Functional programming languages

Part V: functional intermediate representations

Xavier Leroy

INRIA Paris-Rocquencourt

MPRI 2-4, 2012–2013

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

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)
```

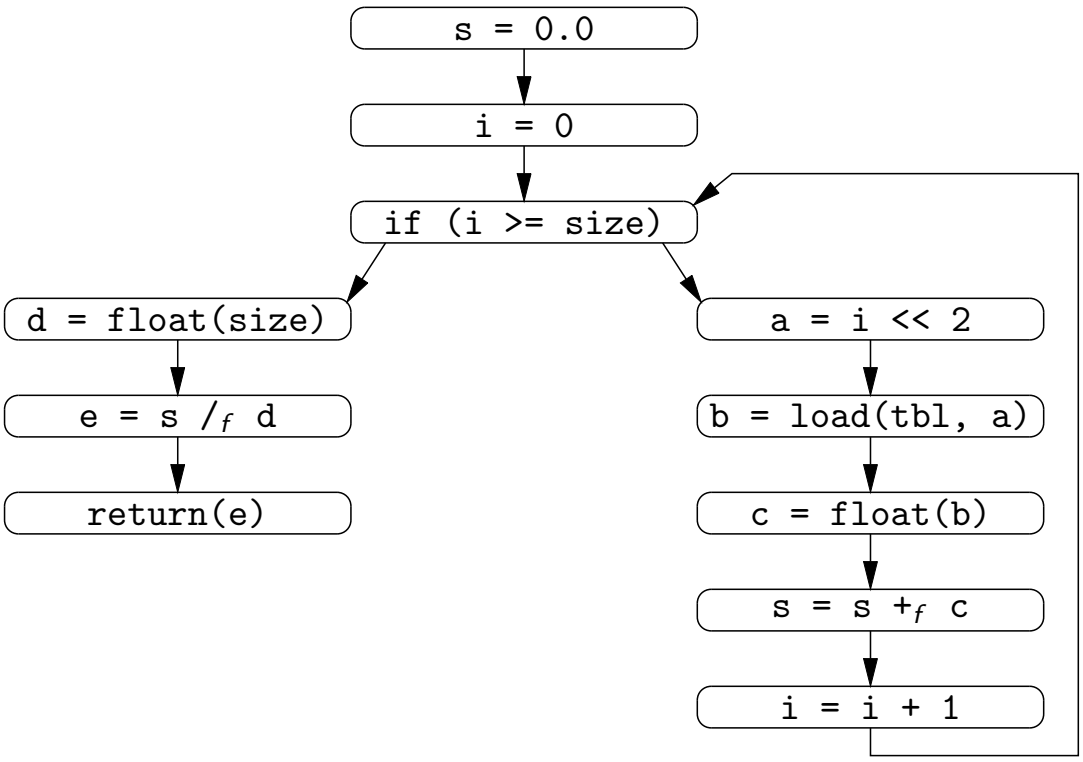
Organized in a control-flow graph:

- 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

```
c = a                          c = a
d = a + b          -->         d = a + b
e = c + b                      e = d
```

- Hoisting of loop-invariant computations

```
L: c = a + b                       c = a + b
   ...                        --> L: ...
   ... -> L                        ... -> L
```

- Induction variable elimination

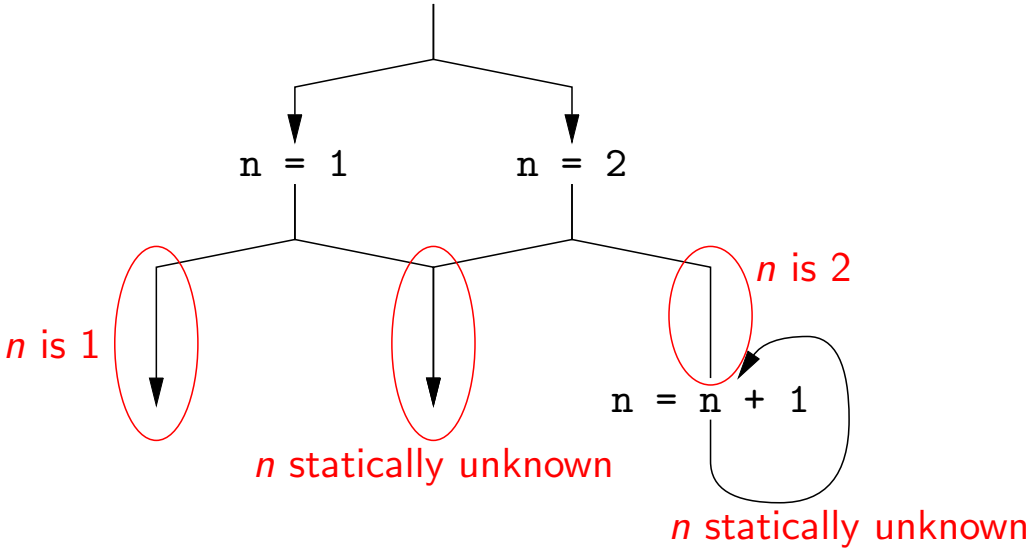
```
   i = 0                           i = 0
L: a = i * 4          -->          b = p
   b = p + a                  L:  ...
   ...                            b = b + 4
   i = i + 1 -> L                 i = i + 1 -> L
```

- ... 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.

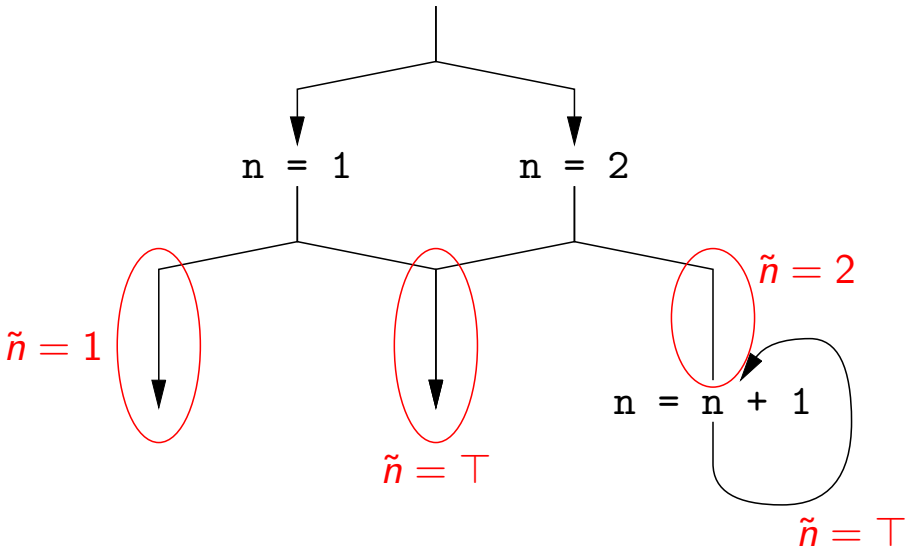


RTL optimizations and dataflow analysis

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

$$a := \top \mid \bot \mid N$$

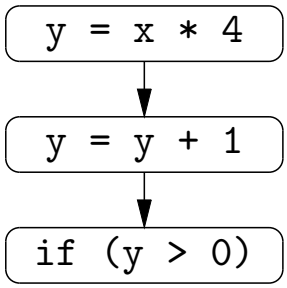


Single Static Assignment (SSA)

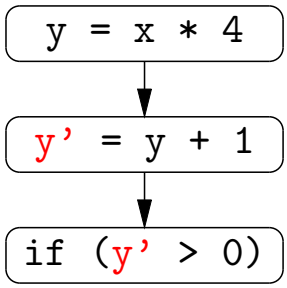
SSA is a variant of RTL where every variable is the result of only one instruction.

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

Not SSA



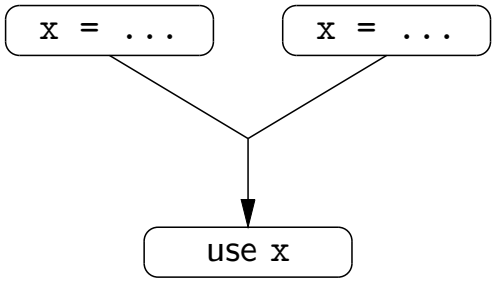
SSA



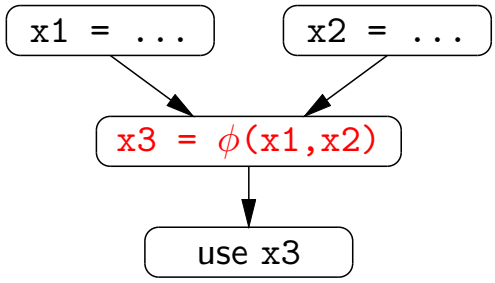
Phi nodes

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

Not SSA



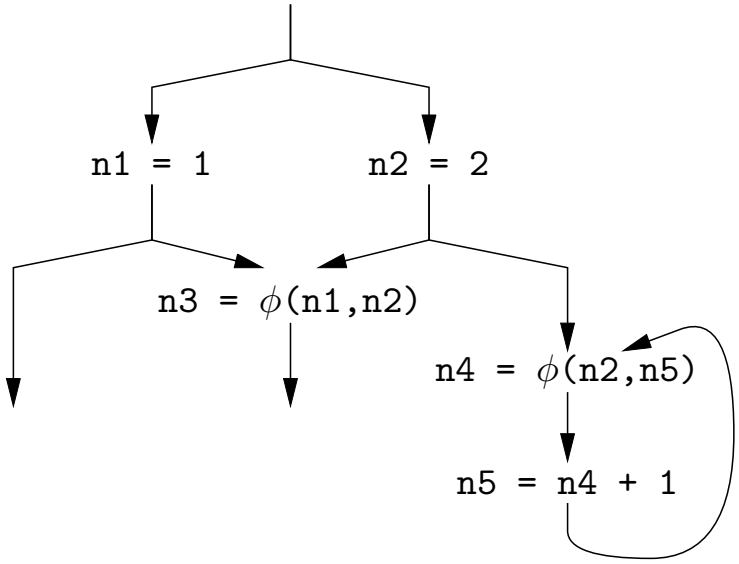
SSA



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



We have $\tilde{\mathtt{n1}} = 1$ and $\tilde{\mathtt{n2}} = 2$ and $\tilde{\mathtt{n3}} = \tilde{\mathtt{n4}} = \tilde{\mathtt{n5}} = \top$ everywhere.

Outline

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)` $\gg$ `(`λ`t.`	`t = y + z;`
`(a - b)` $\gg$ `(`λ`u.`	`u = a - b;`
`(t * u)` $\gg$ `(`λ`x.`	`x = t * u;`
`...)))`	`...`

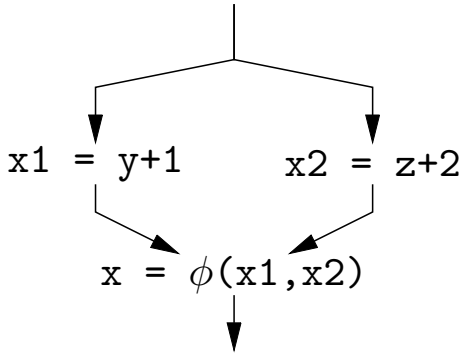
(We write $\gg$ for reverse function application: $a \gg b = b\ a$.)

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 = λx ... 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.

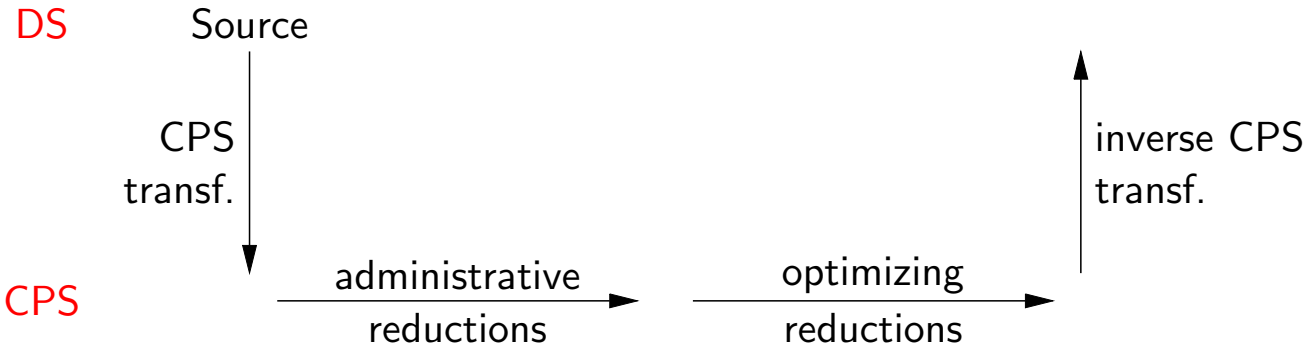
$$\begin{array}{rl} & 1 \gg (\lambda x.\ \ldots\ x+1\ \ldots\ x+y\ \ldots) \\ \rightarrow & \ldots\ 2\ \ldots\ 1+y\ \ldots \end{array}$$

Example: common subexpression elimination $\approx$ inverse β

```
(a + b) ≫ (λx.                (a + b) ≫ (λx.
...                    --->   ...
(a + b) ≫ (λy.                x ≫ (λy.
...                           ...
```


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

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

Inverse CPS transformation:

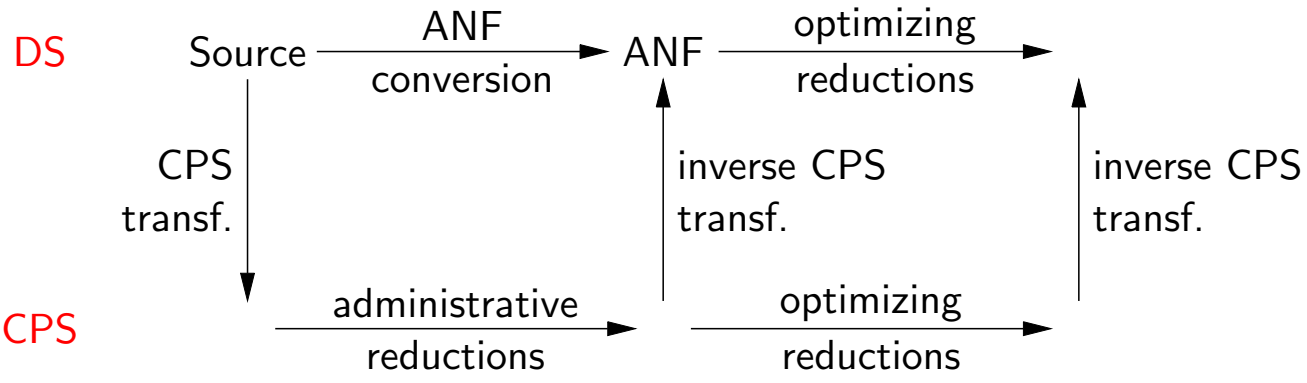
$$\begin{aligned} U(k\ W) &= \overline{W} \\ U(\texttt{let}\ x = W\ \texttt{in}\ P) &= \texttt{let}\ x = \overline{W}\ \texttt{in}\ U(P) \\ U(W_1\ W_2\ k) &= \overline{W_1}\ \overline{W_2} \quad \text{(tail application)} \\ U(W_1\ W_2\ (\lambda x.P)) &= \texttt{let}\ x = \overline{W_1}\ \overline{W_2}\ \texttt{in}\ P \quad \text{(non-tail application)} \end{aligned}$$

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 ∘ adm-red ∘ CPS-transf.

Outline

Syntax of ANF

Atom:

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

Computation:

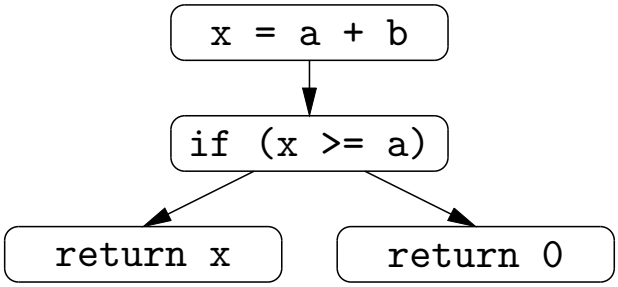
$c ::= a_1\ op\ a_2$	arithmetic
$\mid a(\vec{a})$	function application
$\mid C(\vec{a})$	datatype constructor
$\mid \texttt{closure}(a, \vec{a})$	closure constructor

Body:

$b ::= c$	tail computation
$\mid \texttt{let}\ x = c\ \texttt{in}\ b$	sequencing
$\mid \texttt{if}\ a\ \texttt{then}\ b_1\ \texttt{else}\ b_2$	conditional
$\mid \texttt{match}\ a\ \texttt{with}\ \ldots p_i \rightarrow b_i \ldots$	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: $1 + (\texttt{if}\ x >= 0\ \texttt{then}\ f(x)\ \texttt{else}\ 0)$

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:

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

Example 2

Source term: $1 + (\texttt{if}\ x >= 0\ \texttt{then}\ f(x)\ \texttt{else}\ 0)$

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`.

$$\begin{array}{l} \texttt{let}\ x = (\texttt{let}\ y = a\ \texttt{in}\ b)\ \texttt{in}\ c \\ \quad \rightarrow \quad \texttt{let}\ y = a\ \texttt{in let}\ x = b\ \texttt{in}\ c \quad (\text{if } y \text{ not free in } c) \\ \texttt{let}\ x = (\texttt{match}\ a\ \texttt{with}\ \ldots\ p_i \rightarrow b_i \ldots)\ \texttt{in}\ c \\ \quad \rightarrow \quad \texttt{match}\ a\ \texttt{with}\ \ldots\ p_i \rightarrow \texttt{let}\ x = b_i\ \texttt{in}\ c \ldots \\ \texttt{match}\ (\texttt{match}\ a\ \texttt{with}\ \ldots\ p_i \rightarrow b_i \ldots)\ \texttt{with}\ \ldots\ q_j \rightarrow c_j \ldots \\ \quad \rightarrow \quad \texttt{match}\ a\ \texttt{with}\ \ldots\ p_i \rightarrow (\texttt{match}\ b_i\ \texttt{with}\ \ldots\ q_j \rightarrow c_j \ldots) \ldots \end{array}$$

Example 3

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

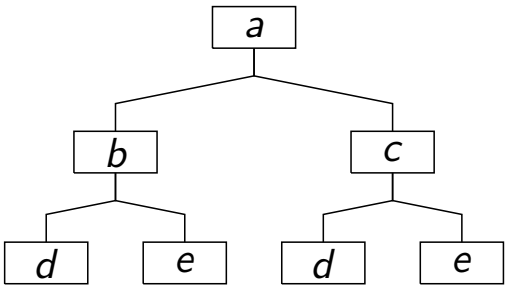
Tail duplication, and how to avoid it

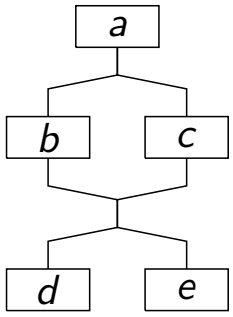
Note that possibly large terms can be duplicated:

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

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

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





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.

$$\texttt{let } x = 1 \texttt{ in } \ldots\ x + 1\ \ldots\ x + y\ \ldots$$
$$\rightarrow \quad \ldots\ 2\ \ldots\ 1 + y\ \ldots$$

Example: common subexpression elimination $\approx$ inverse β

```
let x = a + b in                    let x = a + b in
...                          -->    ...
let y = a + b in                    let y = x in
...                                 ...
```

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.

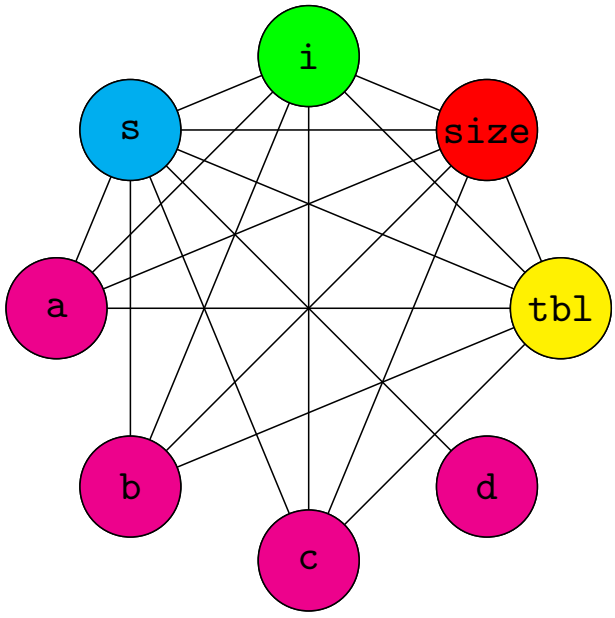
Constructing the interference graph: at each point where a variable x is bound, add edges with all other variables that occur free in the continuation of this binding.

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

`match` a `with` $\ldots C(x_1, \ldots, x_n) \rightarrow b \ldots$
$\rightarrow$ 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)
```



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.

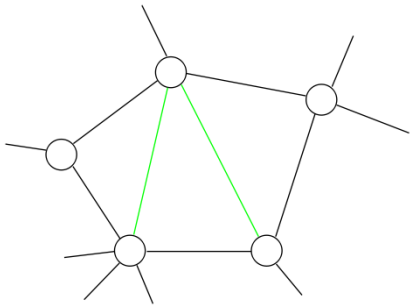
Otherwise, the coloring is a good starting point to determine which variables go into registers.

(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?