Intermediate representations in a compiler

Between high-level languages and machine code, compilers generally go through one or several intermediate representations where, in particular:

- Expressions are decomposed in a sequence of processor-level instructions.

\[
\begin{align*}
x &= (y + z) \times (a - b) \\
\text{--->} \\
t1 &= y + z; \quad t2 = a - b; \quad x = t1 \times t2;
\end{align*}
\]

- Temporary variables \((t1, t2)\) are introduced to hold intermediate results.
- These temporaries, along with program variables, can later be placed in concrete locations: processor registers or stack slots.
- Various optimizations can be performed over the intermediate representation.
A conventional IR: RTL-CFG

A conventional intermediate representation: RTL-CFG

(Register Transfer Language with Control-Flow Graph.)

A function = a set of processor-level instructions operating over variables and temporaries, e.g.

\[
\begin{align*}
    x &= y + z \\
    t &= \text{load}(x + 8) \\
    \text{if } (t == 0)
\end{align*}
\]

Organized in a control-flow graph:
- Nodes = instructions.
- Edge from \( I \) to \( J \) = \( J \) can execute just after \( I \).
Example: some source code

```c
double average(int * tbl, int size)
{
    double s = 0.0;
    int i;
    for (i = 0; i < size; i++)
        s = s + tbl[i];
    return s;
}
```

Example: the corresponding RTL graph
Classic optimizations over RTL

Many classic optimizations can be performed on the RTL form.

- **Constant propagation**
  
  ```
  a = 1
  b = 2
  c = a + b
  d = x - a
  --->
  a = 1
  b = 2
  c = 3
  d = x + (-1)
  ```

- **Dead code elimination**
  
  ```
  a = 1
  b = 2
  c = 3
  --->
  a = 1
  b = 2
  c = 3
  ```

  (if a unused later)

- **Common subexpression elimination**
  
  ```
  c = a
  d = a + b
  e = c + b
  --->
  c = a
  d = a + b
  e = d
  ```

- **Hoisting of loop-invariant computations**
  
  ```
  L: c = a + b
  ... -->
  L: ...
  ...
  --> L
  ... --> L
  ```

- **Induction variable elimination**
  
  ```
  i = 0
  L: a = i * 4
  b = p + a
  ... -->
  b = p
  L: ...
  ...
  b = b + 4
  i = i + 1 -> L
  ```

  ... and much more. (See e.g. Steven Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers.)
RTL optimizations and dataflow analysis

Problem: it is not obvious to see where these optimizations apply, because

1. A given variable or temporary can be defined several times. (Unavoidable if the source language is imperative.)
2. The CFG is not a structured representation of control.

Solution: use static analyses to determine opportunities for optimization, e.g. dataflow analyses (a simple case of abstract interpretation).

Example: for constant propagation, use the abstract lattice

\[ a := T \mid \bot \mid N \]

\[ \tilde{n} = 1 \]
\[ \tilde{n} = \top \]
\[ \tilde{n} = 2 \]
\[ n = n + 1 \]
Single Static Assignment (SSA)

SSA is a variant of RTL where every variable is the result of only one instruction.

For straight-line codes: just rename variables to avoid accidental reuse.

```
Not SSA
y = x * 4
y = y + 1
if (y > 0)

SSA
y = x * 4
y' = y + 1
if (y' > 0)
```

Phi nodes

For join points and loops in the control-flow graph: introduce φ instructions to merge multiple definitions of a variable.

```
Not SSA
x = ...
use x

SSA
x1 = ...
x2 = ...
x3 = φ(x1,x2)
use x3
```

Informal semantics: \( x = \phi(x_1, \ldots, x_n) \) assigns \( x = x_i \) when entered from the \( i \)-th predecessor node.
Why SSA?

SSA simplifies dataflow analysis: a given variable has the same abstract value at every point where the variable is defined.

\[
\begin{align*}
n1 &= 1 \\
n2 &= 2 \\
n3 &= \phi(n1, n2) \\
n4 &= \phi(n2, n5) \\
n5 &= n4 + 1
\end{align*}
\]

We have \(\tilde{n}1 = 1\) and \(\tilde{n}2 = 2\) and \(\tilde{n}3 = \tilde{n}4 = \tilde{n}5 = \top\) everywhere.

Outline

1. A conventional IR: RTL-CFG
2. CPS as a functional IR
3. Another functional IR: A-normal forms
CPS as a functional IR

CPS terms share many features of intermediate representations. In particular, expressions are decomposed in individual operations and intermediate results are named.

Example: source term `let x = (y + z) * (a - b) in ....`

\[
\begin{align*}
\text{CPS} & \quad \text{RTL} \\
(y + z) & \Rightarrow (\lambda t. \ t = y + z; \\
(a - b) & \Rightarrow (\lambda u. \ u = a - b; \\
(t * u) & \Rightarrow (\lambda x. \ x = t * u; \\
\ldots)) & \ldots
\end{align*}
\]

(We write $\Rightarrow$ for reverse function application: $a \Rightarrow b = b a$.)

Likewise, `let`-bound continuations correspond to join points in a control-flow graph. Applying such a continuation corresponds to a $\phi$ node in SSA form.

Example: source term `let x = (if c then y + 1 else z + 2) in ...`

\[
\begin{align*}
\text{let } k = \lambda x \ldots \text{ in} \\
\text{if } c \\
\text{then } k(y+1) \\
\text{else } k(z+2)
\end{align*}
\]
Optimizations on CPS terms

When expressed over CPS terms, many classic optimizations boil down to $\beta$-reduction, or arithmetic reductions, or variants thereof.

Example: constant propagation $\approx \beta$ and arithmetic reduction.

$$1 \gg (\lambda x. \ldots x + 1 \ldots x + y \ldots)$$
$$\rightarrow \ldots 2 \ldots 1 + y \ldots$$

Example: common subexpression elimination $\approx$ inverse $\beta$

$$(a + b) \gg (\lambda x. \ldots \rightarrow (a + b) \gg (\lambda y. \ldots \rightarrow (a + b) \gg (\lambda y. \ldots$$

Back to direct style

To support stack-allocation of activation records, several functional compilers perform an inverse CPS transformation after CPS optimization, to recover direct-style (DS) function calls.
The inverse CPS transformation
(Sabry and Felleisen, LFP 1992)

Grammar of CPS terms after administrative normalization:

Terms: \( P ::= k \ W \ | \ \text{let } x = W \ \text{in } P \ | \ W_1 \ W_2 \ k \ | \ W_1 \ W_2 \ (\lambda k. P) \)

Values: \( W ::= N \ | \ x \ | \ \lambda x. \lambda k. P \)

Inverse CPS transformation:

\[
U(k \ W) = \bar{W}
\]
\[
U(\text{let } x = W \ \text{in } P) = \text{let } x = \bar{W} \ \text{in } U(P)
\]
\[
U(W_1 \ W_2 \ k) = \bar{W_1} \ W_2 \ \text{(tail application)}
\]
\[
U(W_1 \ W_2 \ (\lambda x. P)) = \text{let } x = W_1 \ W_2 \ \text{in } P \ \text{(non-tail application)}
\]

with \( \bar{N} = N \) and \( \bar{x} = x \) and \( \bar{\lambda x. \lambda k. P} = \lambda x. U(P) \)

The origin of ANF
(Flanagan, Sabry, Felleisen, *The essence of compiling with continuations*, PLDI 1993.)

In 1993, Flanagan, Sabry and Felleisen showed that this detour through CPS can be avoided, and indeed is unnecessary in the following formal sense:

ANF stands for “administrative normal form”, and is the direct-style sub-language that is the target of inv-CPS-transf \( \circ \) adm-red \( \circ \) CPS-transf.
Outline

1 A conventional IR: RTL-CFG

2 CPS as a functional IR

3 Another functional IR: A-normal forms

Syntax of ANF

Atom:

\[ a ::= x \mid N \mid \lambda \vec{x}.b \]

Computation:

\[ c ::= a_1 \text{ op } a_2 \quad \text{arithmetic} \]

\[ \mid a(\vec{a}) \quad \text{function application} \]

\[ \mid C(\vec{a}) \quad \text{datatype constructor} \]

\[ \mid \text{closure}(a, \vec{a}) \quad \text{closure constructor} \]

Body:

\[ b ::= c \quad \text{tail computation} \]

\[ \mid \text{let } x = c \text{ in } b \quad \text{sequencing} \]

\[ \mid \text{if } a \text{ then } b_1 \text{ else } b_2 \quad \text{conditional} \]

\[ \mid \text{match } a \text{ with } \ldots p_i \rightarrow b_i \ldots \quad \text{pattern-matching} \]
Another functional IR: A-normal forms

ANF as a CFG

let x = a + b in
if (x >= a)
then x
else 0

Conversion to ANF

Step 1: perform monadic conversion.

Example 1
Source term: 1 + (if x >= 0 then f(x) else 0)

Monadic conversion:
bind (if x >= 0 then f(x) else ret 0)
(λt. 1 + t)
Conversion to ANF

Step 2: interpret the result in the Identity monad:

\[
\begin{align*}
\text{ret } a & \mapsto a \\
\text{bind } a \ (\lambda x. b) & \mapsto \text{let } x = a \ \text{in } b
\end{align*}
\]

Example 2

Source term: \(1 + (\text{if } x \geq 0 \ \text{then } f(x) \ \text{else } 0)\)

Monadic conversion + identity monad:

\[
\begin{align*}
\text{let } t &= \text{if } x \geq 0 \ \text{then } f(x) \ \text{else } \text{ret } 0 \\
\text{in } 1 + t
\end{align*}
\]

Result is in so-called Monadic Intermediate Form, but not yet in ANF.

Conversion to ANF

Step 3: “flatten” the nesting of `let`, `if` and `match`.

\[
\begin{align*}
\text{let } x &= (\text{let } y = a \ \text{in } b) \ \text{in } c \\
\quad &\to \ \text{let } y = a \ \text{in } \text{let } x = b \ \text{in } c \quad (\text{if } y \ \text{not free in } c) \\
\text{let } x &= (\text{match } a \ \text{with } \ldots p_i \to b_i \ldots) \ \text{in } c \\
\quad &\to \ \text{match } a \ \text{with } \ldots p_i \to \text{let } x = b_i \ \text{in } c \ldots \\
\text{match } (\text{match } a \ \text{with } \ldots p_i \to b_i \ldots) \ \text{with } \ldots q_j \to c_j \ldots \\
\quad &\to \ \text{match } a \ \text{with } \ldots p_i \to (\text{match } b_i \ \text{with } \ldots q_j \to c_j \ldots) \ldots
\end{align*}
\]

Example 3

\[
\begin{align*}
\text{if } x \geq 0 \\
\text{then let } t &= f(x) \ \text{in } 1 + t \\
\text{else let } t &= 0 \ \text{in } 1 + t
\end{align*}
\]
Tail duplication, and how to avoid it

Note that possibly large terms can be duplicated:

$$\text{if (if } a \text{ then } b \text{ else } c \text{) then } d \text{ else } e$$

$$\rightarrow \text{if } a \text{ then (if } b \text{ then } d \text{ else } e \text{) else (if } c \text{ then } d \text{ else } e \text{)}$$

This can be avoided by using auxiliary functions: (≈ SSA $\phi$ nodes)

$$\text{if (if } a \text{ then } b \text{ else } c \text{) then } d \text{ else } e$$

$$\rightarrow \text{let } f(x) = \text{if } x \text{ then } d \text{ else } e \text{ in if } a \text{ then } f(b) \text{ else } f(c)$$

Optimizations on ANF terms

As in the case of CPS, classic optimizations boil down to $\beta$-reduction or arithmetic reductions over ANF terms.

Example: constant propagation $\approx \beta$ and arithmetic reduction.

$$\text{let } x = 1 \text{ in } \ldots \ x + 1 \ldots \ x + y \ldots$$

$$\rightarrow \ldots \ 2 \ldots \ 1 + y \ldots$$

Example: common subexpression elimination $\approx$ inverse $\beta$

$$\text{let } x = a + b \text{ in } \ldots$$

$$\text{let } y = a + b \text{ in } \ldots$$

$$\rightarrow \ldots$$

$$\text{let } y = x \text{ in } \ldots$$
Register allocation

The register allocation problem: place every variable in hardware registers or stack locations, maximizing the use of hardware registers.

Naive approach:
Assign the $N$ hardware registers to the $N$ most used variables; assign stack slots to the other variables.

Finer approach:
Notice that the same hardware register can be assigned to several distinct variables, provided they are never used simultaneously.

Example 4

if ... then (let x = ... in ...) else (let y = ... in ...)

x and y can share a register.

Register allocation on ANF

On functional intermediate representations like ANF, register allocation boils down to $\alpha$-conversion.

The register allocation problem, revisited: rename variables, using hardware registers or stack locations as new names, in such a way that

- (Correctness) the renamed term is $\alpha$-equivalent to the original;
- (Efficiency) hardware registers are used as much as possible.

Example 5

if ... then (let x = ... in ...) else (let y = ... in ...)

can be $\alpha$-converted to

if ... then (let R1 = ... in ...) else (let R1 = ... in ...)

The interference graph

An undirected graph,
- Nodes: names of variables
- Edges: between any two variables that cannot be renamed to the same location, as this would violate $\alpha$-equivalence.

Constructing the interference graph: at each point where a variable $x$ is bound, add edges with all other variables that occur free in the continuation of this binding.

```plaintext
let x = c in b
→ add edges between $x$ and all $y \in FV(b) \setminus \{x\}$

match a with ... $C(x_1, \ldots, x_n) \rightarrow b$ ...
→ add edges between $x_i$ and all $y \in FV(b) \setminus \{x_i\}$.
```

Example of an interference graph

```plaintext
let s = 0.0 in
let i = 0 in
let rec f(s,i) =
  if (ri < size) then
    let a = i*4 in
    let b = load(tbl+a) in
    let c = float(b) in
    let s = s +f c in
    let i = i + 1 in
    f(s,i)
  else
    let d = float(size) in
    s /f d
in f(s,i)
```
Register allocation by graph coloring

Correct register allocations correspond to colorings of the interference graph: each node should be assigned a color (= a register or stack location) so that adjacent nodes have different colors.

If the interference graph can be colored with at most $N$ colors (where $N$ is the number of hardware register), we obtain a perfect register allocation.

Otherwise, the coloring is a good starting point to determine which variables go into registers.


Any undirected graph is the interference graph of a CFG $\rightarrow$ perfect register allocation on RTL-CFG is NP-complete.

The interference graphs for SSA graphs and ANF terms are chordal $\rightarrow$ “perfect” register allocation in polynomial time.

(F. Pereira and J. Palsberg, *Register allocation via coloring of chordal graphs*, APLAS 2005.)
Register allocation by graph coloring

Why “perfect” and not just perfect? Two auxiliary problems remain hard (as in NP-hard) even on chordal graphs:

- **Spilling:** when not enough registers, choose which variables to “spill” to stack slots, and insert appropriate stack-reg moves around uses of these variables.

- **Coalescing:** try to give the same color to two variables when doing so suppresses a move instruction, e.g.

  ```latex
  \text{let } f \ x = \ldots \text{ in } \ldots f \ y \ldots f \ z \ldots
  ```

  (Optimal: assign the same register to \(x\), \(y\) and \(z\)).

Some uses of functional intermediate representations

Production-quality compilers:

- Gambit (for Scheme, using CPS)
- Standard ML of New Jersey (for SML, using CPS)
- MLton (for SML, using CPS)
- MLj and SML.net (for SML, using Monadic Intermediate Form)

Simple formally-verified compilers for mini-ML:

- by A. Chlipala (using CPS)
- by Z. Dargaye (using Monadic Intermediate Form)

Functional intermediate representations as path of least resistance towards formally-verified functional compilers?