Intermediate representations

Advanced Compiler Construction
Michel Schinz – 2016-03-03
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.
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:

\[
\text{(rec loop ((x 2016) (y 714))}
\]

\[
\text{(if (= 0 y)}
\]

\[
\text{(int-print x)}
\]

\[
\text{(loop y (% x y)))}
\]
IR #1: CPS/L₃
CPS/L₃: 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/L₃ is the functional IR used by the L₃ compiler.
Local continuations

A crucial notion in CPS/L₃ 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.
Continuations are used for two purposes in CPS/L₃:

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/L₃ grammar

\[ T ::= (\text{let}_{l} ((N \ L)) \ T) \]
\[ | (\text{let}_{p} ((N (P \ N \ ...))) \ T) \]
\[ | (\text{let}_{c} ((N (\text{cnt} (N \ ...)) \ T))) \ T) \]
\[ | (\text{let}_{f} ((N (\text{fun} (N \ N \ ...)) \ T)) \ ...) \ T) \]
\[ | (\text{app}_{c} N \ N \ ...) \]
\[ | (\text{app}_{f} N \ N \ N \ ...) \]
\[ | (\text{if} (C \ N \ N) \ N \ N) \]
\[ | (\text{halt} N) \]

\[ N ::= \text{name} \]
\[ L ::= \text{integer, character, boolean or unit literal} \]
\[ P ::= + | - | \times | / | \% | ... \]
\[ C ::= < | <= | = | != | >= | > \]
CPS/L₃ local bindings

(let₁ ((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.

(letₚ ((n (p n₁ ...))) e)
Binds the name n to the result of the application of primitive p to the value of n₁, ... 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

(\texttt{let}_f ((f_1 (\texttt{fun} (c_1 n_{1,1} ...) b_1)) ...) e)
Binds the names \(f_1, \ldots\) to functions with arguments \(n_{1,1}, \ldots\) and return continuation \(c_1, \ldots\) 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{app}_f f c n_1 \ldots)
Applies the function bound to \(f\) to return continuation \(c\) and arguments \(n_1, \ldots\). The name \(c\) must either be bound by an enclosing \texttt{let}_c or be the name of the return continuation of the current function.
CPS/L₃ local continuations

\((\text{let}_c ((c_1 (\text{cnt} (n_{1,1} \ldots) b_1)) \ldots) e)\)
Binds the names \(c_1, \ldots\) to local continuations with arguments \(n_{1,1}, \ldots\) and body \(b_1, \ldots\) in expression \(e\).
Interpretation: like a local function that never returns.

\((\text{app}_c c n_1 \ldots)\)
Applies the continuation bound to \(c\) to the value of \(n_1, \ldots\)
The name \(c\) must either be bound by an enclosing \(\text{let}_c\) or be the name of the return continuation of the current function. Interpretation: if \(c\) designates a local continuation, \(\text{app}_c\) can be seen as a jump with arguments. If \(c\) designates the current return continuation, \(\text{app}_c\) can be seen as a return from the current function, with the given return value.
(if (p n₁ n₂) cₜ cₙ)
Tests whether the condition p is true for the value of n₁ and n₂, then applies continuation cₜ if it is, or cₙ if it isn't. Both cₜ and cₙ must be parameterless continuations.
The primitive p must be a logical primitive.
Note: if is a branching form of continuation application for parameterless continuations. It is therefore a conditional version of appc.
(halt n)
Halts program execution, exiting with the value bound to n (which must be an integer).
Continuation scopes

The scoping rules of CPS/L₃ 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:

```plaintext
(letᶜ ((c₀ (cnt (r) (appᶠ print r))))
  (letᶠ ((f (fun (c₁ x)
                (letᵖ ((t (+ x x))
                       (c₁ t))))))))
```

c₀ 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.
CPS/L₃ syntactic sugar

To make CPS/L₃ programs easier to read and write, we allow the collapsing of nested occurrences of \texttt{let₁}, \texttt{letₚ} and \texttt{letₖ} expressions to a single \texttt{let*} expression. We also allow the elision of \texttt{app₇} and \texttt{appₖ} in applications.

Example:

\[
\begin{align*}
\text{(let₁ ((c₁ 1))} \\
\text{ (let₁ ((c₂ 2))} \\
\text{ (letₚ ((c₃ (+ c₁ c₂))))} \\
\text{ ... (app₇ f c₃)))}
\end{align*}
\]

\[
\Rightarrow
\]

\[
\begin{align*}
\text{(let* ((c₁ 1))} \\
\text{ (c₂ 2)} \\
\text{ (c₃ (+ c₁ c₂))} \\
\text{ ... (f c₃))}
\end{align*}
\]
GCD in CPS/L₃

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

(\texttt{let}_c ((\texttt{loop}
\begin{align*}
& (\texttt{cnt} (x \ y)) \\
& (\texttt{let}* ((ct (cnt ()
\begin{align*}
& (app_f \ print \ x))) \\
& (cf (cnt ()
\begin{align*}
& (let_p ((t (\% \ x \ y))) \\
& (app_c \ loop \ y \ t)))))) \\
& (z 0)) \\
& (if (= y z) ct cf)))))) \\
& (\texttt{let}* ((x 2016) (y 714)) \\
& (app_c \ loop \ x \ y)))
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\begin{align*}
(To \ simplify, \ no \ return \ continuation \ is \ passed \ to \ print).
Translation of $\text{CL}_3$ to CPS/L$_3$
Translating CL₃ to CPS/L₃

The translation from CL₃ to CPS/L₃ is specified as a function denoted by $⟦·⟧$ and taking two arguments:

1. T, the CL₃ term to be translated,
2. C, the context, a CPS/L₃ 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:

$⟦T⟧C$

The translation function must return a CPS/L₃ term including both the translation of the CL₃ term T and the context C with its hole plugged by a name bound to the value of (the translation of) T.
The translation context is a CPS/L₃ term representing the partial translation of the CL₃ expression surrounding the one being translated. This term contains a single hole, written □, 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 (app_f □) plugged with name m results in the term (app_f m).
Translation rules often build contexts that include other contexts. One (fictional) example could be:

\[
\langle e_1 \rangle \langle \langle e_2 \rangle \ (\text{app} \ f \ \square) \rangle
\]

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

\[
\langle e_1 \rangle \lambda v_1 (\langle e_2 \rangle \lambda v_2 (\text{app} \ f \ v_1 v_2))
\]

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 L₃ project – the translation function \([\cdot]\) is defined as a function with the following profile:

```scala
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 named \(n\) is done using Scala function application:

```scala
c(n)
```
CL₃ to CPS/L₃ 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 = (let₁ ((n l)) C[n])

⟦(let ((n₁ e₁) (n₂ e₂) ...) e)⟧C =  
   [(let ((n₁ e₁)) (let ((n₂ e₂) ...) e))]C

⟦(let ((n₁ e₁)) e)⟧C =  
   [e₁](λv (letₚ ((n₁ (id v))) [e]C))

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

identity primitive (return its argument)
CL₃ to CPS/L₃ translation (2)

\[
[(\text{letrec} \ ((f_1 \ (\text{fun} \ (n_{1,1} \ n_{1,2} \ldots) \ e_1)) \ldots) \ e)) \ C = \\
(\text{let}_f \ ((f_1 \ (\text{fun} \ (c \ n_{1,1} \ n_{1,2} \ldots) \\
\quad[(e_1][\lambda v \ (\text{app}_c \ c \ v)])) \ldots) \\
[e] C) \\
[(e \ e_1 \ e_2 \ldots)] C = \\
[e]\!(\lambda v [e_1]\!(\lambda v_1 [e_2]\!(\lambda v_2 \ldots \\
(\text{let}_c \ ((c \ (\text{cnt} \ (r) \ C[r])))\)) \\
(\text{app}_f \ v \ c \ v_1 \ v_2 \ldots)))]
\]
CL₃ to CPS/L₃ translation (3)

\[ [(\text{if} \ (\@ p \ e₁ \ldots) \ e₂ \ e₃)]C \text{ where } p \text{ is a logical primitive} = \]
\[ (\text{let}_c \ ((c \ (\text{cnt} \ (r) \ C[r]))) \]
\[ \quad (\text{let}_c \ ((\text{ct} \ (\text{cnt} \ ()) [e₂](\lambda v₂ \ (\text{app}_C \ c \ v₂)))) \]
\[ \quad (\text{let}_c \ ((\text{cf} \ (\text{cnt} \ ()) [e₃](\lambda v₃ \ (\text{app}_C \ c \ v₃)))) \]
\[ \quad [e₁](\lambda v₁ \ldots (\text{if} \ (p \ v₁ \ldots) \ \text{ct} \ \text{cf}))) \]

\[ [(\text{if} \ e₁ \ e₂ \ e₃)]C = \]
\[ (\text{let}_c \ ((c \ (\text{cnt} \ (r) \ C[r]))) \]
\[ \quad (\text{let}_c \ ((\text{ct} \ (\text{cnt} \ ()) [e₂](\lambda v₂ \ (\text{app}_C \ c \ v₂)))) \]
\[ \quad (\text{let}_c \ ((\text{cf} \ (\text{cnt} \ ()) [e₃](\lambda v₃ \ (\text{app}_C \ c \ v₃)))) \]
\[ \quad (\text{let}_l \ ((\text{f} \ #f)) \]
\[ \quad \ [e₁](\lambda v₁ \ (\text{if} \ (!= \ v₁ \ f) \ \text{ct} \ \text{cf}))) \]
\[ \langle @p \ e_1 \ e_2 \ldots \rangle \] C where \( p \) is a logical primitive = 
\[ \langle \text{if} (p \ e_1 \ e_2 \ldots) \ #\text{t} \ #\text{f} \rangle \] C 
\[ \langle @p \ e_1 \ e_2 \ldots \rangle \] C where \( p \) is not a logical primitive = 
left as an exercise
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:

\[ \lambda v \ (\text{let}\_1 ((z \ 0)) \ (\text{halt} \ z)) \]

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.
Exercise

Translate the following $L_3$ expression:

$$(f \ 1 \ 2)$$

in the initial context, which we'll abbreviate as $\lambda v \ (\text{halt} \ 0)$. 
Better translation of $\text{CL}_3$ to $\text{CPS/L}_3$
Improving the translation

The translation presented before has two shortcomings:

1. it produces terms containing useless continuations, and
2. it produces suboptimal CPS/L\textsubscript{3} code for some conditionals.

One solution to improve the translation is to define several different translations depending on the source (i.e. L\textsubscript{3}) context in which the expression to translate appears.
Useless continuations

The first problem can be illustrated with the $L_3$ term:

\[
(\text{letrec } ((f \ (\text{fun } (g) \ (g)))) \ f)\]

which – in the empty context – gets translated to:

\[
(\text{let}_f \ ((f \ (\text{fun } (c \ g) \n
\quad \text{(let}_c \ ((j \ (\text{cnt } (r) \n
\quad \quad \text{(app}_c \ c \ r)))) \n
\quad \quad \text{(app}_f \ g \ j)))) \n
\quad f)\]

instead of the equivalent and more compact:

\[
(\text{let}_f \ ((f \ (\text{fun } (c \ g) \ (\text{app}_f \ g \ c)))) \n
\quad f)\]
Suboptimal conditionals (1)

The second problem can be illustrated with the L₃ term:

\[(\text{if} \ (\text{if} \ a \ b \ \#f) \ x \ y)\]

which, in the empty context, gets translated to:

\[
\begin{align*}
\text{(let* } & ((ci1 \ (\text{cnt} \ (v1) \ v1)) \\
& (ct1 \ (\text{cnt}() \ (\text{app}_c ci1 \ x))) \\
& (cf1 \ (\text{cnt}() \ (\text{app}_c ci1 \ y))) \\
& (f1 \ \#f) \\
& (ci2 \ (\text{cnt} \ (v2) \\
& \quad (\text{if} \ (\not= \ v2 \ f1) \ ct1 \ cf1))) \\
& (ct2 \ (\text{cnt}() \ (\text{app}_c ci2 \ b))) \\
& (cf2 \ (\text{cnt}() \\
& \quad \quad (\text{let} \ ((i1 \ \#f)) \ (\text{app}_c ci2 \ i1))) \\
& (f2 \ \#f)) \\
& (\text{if} \ (\not= \ a \ f2) \ ct2 \ cf2))
\end{align*}
\]
Suboptimal conditionals (2)

A much better translation for:

```
(if (if a b #f) x y)
```

would be:

```
(let* ((ci1 (cnt (v1) v1))
       (ct1 (cnt () (app c ci1 x)))
       (cf1 (cnt () (app c ci1 y)))
       (ca1 (cnt ()
            (let ((i1 #f))
                (if (!= b i1) ct1 cf1))))
       (i2 #f))
  (if (= a i2) ca1 cf1))
```

which immediately applies continuation cf1 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. $\langle \cdot \rangle_N C$, 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. $\langle \cdot \rangle_T c$, taking a term to translate and a one-parameter continuation $c$. This continuation is to be applied to the value of the term.

3. $\langle \cdot \rangle_C c_t c_f$, taking a term to translate and two parameterless continuations, $c_t$ and $c_f$. The continuation $c_t$ is to be applied when the term evaluates to a true value, while the continuation $c_f$ is to be applied when it evaluates to a false value.
The non-tail translation

⟦·⟧₍ 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:

\[
\text{⟦(@ p e₁ e₂ ...⟧)}₄ C \text{ where } p \text{ is not a logical primitive} = \]

\[
\text{⟦e₁⟧₄(λv₁ \text{⟦e₂⟧₄(λv₂...}}
\]

\[
(\text{let}_p ((\text{n (p v₁ v₂ ...)))})
\]

\[
C[\text{n}])))
\]
The **tail** translation $\lbrack \cdot \rbrack_T$ is used whenever the context passed to the simple translation has the form $\lambda v \ (\text{app}_c \ c \ v)$. 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:

$$\lbrack (\text{letrec} \ ((f_1 \ (\text{fun} \ (n_{1,1} \ n_{1,2} \ldots) \ e_1)) \ldots) \ e) \rbrack C = \ (\text{letf} \ ((f_1 \ (\text{fun} \ (c \ n_{1,1} \ n_{1,2} \ldots) \ \ [e_1] (\lambda v \ (\text{app}_c \ c \ v)))) \ldots) \ [e] C$$

becomes:

$$\lbrack (\text{letrec} \ ((f_1 \ (\text{fun} \ (n_{1,1} \ n_{1,2} \ldots) \ e_1)) \ldots) \ e) \rbrack_N C = \ (\text{letf} \ ((f_1 \ (\text{fun} \ (c \ n_{1,1} \ n_{1,2} \ldots) \ \ [e_1]_T \ c)) \ldots) \ [e]_N C$$
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:

$$⟦(if\ e_1\ e_2\ e_3)⟧_N\ C =$$

$$(let_c ((c (cnt (r) C[r])))$$

$$(let_c ((ct (cnt ()) ⟦e_2⟧_T\ c)))$$

$$(let_c ((cf (cnt ()) ⟦e_3⟧_T\ c)))$$

$$(⟦e_1⟧_C\ ct\ cf))))$$
The cond translation

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

\[
\begin{align*}
\[(\text{if } e_1 \ e_2 \ #f)\]_c &= (\text{let}_c \ ((\text{ac} \ (\text{cnt} () [e_2]_c \ c_t \ c_f)))) \\
\quad [e_1]_c \ a c \ c_f \\
\[(\text{if } e_1 \ #f \ #t)\]_c &= [e_1]_c \ c_f \ c_t \\
\end{align*}
\]

...and so on for all conditionals with at least one constant branch.
The better 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, c: Symbol): CPSTree

def cond(t: CL3Tree, ct: Symbol, cf: Symbol): CPSTree
```
IR #2:
standard RTL/CFG
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.
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 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.
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.
The same examples as before, but with basic blocks instead of individual instructions.

```
x \leftarrow 2016
y \leftarrow 714
```

```
y == 0
```

```
\text{print } x
```

```
t \leftarrow y
y \leftarrow x \% y
x \leftarrow t
```
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.
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:

\[
\begin{align*}
    x & \leftarrow 12 \\
    y & \leftarrow 15 \\
    x & \leftarrow x + y \\
    y & \leftarrow x + 4 \\
    z & \leftarrow x + y \\
    y & \leftarrow y + 1
\end{align*}
\]

\[
\begin{align*}
    x_1 & \leftarrow 12 \\
    y_1 & \leftarrow 15 \\
    x_2 & \leftarrow x_1 + y_1 \\
    y_2 & \leftarrow x_2 + 4 \\
    z_1 & \leftarrow x_2 + y_2 \\
    y_3 & \leftarrow y_2 + 1
\end{align*}
\]
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 $\phi$-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.
\( \text{\(\phi\)-functions example} \)

**not in SSA form**

\[
\begin{align*}
x & \leftarrow 2016 \\
y & \leftarrow 714 \\
y & \equiv 0 \\
\text{print } x \\
t & \leftarrow y \\
y & \leftarrow x \% y \\
x & \leftarrow t
\end{align*}
\]

**in SSA form**

\[
\begin{align*}
x_1 & \leftarrow 2016 \\
y_1 & \leftarrow 714 \\
x_2 & \leftarrow \phi(x_1, x_3) \\
y_2 & \leftarrow \phi(y_1, y_3) \\
y_2 & \equiv 0 \\
\text{print } x_2 \\
t_1 & \leftarrow y_2 \\
y_3 & \leftarrow x_2 \% y_2 \\
x_3 & \leftarrow t_1
\end{align*}
\]

All \( \phi \)-functions are evaluated in parallel.
Evaluation of $\phi$-functions

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

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

\[
(x_2, y_2) \leftarrow \phi \begin{pmatrix} x_1 & y_1 \\ x_3 & y_3 \end{pmatrix}
\]

instead of

\[
x_2 \leftarrow \phi(x_1, x_3) \\
y_2 \leftarrow \phi(y_1, y_3)
\]

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

The following loop extract illustrates why $\phi$-functions must be evaluated in parallel.

**not SSA**

\[
\begin{align*}
&\text{x } \leftarrow \ldots \\
&\text{y } \leftarrow \ldots \\
&\text{y } \leftarrow \text{0}
\end{align*}
\]

**SSA**

\[
\begin{align*}
&\text{x}_1 \leftarrow \ldots \\
&\text{y}_1 \leftarrow \ldots \\
&\text{x}_2 \leftarrow \phi(\text{x}_1, \text{x}_3) \\
&\text{y}_2 \leftarrow \phi(\text{y}_1, \text{y}_3) \\
&\text{y}_2 \leftarrow \text{0}
\end{align*}
\]

**optimized SSA**

\[
\begin{align*}
&\text{x}_1 \leftarrow \ldots \\
&\text{y}_1 \leftarrow \ldots \\
&\text{x}_2 \leftarrow \phi(\text{x}_1, \text{y}_2) \\
&\text{y}_2 \leftarrow \phi(\text{y}_1, \text{x}_2) \\
&\text{y}_2 \leftarrow \text{0}
\end{align*}
\]
(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 \( \phi \)-function of the form \( x = \phi(x, \ldots, x) \) 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

After phase 1

After phase 2

$\phi(x,x)$

$\phi(y,y)$

$\phi(z,z)$
(Naïve) building of SSA form

CFG

\[
\begin{align*}
x & \leftarrow 1 \\
y & \leftarrow 2 \\
z & \leftarrow x + y \\
y & \leftarrow y - 1 \\
x & \leftarrow x + y \\
y & \leftarrow x \ast 2 \\
z & \leftarrow z + x \\
\end{align*}
\]

After phase 1

\[
\begin{align*}
x & \leftarrow 1 \\
y & \leftarrow 2 \\
z & \leftarrow x + y \\
y & \leftarrow y - 1 \\
x & \leftarrow x + y \\
y & \leftarrow y + 1 \\
x & \leftarrow y \\
y & \leftarrow x \ast 2 \\
z & \leftarrow z + x \\
\end{align*}
\]

After phase 2

\[
\begin{align*}
x & \leftarrow 1 \\
y & \leftarrow 2 \\
z & \leftarrow x + y \\
y & \leftarrow y - 1 \\
x & \leftarrow x + y \\
y & \leftarrow y + 1 \\
x & \leftarrow y \\
y & \leftarrow x \ast 2 \\
z & \leftarrow z + x \\
x_1 & \leftarrow 1 \\
y_1 & \leftarrow 2 \\
z_1 & \leftarrow x_1 + y_1 \\
y_2 & \leftarrow y_1 - 1 \\
x_2 & \leftarrow x_1 + y_2 \\
y_3 & \leftarrow y_1 + 1 \\
x_3 & \leftarrow y_3 \\
x_4 & \leftarrow \phi(x_2, x_3) \\
y_4 & \leftarrow \phi(y_2, y_3) \\
z_2 & \leftarrow \phi(z_1, z_1) \\
y_5 & \leftarrow x_4 \ast 2 \\
z_3 & \leftarrow z_2 + x_4 \\
dead
\end{align*}
\]
(Naïve) building of SSA form

**CFG**

- $x ← 1$
- $y ← 2$
- $z ← x + y$

- $y ← y - 1$
- $x ← x + y$
- $y ← y + 1$
- $x ← y$

- $y ← x * 2$
- $z ← z + x$

**After phase 1**

- $x ← 1$
- $y ← 2$
- $z ← x + y$

- $y ← y - 1$
- $x ← x + y$
- $y ← y + 1$
- $x ← y$

- $y ← x * 2$
- $z ← z + x$

**After phase 2**

- $x_1 ← 1$
- $y_1 ← 2$
- $z_1 ← x_1 + y_1$

- $y_2 ← y_1 - 1$
- $x_2 ← x_1 + y_2$
- $y_3 ← y_1 + 1$
- $x_3 ← y_3$

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

dead

redundant
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 $\phi$-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.
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**

- $x_1 \leftarrow 1$
- $y_1 \leftarrow 1$
- $x_2 \leftarrow 2$
- $x_3 \leftarrow x_1 + x_1$

**Non strict**

- $x_1 \leftarrow 1$
- $y_1 \leftarrow 1$
- $x_2 \leftarrow 2$
- $x_3 \leftarrow x_1 + x_2$
Comparing IRs
CPS/L₃ vs RTL/CFG in SSA

As the correspondences in the table below illustrate, CPS/L₃ is very close to RTL/CFG in SSA form.

<table>
<thead>
<tr>
<th>RTL/CFG in SSA</th>
<th>≈</th>
<th>CPS/L₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>(named) basic block</td>
<td>≈</td>
<td>continuation</td>
</tr>
<tr>
<td>(\phi)-function</td>
<td>≈</td>
<td>continuation argument</td>
</tr>
<tr>
<td>jump</td>
<td>≈</td>
<td>continuation invocation</td>
</tr>
<tr>
<td>strict form</td>
<td>≈</td>
<td>scoping rules</td>
</tr>
</tbody>
</table>
CPS/L$_3$ vs RTL/CFG in SSA

**RTL/CFG in SSA form**

- $x_1 \leftarrow 2016$
- $y_1 \leftarrow 714$
- $x_2 \leftarrow \phi(x_1, y_2)$
- $y_2 \leftarrow \phi(y_1, y_3)$
- $y_2 \equiv 0$

**CPS/L$_3$**

```
(letc (loop (cnt (x_2 y_2) (let* ((ct (cnt (appf print x_2)))
                                      (cf (cnt (letp ((y_3 (% x_2 y_2)))
                                                (appc loop y_2 y_3))))
                                      (z 0))
                                    (if (= y_2 z) ct cf))))
  (let* ((x_1 2016) (y_1 714)) (appc loop x_1 y_1))
```
Summary and references

Claim: continuation-based, functional IRs like CPS/L₃ are SSA done right, and should replace it – or, at the very least, \( \Phi \)-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/L₃ 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.