

# Functional programming languages

## Part V: functional intermediate representations

Xavier Leroy

INRIA Rocquencourt

MPRI 2-4-2, 2007

## 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.

# 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)
```

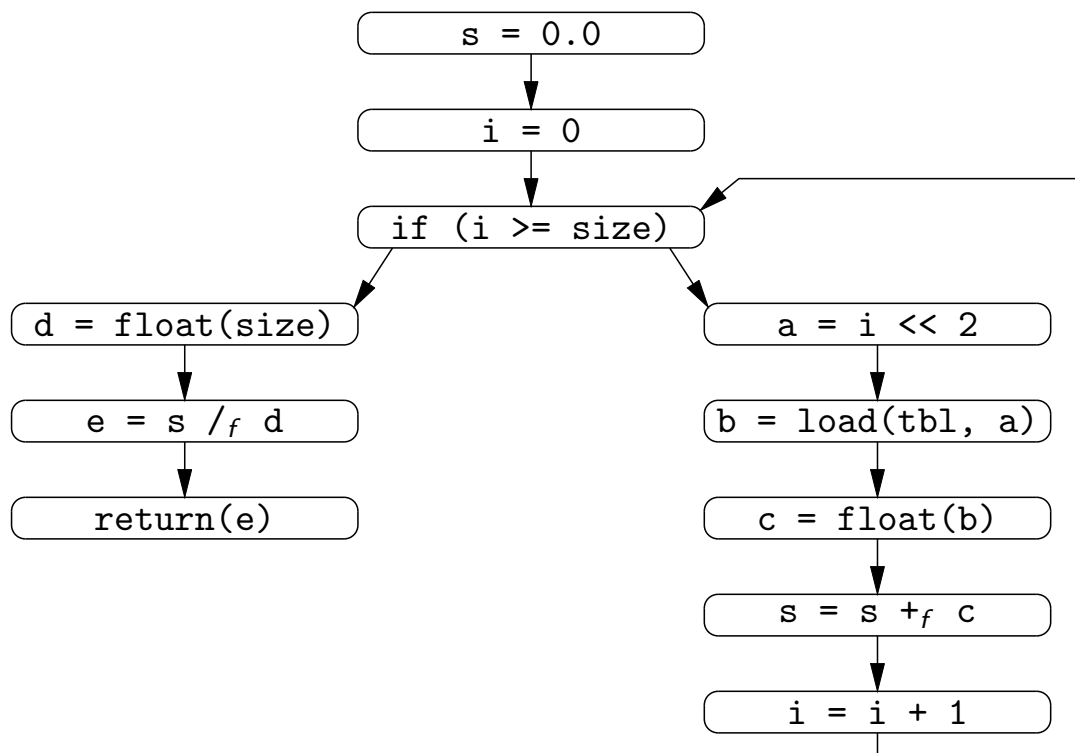
Organized in a control-flow graph:

- 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

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

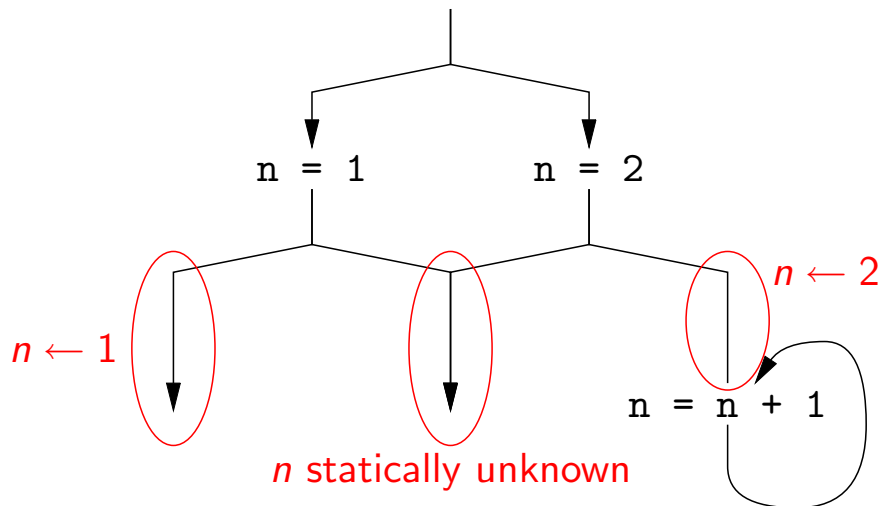
i = 0			i = 0
L: a = i * 4		-->	b = p
b = p + a			L: ...
...			b = b + 4
i = i + 1 -> L			i = i + 1 -> L

- ... 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

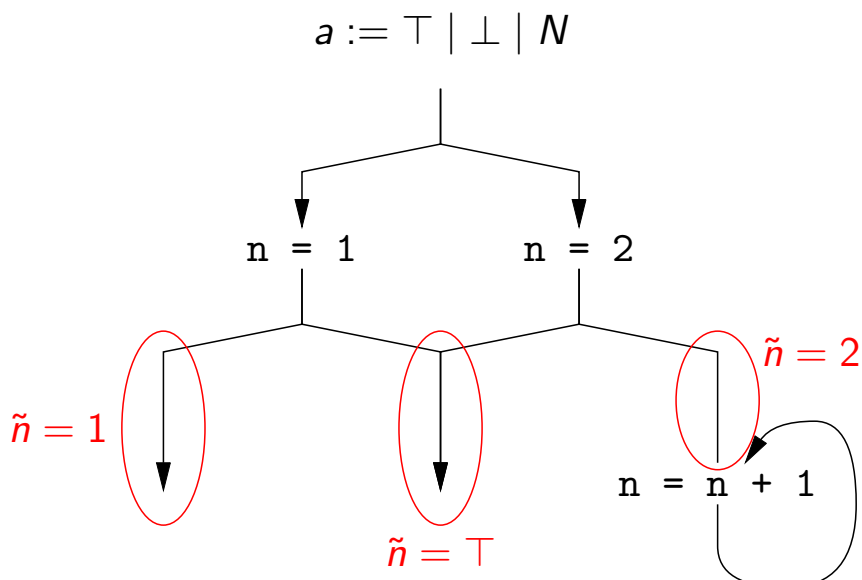
- ① A given variable or temporary can be defined several times.  
(Unavoidable if the source language is imperative.)
- ② The CFG is not a structured representation of control.



## 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



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

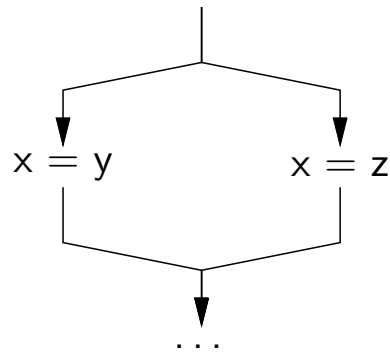
(We write  $\$$  for reverse function application:  $a \$ b = b a$ .)

# CPS as a functional IR

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

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

```
let k = λx ... in
if c
then k y
else k z
```



## 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.

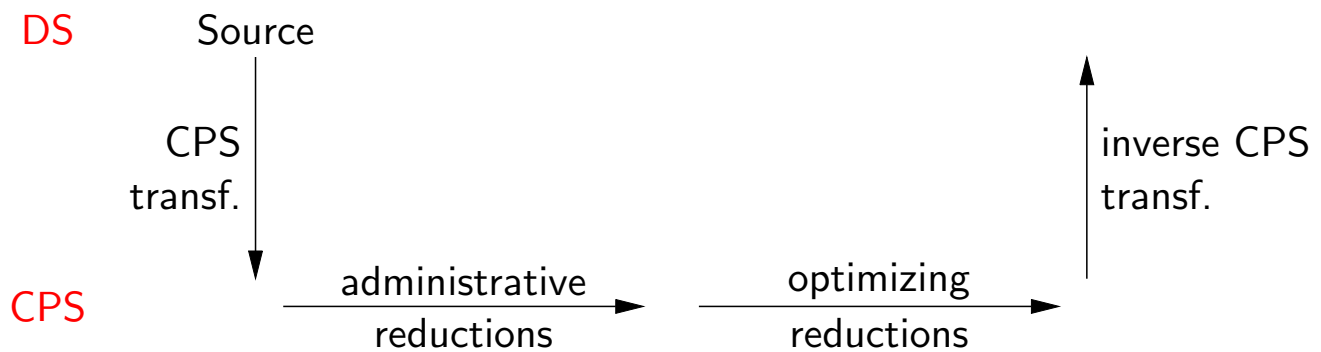
$$\begin{aligned}
 & 1 \$ (\lambda x. \dots x + 1 \dots x + y \dots) \\
 \rightarrow & \dots 2 \dots 1 + y \dots
 \end{aligned}$$

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

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

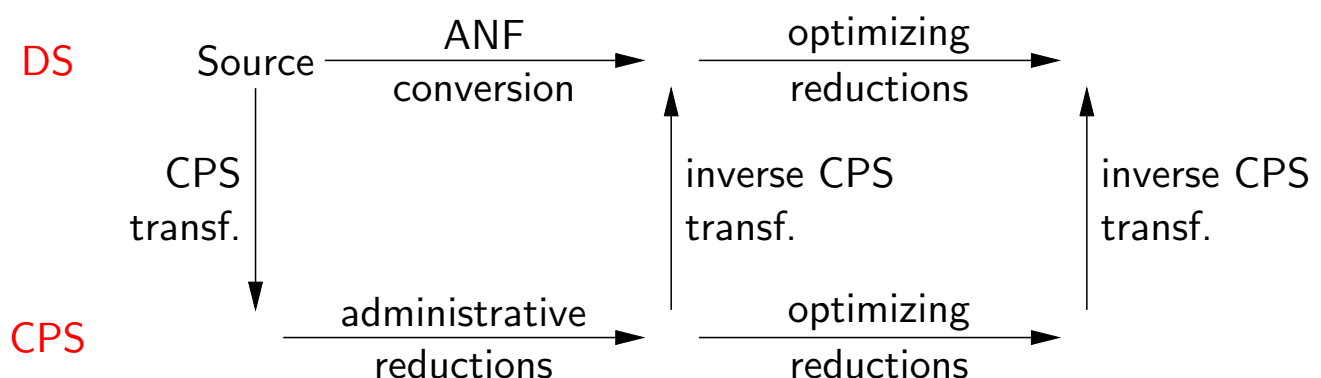
## Back to direct style

To support stack-allocation of activation records, several functional compilers perform an **inverse CPS transformation** after CPS optimization, to recover direct-style 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  $\text{inv-CPS-transf} \circ \text{adm-red} \circ \text{CPS-transf}$ .

(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:

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

Computation:

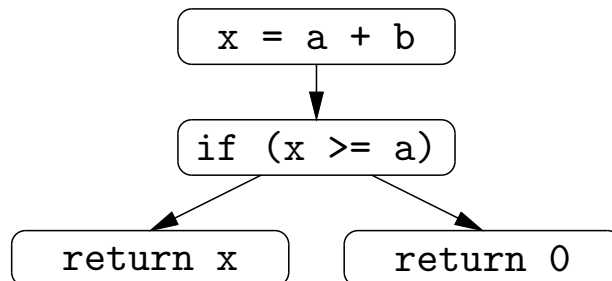
$c ::= a_1 \text{ op } a_2$	arithmetic
$\mid a(\vec{a})$	function application
$\mid C(\vec{a})$	datatype constructor
$\mid \text{closure}(a, \vec{a})$	closure constructor

Body:

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

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
```

## Conversion to ANF

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

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

### Example 3

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

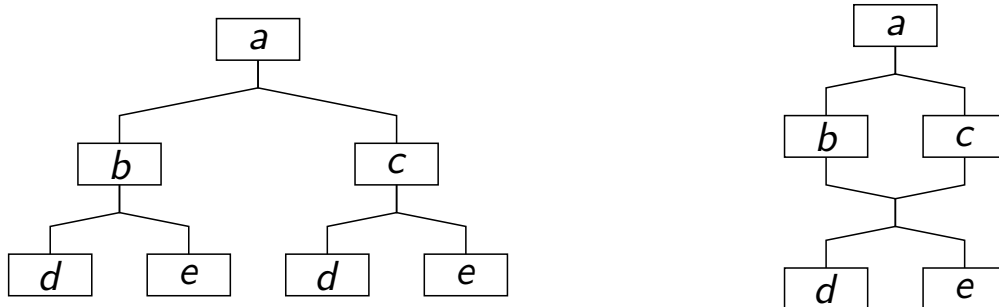
## Tail duplication, and how to avoid it

Note that possibly large terms can be duplicated:

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

This can be avoided by using auxiliary functions:

if (if  $a$  then  $b$  else  $c$ ) then  $d$  else  $e$   
 $\rightarrow$  let  $f(x) =$  if  $x$  then  $d$  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  $\beta$  or arithmetic reductions over ANF terms.

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

let  $x = 1$  in ...  $x + 1$  ...  $x + y$  ...  
 $\rightarrow$  ...  $2$  ...  $1 + y$  ...

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

let  $x = a + b$  in  
 ...  
 let  $y = a + b$  in  
 ...  
 $\rightarrow$   
 let  $x = a + b$  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  $\alpha$ -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.

let  $x = c$  in  $b$

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

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

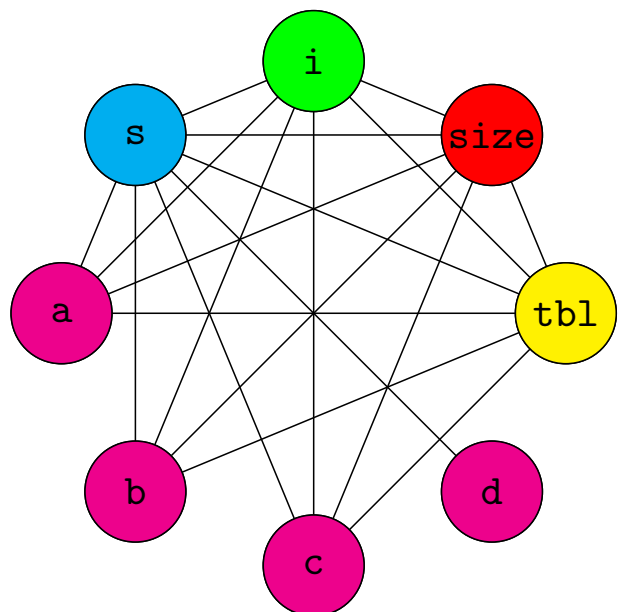
→ add edges between  $x_i$  and all  $y \in FV(b) \setminus \{x_i\}$ .

## 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)

```



## 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 U. Press, esp. chapter 11.

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