z+x
```

After phase 2

```
x₁←1
y₁←2
z₁←x₁+y₁
```

```
y₂←y₁-1      y₃←y₁+1
x₂←x₁+y₂     x₃←y₃
```

```
x₄←φ(x₂,x₃)      dead
y₄←φ(y₂,y₃)
z₂←φ(z₁,z₁)
y₅←x₄*2
z₃←z₂+x₄
```

Better building techniques

The naïve 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 ɸ-functions – some dead, some redundant – to be useful in practice. It builds the **maximal** SSA form.
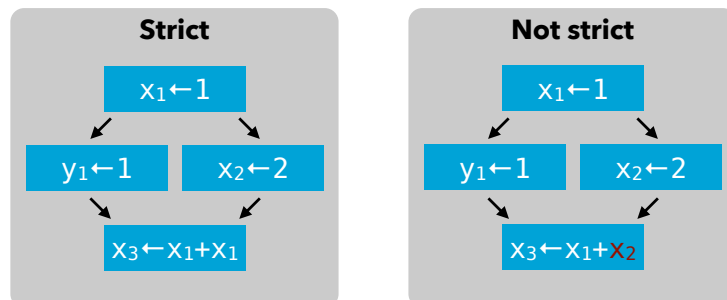
Better techniques exist to translate a program to SSA form.

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.

Strict

```
x₁←1
```

```
y₁←1      x₂←2
```

```
x₃←x₁+x₁
```

Not strict

```
x₁←1
```

```
y₁←1      x₂←2
```

```
x₃←x₁+x₂
```

Comparing IRs

CPS/L$_3$ vs RTL/CFG in SSA

As the correspondences in the table below illustrate, CPS/L$_3$ is very close to RTL/CFG in SSA form.

RTL/CFG in SSA	$\cong$	CPS/L$_3$
(named) basic block	$\cong$	continuation
ϕ-function	$\cong$	continuation argument
jump	$\cong$	continuation invocation
strict form	$\cong$	scoping rules

RTL/CFG in SSA vs CPS/L$_3$

RTL/CFG in SSA form

```
x₁←2016
y₁←714

loop

x₂←ϕ(x₁,y₂)
y₂←ϕ(y₁,y₃)
y₂ == 0

kt          kf

print x₂    y₃←x₂%y₂
```

CPS/L$_3$

```
(letₖ (
  (loop
    (cont (x₂ y₂)
      (let* ((kt (cont ()
                   (appf print x₂)))
             (kf (cont ()
                   (letp ((y₃ (% x₂ y₂)))
                     (appk loop y₂ y₃))))
             (z 0))
        (if (= y₂ z) kt kf)))))
  (let* ((x₁ 2016) (y₁ 714))
    (appk loop x₁ y₁)))
```

Summary and references

A bold claim: continuation-based, functional IRs like CPS/L$_3$ are SSA done right, and should replace it!

* * *

CPS/L$_3$ 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.