

Functional programming languages

Part V: functional intermediate representations

Xavier Leroy

INRIA Paris-Rocquencourt

MPRI 2-4, 2015–2017

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) * (a - b)
-->
t1 = y + z; t2 = a - b; x = t1 * 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.

```
x = y + z
t = load(x + 8)
if (t == 0)
```

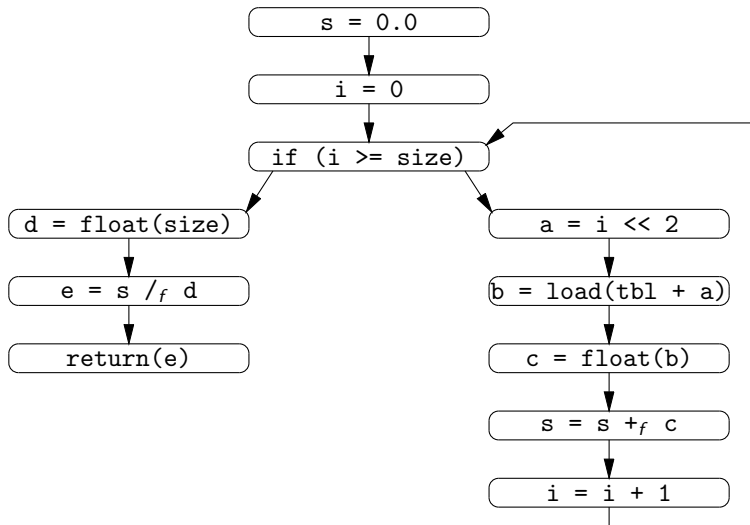
Organized in a control-flow graph:

- 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

a = 1		a = 1
b = 2	-->	b = 2
c = a + b		c = 3
d = x - a		d = x + (-1)

- Dead code elimination

a = 1		nop
b = 2	-->	b = 2
c = 3		c = 3

(if a unused later)

- Common subexpression elimination

$$\begin{array}{l}
 c = a \\
 d = a + b \\
 e = c + b
 \end{array}
 \quad \text{-->} \quad
 \begin{array}{l}
 c = a \\
 d = a + b \\
 e = d
 \end{array}$$

- Hoisting of loop-invariant computations

$$\begin{array}{l}
 L: c = a + b \\
 \dots \\
 \dots \text{ -> } L
 \end{array}
 \quad \text{-->} \quad
 L: \begin{array}{l}
 c = a + b \\
 \dots \\
 \dots \text{ -> } L
 \end{array}$$

- Induction variable elimination

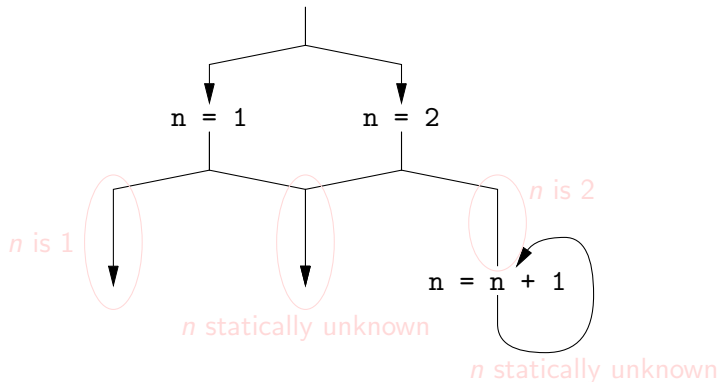
$$\begin{array}{l}
 i = 0 \\
 L: a = i * 4 \\
 b = p + a \\
 \dots \\
 i = i + 1 \text{ -> } L
 \end{array}
 \quad \text{-->} \quad
 \begin{array}{l}
 i = 0 \\
 b = p \\
 L: \dots \\
 b = b + 4 \\
 i = i + 1 \text{ -> } L
 \end{array}$$

- ... 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

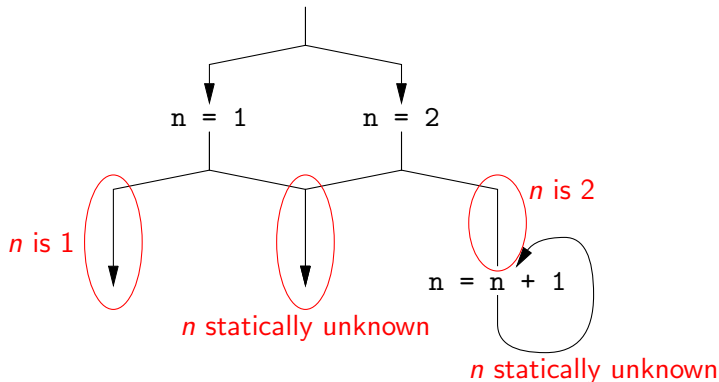
- 1 A given variable or temporary can be defined several times.
(Unavoidable if the source language is imperative.)
- 2 The CFG is not a structured representation of control.



RTL optimizations and dataflow analysis

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

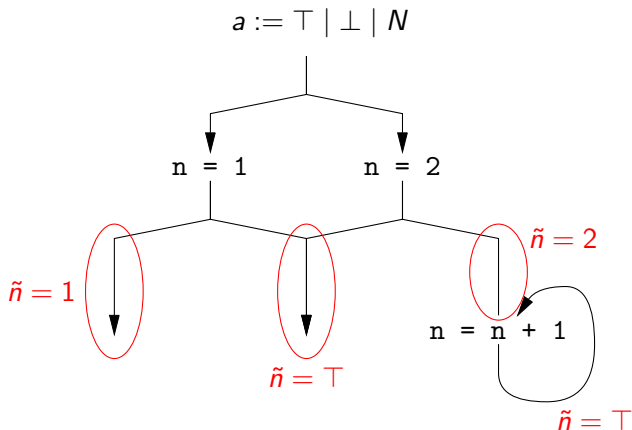
- 1 A given variable or temporary can be defined several times.
(Unavoidable if the source language is imperative.)
- 2 The CFG is not a structured representation of control.



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

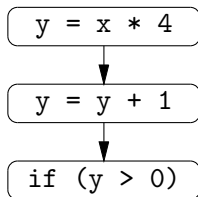


Single Static Assignment (SSA)

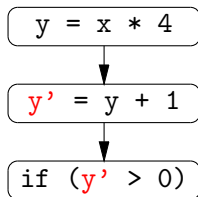
SSA is a variant of RTL where every variable is the result of only one instruction.

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

Not SSA



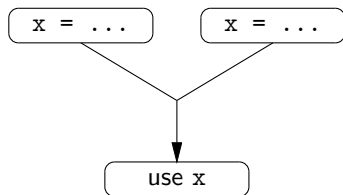
SSA



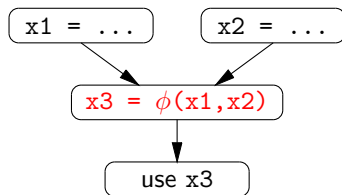
Phi nodes

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

Not SSA



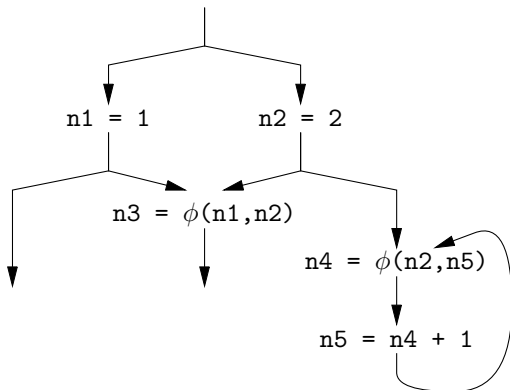
SSA



Informal semantics: $x = \phi(x_1, \dots, x_n)$ 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.



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 ...`

CPS	RTL
$(y + z) \gg (\lambda t.$	$t = y + z;$
$(a - b) \gg (\lambda u.$	$u = a - b;$
$(t * u) \gg (\lambda x.$	$x = t * u;$
$...)))$	$...$

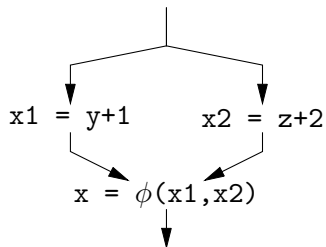
(We write \gg for reverse function application: $a \gg b = b a$.)

CPS as a functional IR

Likewise, let-bound continuations correspond to **join points** in a control-flow graph. Applying such a continuation corresponds to a **ϕ node** in SSA form.

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

```
let k =  $\lambda x$  ... in
if c
then k(y+1)
else k(z+2)
```



Optimizations on CPS terms

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

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

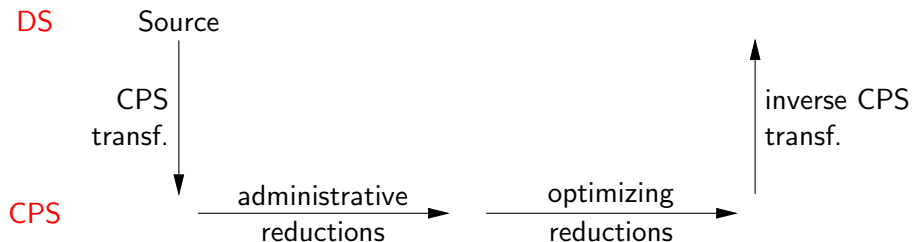
$$\begin{aligned}
 & 1 \gg (\lambda x. \dots x + 1 \dots x + y \dots) \\
 \rightarrow & \dots 2 \dots 1 + y \dots
 \end{aligned}$$

Example: common subexpression elimination \approx inverse β

$$\begin{array}{ccc}
 (a + b) \gg (\lambda x. & & (a + b) \gg (\lambda x. \\
 \dots & \dashrightarrow & \dots \\
 (a + b) \gg (\lambda y. & & \mathbf{x} \gg (\lambda y. \\
 \dots & & \dots
 \end{array}$$

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

Values: $W ::= N \mid x \mid \lambda x.\lambda k.P$

Inverse CPS transformation:

$$U(k W) = \overline{W}$$

$$U(\text{let } x = W \text{ in } P) = \text{let } x = \overline{W} \text{ in } U(P)$$

$$U(W_1 W_2 k) = \overline{W_1} \overline{W_2} \quad (\text{tail application})$$

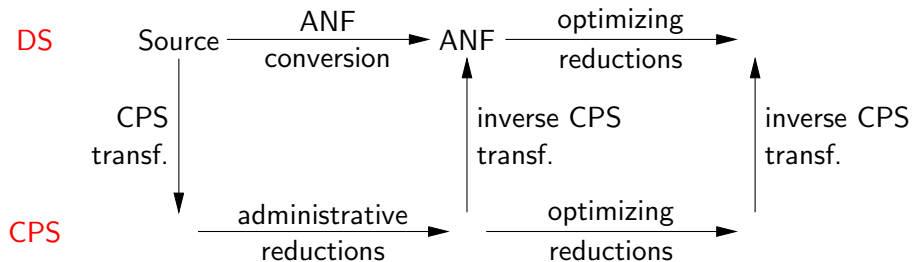
$$U(W_1 W_2 (\lambda x.P)) = \text{let } x = \overline{W_1} \overline{W_2} \text{ in } P \quad (\text{non-tail application})$$

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 $\text{inv-CPS-transf} \circ \text{adm-red} \circ \text{CPS-transf}$.

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:

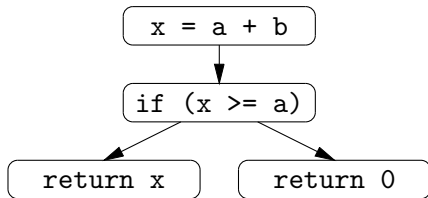
$c ::= a_1 \text{ op } a_2$	arithmetic
$a(\vec{a})$	function application
$C(\vec{a})$	datatype constructor
$\text{closure}(a, \vec{a})$	closure constructor

Body:

$b ::= c$	tail computation
$\text{let } x = c \text{ in } b$	sequencing
$\text{if } a \text{ then } b_1 \text{ else } b_2$	conditional
$\text{match } a \text{ with } \dots p_i \rightarrow b_i \dots$	pattern-matching

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)
      ( $\lambda t. 1 + t$ )
```

Conversion to ANF

Step 2: interpret the result in the Identity monad:

$$\begin{aligned} \text{ret } a &\mapsto a \\ \text{bind } a (\lambda x.b) &\mapsto \text{let } x = a \text{ in } b \end{aligned}$$

Example 2

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

Monadic conversion + identity monad:

```
let t = if x >= 0 then f(x) else ret 0
in 1 + t
```

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.

`let x = (let y = a in b) in c`

→ `let y = a in let x = b in c` (if y not free in c)

`let x = (if a then b else c) in d`

→ `if a then (let x = b in d) else (let x = c in d)`

`let x = (match a with ... $p_i \rightarrow b_i$...) in c`

→ `match a with ... $p_i \rightarrow$ let x = b_i in c ...`

`match (match a with ... $p_i \rightarrow b_i$...) with ... $q_j \rightarrow c_j$...`

→ `match a with ... $p_i \rightarrow$ (match b_i with ... $q_j \rightarrow c_j$...) ...`

Example 3

```
if x >= 0
then let t = f(x) in 1 + t
else let t = 0 in 1 + t
```

Tail duplication, and how to avoid it

Note that possibly large terms can be duplicated:

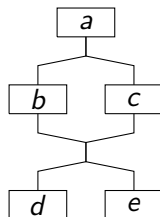
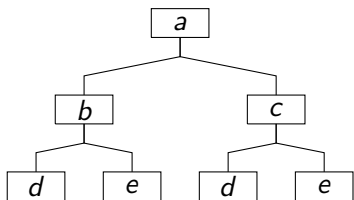
if (if a then b else c) then d else e

→ if a then (if b then d else e) else (if c then d else e)

This can be avoided by using auxiliary functions: (\approx SSA ϕ nodes)

if (if a then b else c) then d else e

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



Optimizations on ANF terms

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

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

$$\begin{array}{l} \text{let } x = 1 \text{ in } \dots x + 1 \dots x + y \dots \\ \rightarrow \dots 2 \dots 1 + y \dots \end{array}$$

Example: common subexpression elimination \approx inverse β

$$\begin{array}{l} \text{let } x = a + b \text{ in} \\ \dots \\ \text{let } y = a + b \text{ in} \\ \dots \end{array} \quad \dashrightarrow \quad \begin{array}{l} \text{let } x = a + b \text{ in} \\ \dots \\ \text{let } y = x \text{ in} \\ \dots \end{array}$$

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 α -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 α -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 α -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 α -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.

let $x = c$ in b

→ add edges between x and all $y \in FV(b) \setminus \{x\}$

match a with ... $C(x_1, \dots, x_n) \rightarrow b \dots$

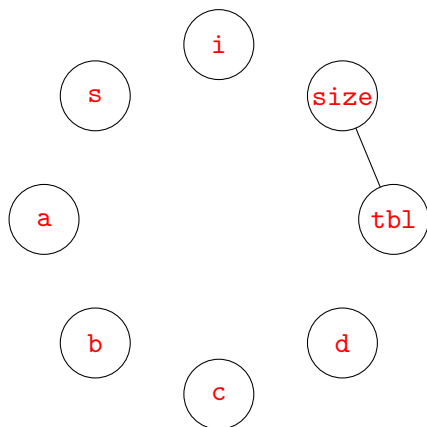
→ add edges between x_i and all $y \in FV(b) \setminus \{x_i\}$.

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)

```

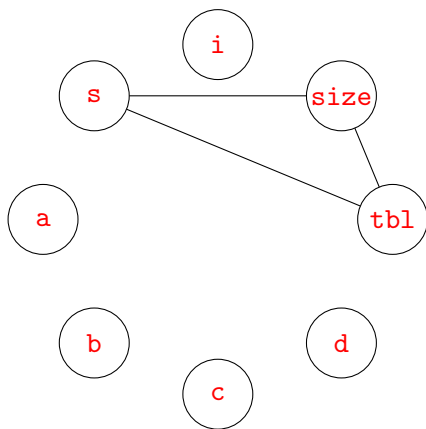


Example of an interference graph

```

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)

```

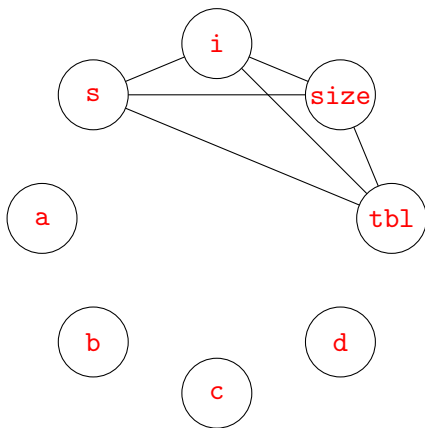


Example of an interference graph

```

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)

```

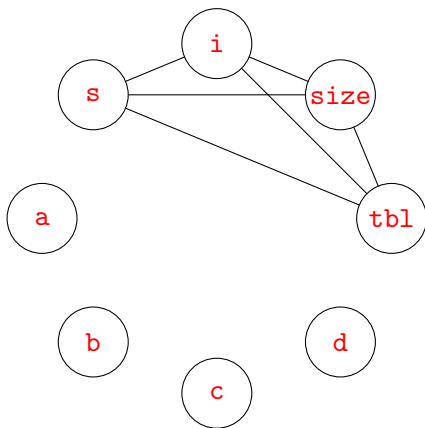


Example of an interference graph

```

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)

```

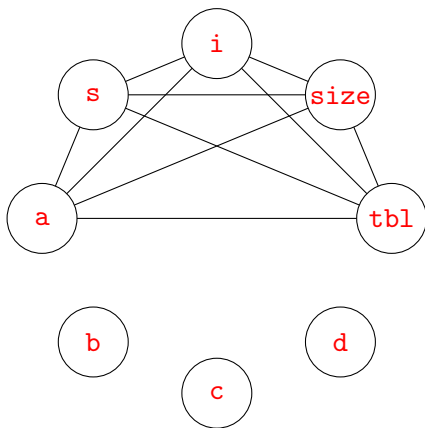


Example of an interference graph

```

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)

```

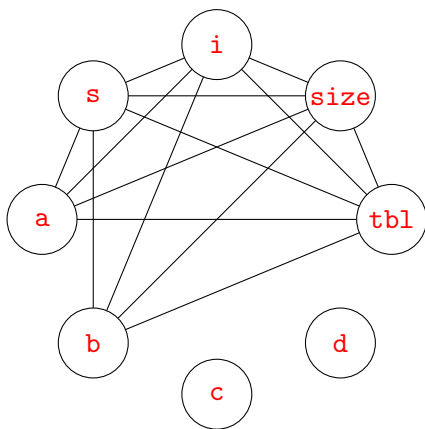


Example of an interference graph

```

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)

```

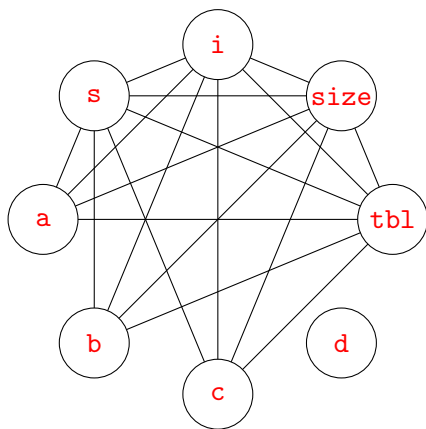


Example of an interference graph

```

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)

```

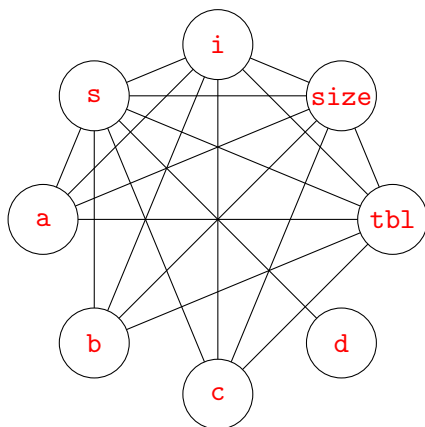


Example of an interference graph

```

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)

```

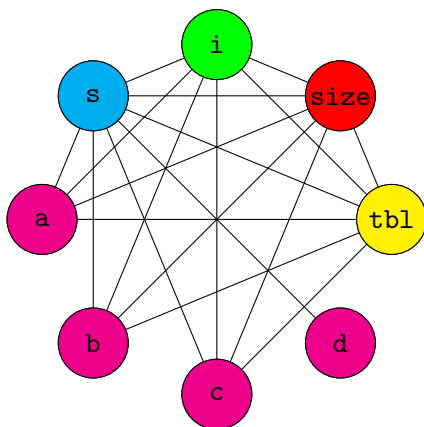


Example of an interference graph

```

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)

```



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.

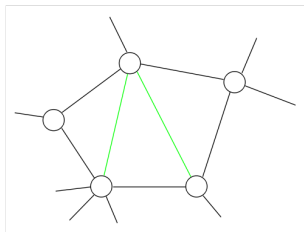
Otherwise, the coloring is a good starting point to determine which variables go into registers.

(A. Appel, *Modern compiler implementation in ML*, Cambridge University Press, chapter 11.)

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?