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.
  \[ x = (y + z) \times (a - b) \]
  \[ \rightarrow \]
  \[ t1 = y + z; \ t2 = a - b; \ x = t1 \times t2; \]

- 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.
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

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**
  
  \[
  \begin{align*}
  a &= 1 \\
  b &= 2 \\
  c &= a + b \\
  d &= x - a \\
  \end{align*}
  \]
  
  \[
  \begin{align*}
  a &= 1 \\
  b &= 2 \\
  c &= 3 \\
  d &= x + (-1) \\
  \end{align*}
  \]

- **Dead code elimination**
  
  \[
  \begin{align*}
  a &= 1 \\
  b &= 2 \\
  c &= 3 \\
  \end{align*}
  \]
  
  \[
  \begin{align*}
  &\text{nop} \\
  b &= 2 \\
  c &= 3 \\
  \end{align*}
  \]
  
  (if \(a\) unused later)

- **Common subexpression elimination**
  
  \[
  \begin{align*}
  c &= a \\
  d &= a + b \\
  e &= c + b \\
  \end{align*}
  \]
  
  \[
  \begin{align*}
  c &= a \\
  d &= a + b \\
  e &= d \\
  \end{align*}
  \]

- **Hoisting of loop-invariant computations**
  
  \[
  \begin{align*}
  L: &\quad c = a + b \\
  &\quad \ldots \\
  &\quad \ldots -> L \\
  \end{align*}
  \]
  
  \[
  \begin{align*}
  L: &\quad c = a + b \\
  &\quad \ldots \\
  &\quad \ldots -> L \\
  \end{align*}
  \]

- **Induction variable elimination**
  
  \[
  \begin{align*}
  i &= 0 \\
  L: &\quad a = i * 4 \\
  &\quad b = p + a \\
  &\quad \ldots \\
  &\quad i = i + 1 -> L \\
  \end{align*}
  \]
  
  \[
  \begin{align*}
  i &= 0 \\
  L: &\quad b = p \\
  &\quad \ldots \\
  &\quad b = b + 4 \\
  &\quad i = i + 1 -> L \\
  \end{align*}
  \]

- \ldots 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

- A given variable or temporary can be defined several times. (Unavoidable if the source language is imperative.)
- 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

\[
a := \top | \bot | N
\]

\[
n = 1 \quad n = 2
\]

\[
n \leftarrow 1 \quad n = n + 1
\]

\[\tilde{n} = 1 \quad \tilde{n} = \top \]

\[\tilde{n} = 2 \quad n = n + 1\]
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 \( \text{let } x = (y + z) \times (a - b) \text{ in } \ldots \)

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

(We write $ for reverse function application: \( a \$ b = b \ a \).)
CPS as a functional IR

Likewise, let-bound continuations correspond to join points in a control-flow graph.

Example: source term \( \text{let } x = (\text{if } c \text{ then } y \text{ else } z) \text{ in } \ldots \)

\[
\begin{align*}
\text{let } k &= \lambda x \ldots \text{ in} \\
\text{if } c \\
\text{then } k y \\
\text{else } k z
\end{align*}
\]

Optimizations on CPS terms

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

Example: constant propagation \( \approx \beta \), arithmetic reduction.

\[
1 \,(\lambda x. \ldots x + 1 \ldots x + y \ldots) \\
\rightarrow \ldots 2 \ldots 1 + y \ldots
\]

Example: common subexpression elimination \( \approx \) inverse \( \beta \)

\[
(\text{a + b) } \,(\lambda x. \ldots \text{a + b) } \,(\lambda x. \ldots \longrightarrow \ldots \\
\ldots \ldots \ldots \ldots \ldots \\
\ldots \ldots \ldots \ldots \ldots
\]

\[
(\text{a + b) } \,(\lambda y. \ldots \,x \,(\lambda y. \ldots \,\ldots \ldots \ldots \ldots
\]

\[
\ldots \ldots \ldots \ldots \ldots
\]
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 function calls.

The origin of ANF

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.

(C. Flanagan, A. Sabry, M. Felleisen, *The essence of compiling with continuations*, PLDI 1993.)
Outline

1. A conventional IR: RTL-CFG

2. CPS as a functional IR

3. Another functional IR: A-normal forms

Syntax of ANF

Atom:
\[ a ::= x \mid N \mid \lambda x.b \]

Computation:
\[ c ::= a_1 \text{ op } a_2 \]
\[ \mid a(\vec{a}) \]
\[ \mid C(\vec{a}) \]
\[ \mid \text{closure}(a, \vec{a}) \]

Body:
\[ b ::= c \]
\[ \mid \text{let } x = c \text{ in } b \]
\[ \mid \text{if } a \text{ then } b_1 \text{ else } b_2 \]
\[ \mid \text{match } a \text{ with } \ldots p_i \rightarrow b_i \ldots \]
Another functional IR: A-normal forms

\[ \text{let } x = a + b \text{ in} \]
\[ \text{if } (x \geq a) \]
\[ \text{then } x \]
\[ \text{else } 0 \]

Conversion to ANF

Step 1: perform monadic conversion.

**Example 1**

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

Monadic conversion:

\[ \text{bind } (\text{if } x \geq 0 \text{ then } f(x) \text{ else } \text{ret } 0) \]
\[ (\lambda t. 1 + t) \]
Conversion to ANF

Step 2: interpret the result in the Identity monad:

\[ \text{ret } a \mapsto a \]
\[ \text{bind } a (\lambda x. b) \mapsto \text{let } x = a \text{ in } b \]

Example 2

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

Monadic conversion + identity monad:

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

Conversion to ANF

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

\[ \text{let } x = (\text{let } y = a \text{ in } b) \text{ in } c \]
\[ \quad \rightarrow \text{ let } y = a \text{ in } \text{let } x = b \text{ in } c \quad \text{(if } y \text{ not free in } c) \]
\[ \text{let } x = (\text{match } a \text{ with } \ldots p_i \rightarrow b_i \ldots) \text{ in } c \]
\[ \quad \rightarrow \text{ match } a \text{ with } \ldots p_i \rightarrow \text{let } x = b_i \text{ in } c \ldots \]
\[ \text{match } (\text{match } a \text{ with } \ldots p_i \rightarrow b_i \ldots) \text{ with } \ldots q_j \rightarrow c_j \ldots \]
\[ \quad \rightarrow \text{ match } a \text{ with } \ldots p_i \rightarrow (\text{match } b_i \text{ with } \ldots q_j \rightarrow c_j \ldots) \ldots \]

Example 3

\[ \text{if } x \geq 0 \]
\[ \text{then let } t = f(x) \text{ in } 1 + t \]
\[ \text{else let } t = 0 \text{ in } 1 + t \]
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:

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

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

Optimizations on ANF terms

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

Example: constant propagation \(\approx\) \(\beta\), 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 \(\beta\)

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

\[\rightarrow \text{ let } y = a + b \text{ in } \ldots \text{ let } y = x \text{ in } \ldots\]
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.

\[
\text{let } x = c \text{ in } b \\
\quad \rightarrow \text{ add edges between } x \text{ and all } y \in FV(b) \setminus \{x\}
\]

\[
\text{match } a \text{ with } \ldots \ C(x_1, \ldots, x_n) \rightarrow b \ldots \\
\quad \rightarrow \text{ add edges between } x_i \text{ and all } y \in FV(b) \setminus \{x_i\}.
\]

Example of an interference graph

\[
\text{let } s = 0.0 \text{ in} \\
\text{let } i = 0 \text{ in} \\
\text{let rec } f(s,i) = \\
\quad \text{if } (ri < \text{size}) \text{ then} \\
\quad \quad \text{let } a = i*4 \text{ in} \\
\quad \quad \text{let } b = \text{load(tbl+a)} \text{ in} \\
\quad \quad \text{let } c = \text{float}(b) \text{ in} \\
\quad \quad \text{let } s = s +f c \text{ in} \\
\quad \quad \text{let } i = i + 1 \text{ in} \\
\quad \quad f(s,i) \\
\quad \text{else} \\
\quad \quad \text{let } d = \text{float}(\text{size}) \text{ in} \\
\quad \quad s /f d \\
\quad \text{in } f(s,i)
\]
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.