Functional programming languages
Part V: functional intermediate representations

Xavier Leroy

INRIA Paris-Rocquencourt

MPRI 2-4, 2015–2016
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.
- Various optimizations can be performed over the intermediate representation.
Outline

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

- \( s = 0.0 \)
- \( i = 0 \)
- \( \text{if } (i \geq \text{size}) \)
  - \( d = \text{float(size)} \)
  - \( e = s / d \)
  - \( \text{return(e)} \)
  - \( a = i \ll 2 \)
  - \( b = \text{load(tbl + a)} \)
  - \( c = \text{float(b)} \)
  - \( s = s + f c \)
  - \( i = i + 1 \)
  - \( e = s / d \)
  - \( \text{return(e)} \)
Classic optimizations over RTL

Many classic optimizations can be performed on the RTL form.

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

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

\[
\begin{align*}
  L: c &= a + b \\
  \ldots &\rightarrow L: \ldots \\
  \ldots &\rightarrow L
\end{align*}
\]

- Induction variable elimination

\[
\begin{align*}
  i &= 0 \\
  L: a &= i \times 4 &\rightarrow b &= p \\
  b &= p + a \\
  \ldots &\rightarrow L \\
  i &= i + 1 \rightarrow L
\end{align*}
\]

\[
\begin{align*}
  i &= 0 \\
  L: b &= p \\
  \ldots &\rightarrow L \\
  b &= b + 4 \\
  i &= i + 1 \rightarrow L
\end{align*}
\]

- ... 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.

\[
\begin{align*}
  n &= 1 \\
  n &= 2 \\
  n &= n + 1
\end{align*}
\]
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.

```
n = 1
```

```
n = 2
```

```
n = n + 1
```

$n$ is statically unknown

$n$ is 1

$n$ is 2

X. Leroy (INRIA)
RTL optimizations and dataflow analysis

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

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

Diagram:

\[ \tilde{n} = 1 \]

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

\[ \tilde{n} = 2 \]

\[ n = n + 1 \]

\[ n = 1 \]

\[ n = 2 \]
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**

\[
\begin{align*}
y &= x \times 4 \\
y &= y + 1 \\
\text{if } (y > 0)
\end{align*}
\]

**SSA**

\[
\begin{align*}
y &= x \times 4 \\
y' &= y + 1 \\
\text{if } (y' > 0)
\end{align*}
\]
Phi nodes

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

**Not SSA**

\[
x = \ldots \quad \text{use } x
\]

\[
x = \ldots
\]

**SSA**

\[
x_1 = \ldots
\]

\[
x_2 = \ldots
\]

\[
x_3 = \phi(x_1, x_2)
\]

\[
\text{use } x_3
\]

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.

\[
n_1 = 1 \quad n_2 = 2
\]

\[
n_3 = \phi(n_1, n_2)
\]

\[
n_4 = \phi(n_2, n_5)
\]

\[
n_5 = n_4 + 1
\]

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 \( \text{let } x = (y + z) \ast (a - b) \text{ in } \ldots \).

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

(We write \( \gg \) for reverse function application: \( a \gg b = b \ a \).)
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 \( \text{let } x = (\text{if } c \text{ then } y + 1 \text{ else } z + 2) \text{ in } \ldots \)

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

\[
\begin{array}{c}
x = \phi(x_1, x_2) \\
x_1 = y + 1 \\
x_2 = z + 2
\end{array}
\]
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 $\beta$

\[
(\ a + b) \gg (\lambda x. \ldots \ldots ) \rightarrow \ldots
\]

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

![Diagram](image)

- **CPS**
- **Source**
- **inverse CPS transf.**

1. CPS transf.
2. Administrative reductions
3. Optimizing reductions
4. Inverse CPS transf.
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:

\[
\begin{align*}
U(k \ W) &= \overline{W} \\
U(\text{let } x = W \text{ in } P) &= \text{let } x = \overline{W} \text{ in } U(P) \\
U(W_1 \ W_2 \ k) &= \overline{W}_1 \overline{W}_2 \quad \text{(tail application)} \\
U(W_1 \ W_2 (\lambda x. P)) &= \text{let } x = \overline{W}_1 \overline{W}_2 \text{ in } P \quad \text{(non-tail application)}
\end{align*}
\]

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

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

arithmetic
function application
datatype constructor
closure constructor

Body:

\[ b ::= c \]
\[ \mid \text{ let } x = c \text{ in } b \]
\[ \mid \text{ if } a \text{ then } b_1 \text{ else } b_2 \]
\[ \mid \text{ match } a \text{ with } \ldots p_i \rightarrow b_i \ldots \]

tail computation
sequencing
conditional
pattern-matching
Another functional IR: A-normal forms

ANF as a CFG

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

\[ x = a + b \]
\[ \text{if } (x >= a) \]
\[ \text{return } x \quad \text{return } 0 \]
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:

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

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.

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

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: \( \approx \) 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.

\[
\begin{align*}
\text{let } x &= 1 \text{ in } \ldots & x + 1 & \ldots & x + y & \ldots \\
\rightarrow & \quad \ldots & 2 & \ldots & 1 + y & \ldots 
\end{align*}
\]

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

\[
\begin{align*}
\text{let } x &= a + b \text{ in } & \text{let } x &= a + b \text{ in} \\
\quad & \ldots & \quad & \quad & \quad \\
\quad \text{let } y &= a + b \text{ in } & \text{let } y &= x \text{ in} \\
\quad & \ldots & \quad & \quad & \quad
\end{align*}
\]
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**

```plaintext
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.

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

\[
\text{match } a \text{ with } \ldots C(x_1, \ldots, x_n) \rightarrow b \ldots \\
\quad \rightarrow \text{ add edges between } x_i \text{ and all } y \in FV(b) \setminus \{x_i\}.
\]
Another functional IR: A-normal forms

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)
```

X. Leroy (INRIA)
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)
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)
Example of an interference graph

```
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)
```
Another functional IR: A-normal forms

Example of an interference graph

let $s = 0.0$ in
let $i = 0$ in
let rec $f(s,i) =$
  if ($ri < size$) then
    let $a = i*4$ in
    let $b = \text{load}(tbl+a)$ in
    let $c = \text{float}(b)$ in
    let $s = s + f c$ in
    let $i = i + 1$ in
    $f(s,i)$
  else
    let $d = \text{float}(size)$ in
    $s / f d$
  in $f(s,i)$
Example of an interference graph

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)
Another functional IR: A-normal forms

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)
```
Another functional IR: A-normal forms

Example of an interference graph

```
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)
```
Example of an interference graph

```
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)
```
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.

Register allocation by graph coloring

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

The interference graphs for SSA graphs and ANF terms are chordal → “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:

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

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

   ```
   let f x = ... in ... f y ... f z ...
   ```

   (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?