Compiling functional languages

Xavier Leroy

INRIA Rocquencourt

Spring school “Semantics of programming languages”
Agay, 24-29 March 2002

Introduction

Goal of this lecture: survey implementation techniques for functional languages, and show some of the (syntactic) theories explaining these techniques.

What is a functional language?

- Examples: Caml, Haskell, Scheme, SML, ...
- "A language where functions are taken seriously".
- A language that supports the manipulation of functions as first-class values, roughly as predicted by the λ-calculus.

Functional languages = applied λ-calculus

Recipe for a functional language:

- Fix an evaluation strategy on the λ-calculus.
  - Weak evaluation (no reductions under λ).
  - Call-by-name or call-by-value.
- Add constants and primitive operations
  - Integers and arithmetic operations.
  - Booleans, strings, characters, input/output, ...
- Add primitive data structures
  - Predefined: tuples, lists, vectors, ...
  - User-defined: records, sums, recursive data types.

Outline

1. Representing functions as values: closures and environments.
2. Abstract machines for functional languages.
3. Optimizing functions: control-flow analysis.
4. Optimizing data representations.

Part 1

Function closures

Functions as values in the presence of free variables

```
let scale = λn. λx. n * x

let scale_by_2 = scale 2

let scale_by_10 = scale 10
```

`scale 2` should return a function computationally equivalent to `λx. x * 2`, in accordance with the β-reduction rule:

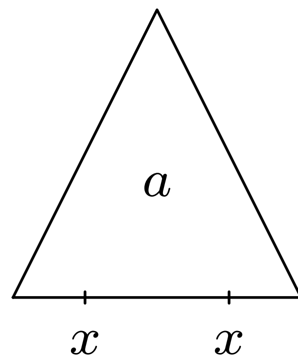
$$(\lambda x.a)\ b \rightarrow a\{x \leftarrow b\}$$

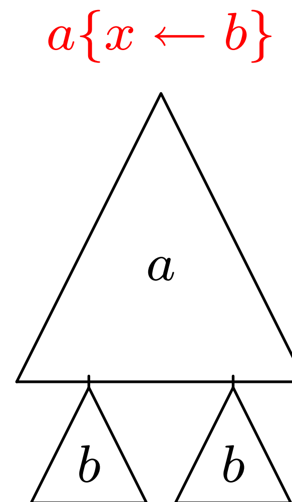
Textual substitution is the problem, not the solution

$$(\lambda x.a)\ b \rightarrow a\{x \leftarrow b\}$$

Naive substitution involves a full copy of the term a, replacing occurrences of x by b.

(If b has free variables, a copy of b may also be necessary to avoid variable captures. However, with head reductions, b is guaranteed to be closed.)





The problem gets worse in a compiled setting

In a compiled setting, the standard representation for a function value is a pointer to a piece of compiled code that:

- expects its argument x in a register **`arg`**;
- computes the function body a;
- leaves the result of a in a register **`res`**;
- returns to caller.

If we naively apply this "function as code pointer" model to a functional language, functions that return functions should dynamically generate a piece of compiled code representing the returned function.

Example:

```
let scale = λn. λx. n * x
```

```
scale 2 returns                 scale 10 returns

   mov res, arg                   mov res, arg
   mul res, 2                     mul res, 10
   return                         return
```

Problem: this requires run-time code generation, which is complex and expensive (in time and in space for the generated code blocks).

Towards a better solution

Remark: the generated blocks of code share the same "shape": they differ only in the value of the variable `n` that is free in the returned function λ`x. x * n`.

```
mov res, arg
mul res, <the value of n passed to scale>
return
```

Idea: share the common code and put the varying parts (i.e. the values of free variables) in some separate data structure called an environment.

The same idea applies to term-level substitution: to represent $a\{x \leftarrow b\}$, just keep a unchanged and record separately the binding of x to b in an environment.

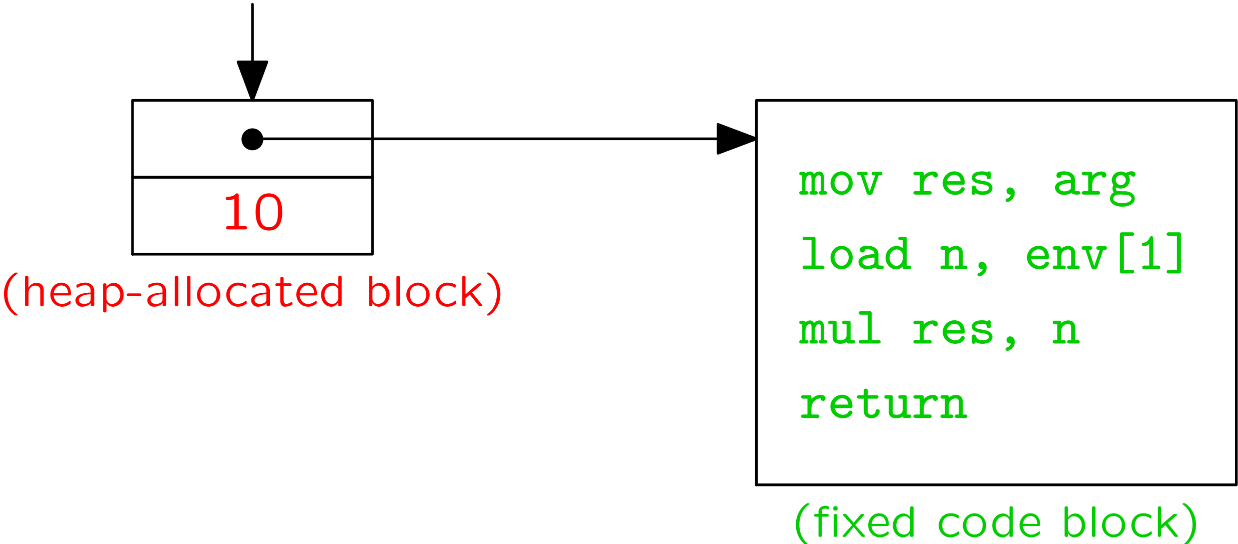
Closures (P. J. Landin, 1964)

All functional values are represented by closures.

Closures are dynamically-allocated data structures containing:

- a code pointer, pointing to a fixed piece of code computing the function result;
- an environment: a record providing values for the free variables of the function.

To apply a closure, move the environment part in a conventional register `env` and call the code pointer.


return value of scale 10
10
(heap-allocated block)
mov res, arg
load n, env[1]
mul res, n
return
(fixed code block)

Closures in an interpreted, term-based setting

Closures can also be viewed as terms

$$(\lambda x.a)[e]$$

where $\lambda x.a$ is a function with free variables y, z, . . . , and the environment e is an explicit substitution binding terms b, c, . . . to y, z, . . .

This is the key to efficient interpreters for functional languages, as we shall now see via the following route:

- Standard semantics: small-step (reduction-based), classical substitutions.
- Big-step semantics with classical substitutions.
- Big-step semantics with environments.
- Small-step semantics with environments / explicit substitutions.

Small-step semantics with classical substitutions

Terms: $a ::= x \mid \lambda x.a \mid a_1\ a_2 \mid cst \mid op(a_1, \ldots, a_n)$

Constants: $c ::= 0 \mid 1 \mid \ldots$

Operators: $op ::= + \mid \mathtt{pair} \mid \mathtt{fst} \mid \mathtt{snd} \mid \ldots$

Values: $v ::= \lambda x.a \mid cst$

Reduction rules:

$$\begin{array}{rcll} (\lambda x.a)\ v & \rightarrow & a\{x \leftarrow v\} & (\beta_v) \\ op(v_1, \ldots, v_n) & \rightarrow & v \text{ if } v = \overline{op}(v_1, \ldots, v_n) & (\delta) \end{array}$$

Strategy: head reductions, call-by-value, left-to-right

$$\frac{a \rightarrow a'}{a\ b \rightarrow a'\ b} \qquad \frac{b \rightarrow b'}{v\ b \rightarrow v\ b'}$$

$$\frac{a \rightarrow a'}{op(v_1, \ldots, v_{k-1}, a, b_{k+1}, \ldots, b_n) \rightarrow op(v_1, \ldots, v_{k-1}, a', b_{k+1}, \ldots, b_n)}$$

Big-step semantics with classical substitutions

Rather than chain elementary reductions $a \rightarrow a_1 \rightarrow \ldots \rightarrow v$, define a "big step" evaluation relation $a \Rightarrow v$ that "jumps" from a to its value v. (a must be closed.)

$$(\lambda x.a) \Rightarrow (\lambda x.a) \qquad\qquad cst \Rightarrow cst$$

$$\frac{a \Rightarrow (\lambda x.c) \qquad b \Rightarrow v \qquad c\{x \leftarrow a\} \Rightarrow v'}{a\ b \Rightarrow v'}$$

$$\frac{a_1 \Rightarrow v_1 \qquad \ldots \qquad a_n \Rightarrow v_n \qquad v = \overline{op}(v_1, \ldots, v_n)}{op(a_1, \ldots, a_n) \Rightarrow v}$$

Theorem: if a is closed, $a \xrightarrow{*} v$ if and only if $a \Rightarrow v$.

Big-step semantics with environments

Function values are closures $(\lambda x.a)[e]$.

Terms: $a ::= x \mid \lambda x.a \mid a_1\ a_2 \mid cst \mid op(a_1, \ldots, a_n)$

Values: $v ::= (\lambda x.a)[e] \mid cst$

Environments: $e ::= \{x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n\}$

The evaluation relation becomes $e \vdash a \Rightarrow v$.

a may now contain free variables, but those must be bound by the environment e.

$$e \vdash x \Rightarrow e(x) \qquad e \vdash (\lambda x.a) \Rightarrow (\lambda x.a)[e] \qquad e \vdash cst \Rightarrow cst$$

$$\frac{e \vdash a \Rightarrow (\lambda x.c)[e'] \qquad e \vdash b \Rightarrow v \qquad e' + \{x \leftarrow v\} \vdash c \Rightarrow v'}{e \vdash a\ b \Rightarrow v'}$$

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad \ldots \qquad e \vdash a_n \Rightarrow v_n \qquad v = \overline{op}(v_1, \ldots, v_n)}{e \vdash op(a_1, \ldots, a_n) \Rightarrow v}$$

Reformulation with de Bruijn indices

De Bruijn notation: rather than identify variables by names, identify them by position.

```
λx. (λy. y x) x
         | |  |
λ   (λ   1 2) 1
```

Environments become sequences of values $v_1 \ldots v_n.\varepsilon$, accessed by position: variable number n is bound to v_n.

Reformulation with de Bruijn indices

Terms: $a ::= n \mid \lambda a \mid a_1\ a_2 \mid cst \mid op(a_1, \ldots, a_n)$

Values: $v ::= (\lambda a)[e] \mid cst$

Environments: $e ::= \varepsilon \mid v.e$

$$e \vdash n \Rightarrow e(n) \qquad e \vdash (\lambda a) \Rightarrow (\lambda a)[e] \qquad e \vdash cst \Rightarrow cst$$

$$\frac{e \vdash a \Rightarrow (\lambda c)[e'] \qquad e \vdash b \Rightarrow v \qquad v.e' \vdash c \Rightarrow v'}{e \vdash a\ b \Rightarrow v'}$$

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad \ldots \qquad e \vdash a_n \Rightarrow v_n \qquad v = \overline{op}(v_1, \ldots, v_n)}{e \vdash op(a_1, \ldots, a_n) \Rightarrow v}$$

A straightforward yet efficient interpreter

All this leads to the canonical efficient interpreter for call-by-value λ-calculus:

```
type term = Var of int | Lambda of term | App of term * term
          | Constant of int | Primitive of primitive * term list
and value = Int of int | Closure of term * environment
and environment = value list
and primitive = value list -> value

let rec eval env term =
  match term with
    Var n -> List.nth env n
  | Lambda a -> Closure(a, env)
  | App(a, b) ->
      let (Closure(c, env')) = eval env a in
      let v = eval env b in
      eval (v :: env') c
  | Constant n ->
      Int n
  | Primitive(p, arguments) ->
      p (List.map (eval env) arguments)
```

Small-step semantics with explicit substitutions

Big-step semantics are good for implementing interpreters, but small-step semantics allow easier reasoning on non-terminating or "stuck" evaluations.

The notion of environment can be internalized in a small-step, reduction-based semantics: λ-calculus with explicit substitutions.

Terms: $a ::= n \mid \lambda a \mid a_1\ a_2 \mid a[e]$

Environments: $e ::= \varepsilon \mid a.e$

Basic reduction rules:

$$
\begin{array}{rcll}
1[a.e] & \rightarrow & a & \text{(FVar)} \\
(n+1)[a.e] & \rightarrow & n[e] & \text{(RVar)} \\
(\lambda a)[e]\ b & \rightarrow & a[b.e] & \text{(Beta)} \\
(\lambda a)\ b & \rightarrow & a[b.\varepsilon] & \text{(Beta1)} \\
(a\ b)[e] & \rightarrow & a[e]\ b[e] & \text{(App)}
\end{array}
$$

More complete calculi of explicit substitutions

The previous rules are the minimal rules we need to describe weak call-by-value reduction.

To describe other strategies, including strong reduction, richer calculi of explicit substitutions are needed:

Explicit substitutions, M. Abadi, L. Cardelli, P.L. Curien, J.J. Lévy, *Journal of Functional Programming* 6(2), 1996.

Confluence properties of weak and strong calculi of explicit substitutions, P.L. Curien, T. Hardin, J.J. Lévy, *Journal of the ACM* 43(2), 1996.

Closure representation strategies

- When compiling an application, nothing is known about the closure being called (this can be the closure of any function in the program).

  $\rightarrow$ The code pointer must be at a fixed, predictable position in the closure block.

- The environment part of a closure is not accessed during application. Its structure matters only to the code that builds the closure and the code for the function body.

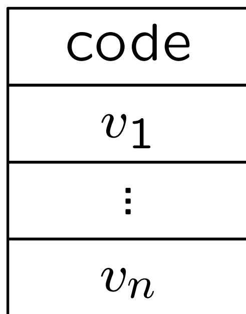
  $\rightarrow$ Considerable flexibility in choosing the layout for the environment part.

- The environment used for evaluation (e in $e \vdash a \Rightarrow v$) need not have the same structure as the environment put in closures (e' in $(\lambda a)[e']$).

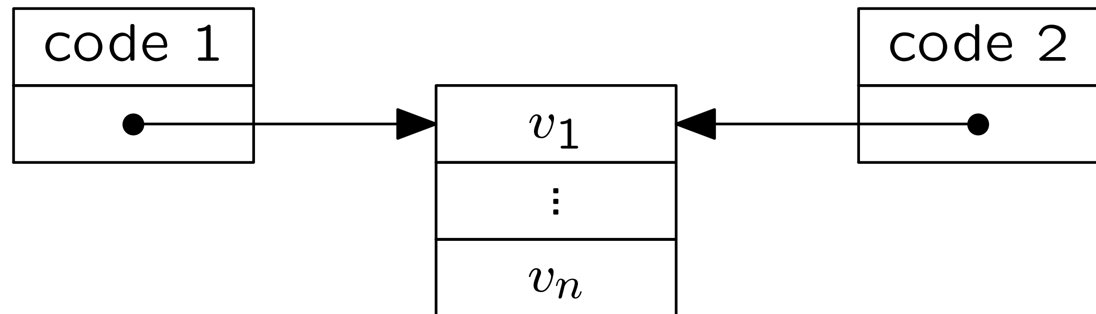
  $\rightarrow$ Even more flexibility in putting part of the evaluation environment e in registers or on the stack rather than in a heap-allocated block.

Examples of closure representations

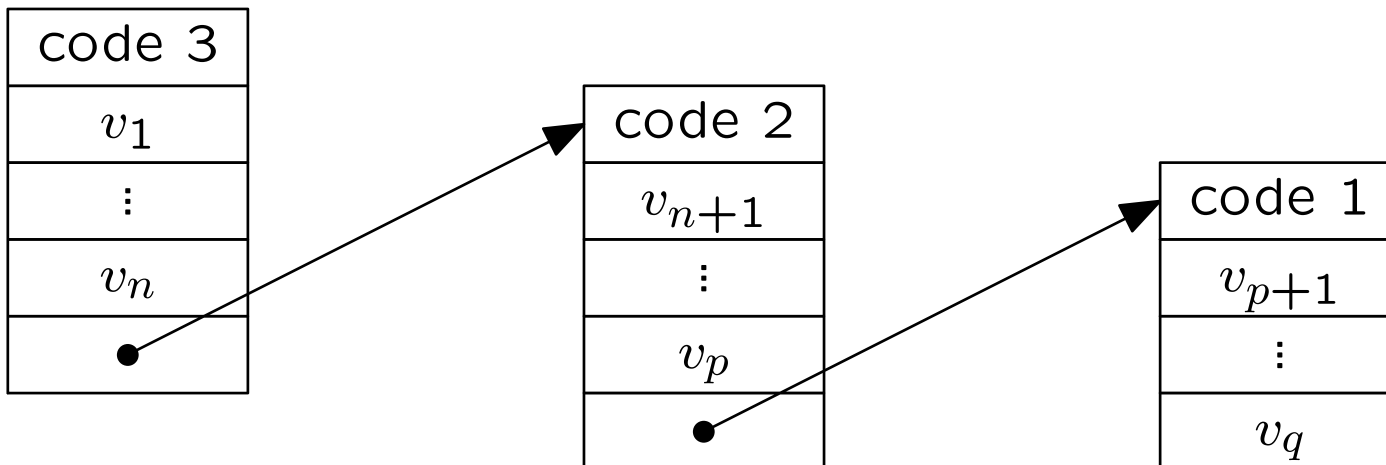
One-block closure



Two-block closures
(with environnent sharing)



Linked closures



Choosing a closure representation

Time trade-off:

- One-block closures: slower to build; faster access to variables.
- Linked closures: faster to build; slower access to variables.

Space trade-off:

- Minimal environments: (bind only the free variables) fewer opportunities for sharing; avoid space leaks.
- Larger environments: (may bind more variables) more opportunities for sharing; may cause severe space leaks.

Modern implementations use one-block closures with minimal environments.

Recursive functions

Recursive functions need access to their own closure:

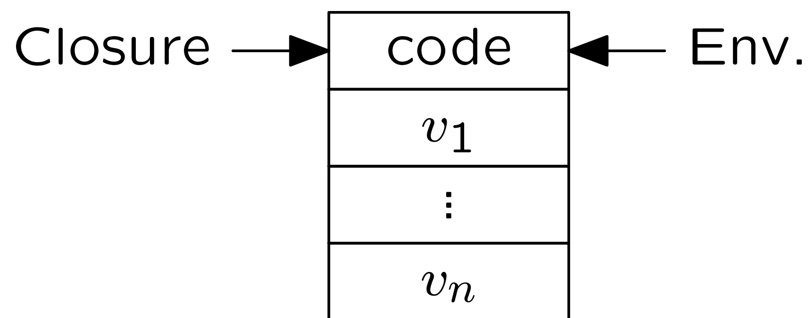
```
let rec f x = ... List.map f l ...
```

(The body of `f` needs to pass the closure of `f` as first argument to the higher-order function `List.map`.)

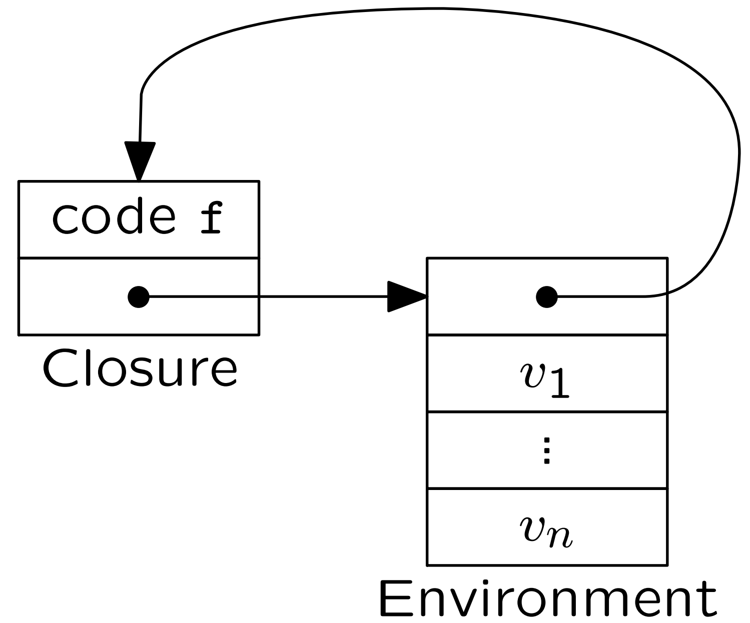
This can be done in several ways:

- Reconstruct the closure of `f` from the current environment.
- Treat `f` as a free variable of the function body: put a pointer to the closure of `f` in the environment of that closure (cyclic closures).
- In the one-block approach: the environment passed to `f` is its own closure, just reuse it.

One-block closure



Cyclic two-block closure



Mutually recursive functions

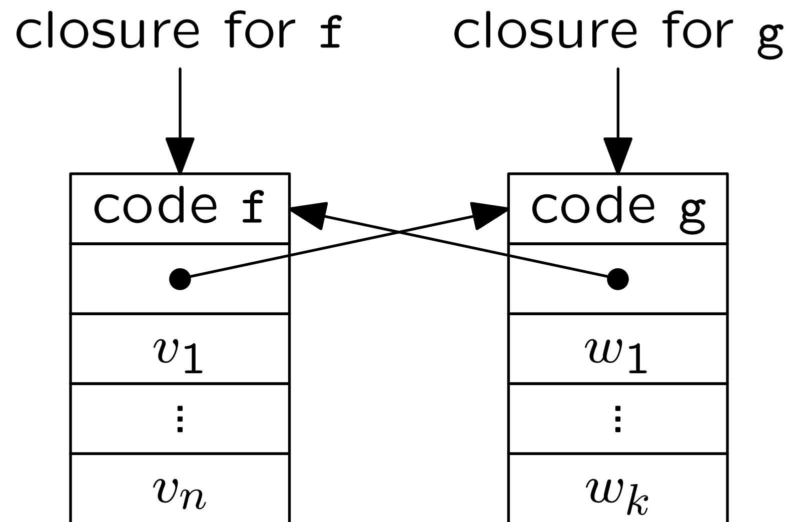
Mutually recursive functions need access to the closures of all functions in the mutual recursive definition:

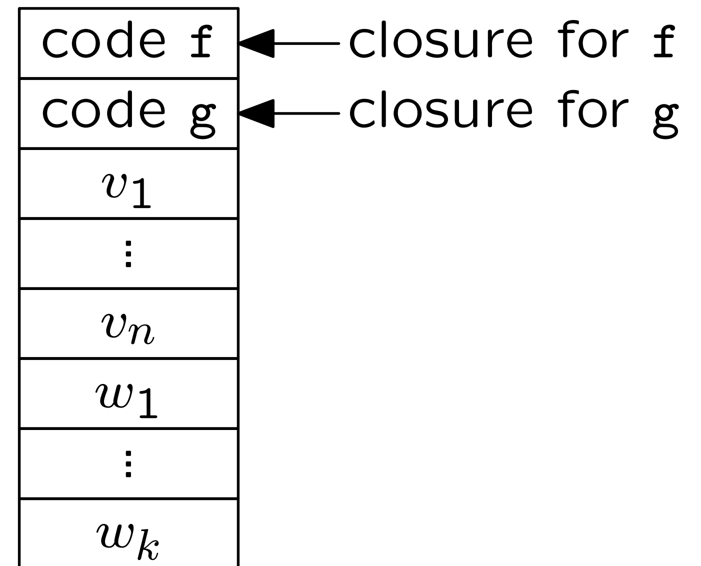
```
let rec f x = ... List.map f l1 ... List.map g l2 ...
and     g y = ... List.map f l3 ...
```

This can be done in two ways:

- The closure for `f` contains a pointer to that of `g` and conversely (cyclic closures).

- Share a closure between `f` and `g` using infix pointers (*Compiling with Continuations*, A. Appel, Cambridge U. Press, 1992.)


Cyclic closures
closure for f
closure for g
code f
code g
v_1
⋮
v_n
w_1
⋮
w_k


Shared closure
code f
code g
closure for f
closure for g
v_1
⋮
v_n
w_1
⋮
w_k

Part 2

Abstract machines

Three execution models

- Interpretation:
  control (the sequencing of computations) is represented by a source-level, tree-shaped term. The interpreter walks this tree at run-time.

- Native compilation:
  control is compiled down to a sequence of machine instructions. These instructions are those of a real processor, and are executed by hardware.

- Compilation to abstract machine code:
  control is compiled down to a sequence of abstract instructions. These instructions are those of an abstract machine: they do not match existing hardware, but are chosen to match closely the operations of the source language.

An abstract machine for arithmetic expressions

Arithmetic expressions: $a ::= cst \mid op(a_1, \ldots, a_n)$

The machine uses a stack to hold intermediate results.

Compilation is translation to "reverse Polish notation".

$$\begin{array}{rcl} \mathcal{C}(cst) & = & \mathtt{CONST}(cst) \\ \mathcal{C}(op(a_1, \ldots, a_n)) & = & \mathcal{C}(a_1); \mathcal{C}(a_2); \ldots; \mathcal{C}(a_n); I(op) \end{array}$$

where $I(+) = \mathtt{ADD}$, $I(-) = \mathtt{SUB}$, etc.

Transitions of the abstract machine

The machine has two components:

- a code pointer c giving the next instruction to execute;
- a stack s holding intermediate results.

Notation for stacks: top of stack is on the left

push v on s: $s \longrightarrow v.s$ pop v off s: $v.s \longrightarrow s$

Transitions of the machine:

State before		State after	
Code	Stack	Code	Stack
$\texttt{CONST}(cst); c$	s	c	$cst.s$
$\texttt{ADD}; c$	$n_2.n_1.s$	c	$(n_1 + n_2).s$
$\texttt{SUB}; c$	$n_2.n_1.s$	c	$(n_1 - n_2).s$

Halting state: code $= \varepsilon$ and stack $= v.s$.
Result of computation is v.

An abstract machine for call-by-value

(Similar to Landin's SECD and Cardelli's FAM.)

Three components in this machine:

- a code pointer c giving the next instruction to execute;
- an environment e giving values to free variables;
- a stack s holding intermediate results and return frames.

Compilation scheme:

$$
\begin{aligned}
\mathcal{C}(n) &= \mathtt{ACCESS}(n) \\
\mathcal{C}(\lambda a) &= \mathtt{CLOSURE}(\mathcal{C}(a); \mathtt{RETURN}) \\
\mathcal{C}(a\ b) &= \mathcal{C}(a); \mathcal{C}(b); \mathtt{APPLY}
\end{aligned}
$$

(Constants and arithmetic: as before.)

Transitions

State before			State after		
Code	Env	Stack	Code	Env	Stack
$\texttt{ACCESS}(n); c$	e	s	c	e	$e(n).s$
$\texttt{CLOSURE}(c'); c$	e	s	c	e	$[c', e].s$
$\texttt{APPLY}; c$	e	$v.[c', e'].s$	c'	$v.e'$	$c.e.s$
$\texttt{RETURN}; c$	e	$v.c'.e'.s$	c'	e'	$v.s$

- $\texttt{ACCESS}(n)$: push value of variable number n
- $\texttt{CLOSURE}(c')$: push closure $[c, e]$ of c with current env.
- $\texttt{APPLY}$: pop argument, pop closure, push return frame, jump to closure code.
- $\texttt{RETURN}$: restore saved code and environment from stack.

Executing abstract machine code

Code for a stack-based abstract machine can be executed either

- By expansion of abstract machine instructions to real machine instructions, e.g.

```
CONST(i)        --->     pushl $i

ADD             --->     popl %eax
                         addl 0(%esp), %eax
```

- By efficient interpretation.
  The interpreter is typically written in C as shown on the next slide, and is about one order of magnitude faster than term-level interpretation.

A typical abstract machine interpreter

```
value interpret(int * start_code)
{
  register int * c = start_code;
  register value * s = bottom_of_stack;
  register environment e;

  while(1) {
    switch (*c++) {
      case CONST:   *s++ = *c++; break;
      case ADD:     s[-2] = s[-2] + s[-1]; s--; break;
      case ACCESS:  *s++ = Lookup(e, *c++); break;
      case CLOSURE: *s++ = MakeClosure(*c++, e); break;
      case APPLY:   arg = *--sp; clos = *--sp;
                    *sp++ = (value) c; *sp++ = (value) e;
                    c = Code(clos);
                    e = AddEnv(arg, Environment(clos));
                    break;
      case RETURN:  res = *--sp; e = (environment) *--sp; c = (int *) *--sp;
                    *sp++ = res; break;
      case STOP:    return *--sp;
    }
  }
}
```

An abstract machine for call-by-name: Krivine's machine

As before, three components in this machine:
code c, environment e, stack s.

However, the stack does not contain values, but thunks:
closures $[c, e]$ representing expressions whose evaluation is delayed until needed.

Compilation scheme:

$$
\begin{aligned}
\mathcal{C}(n) &= \mathtt{ACCESS}(n) \\
\mathcal{C}(\lambda a) &= \mathtt{GRAB}; \mathcal{C}(a) \\
\mathcal{C}(a\ b) &= \mathtt{PUSH}(\mathcal{C}(b)); \mathcal{C}(a)
\end{aligned}
$$

Transitions of Krivine's machine

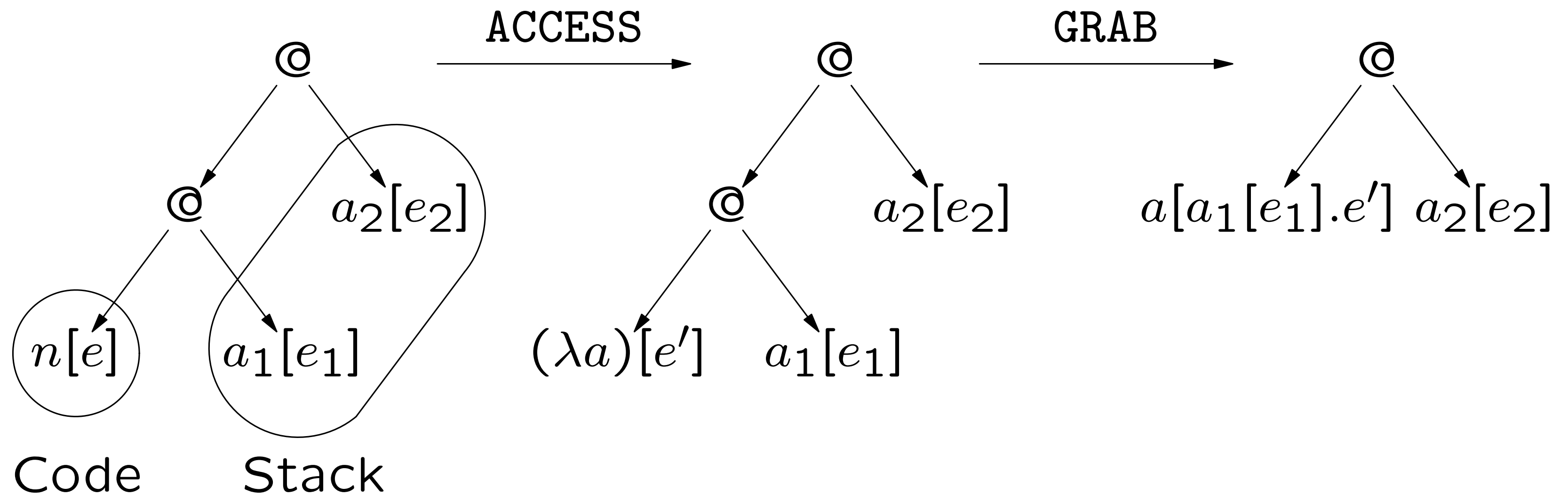
State before			State after			
Code	Env	Stack	Code	Env	Stack	
$\texttt{ACCESS}(n); c$	e	s	c'	e'	s	if $e(n) = [c', e']$
$\texttt{GRAB}; c$	e	$[c', e'].s$	c	$[c', e'].e$	s	
$\texttt{PUSH}(c'); c$	e	s	c	e	$[c', e].s$	

- $\texttt{ACCESS}(n)$: fetch thunk bound to variable number n, and proceed evaluating it
- $\texttt{GRAB}$: pop the next argument provided off the stack, and add it to the environment (β-reduction step)
- $\texttt{PUSH}(c)$: build thunk for c and push it

Why does it work?

The stack maintains the current spine of applications.

The code is the leftmost outermost part of the spine.



Making call-by-name practical

Realistic abstract machines for call-by-name functional languages are more complex than Krivine's machine in two aspects:

- **Constants and strict primitive operations:**

  Operators such as integer addition are strict in their arguments. Extra machinery is required to reduce sub-expressions to head normal form in a strict way.

- **Sharing of evaluations (lazy evaluation):**

  Call-by-name reduces an expression every time its head normal form is needed. Lazy evaluation performs the reduction the first time it is needed, then caches the result for further reference.

See e.g. *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*, S.L. Peyton Jones, Journal of Functional Programming 2(2), Apr 1992.

Proving the correctness of an abstract machine

At this point, we have two notions of evaluation for terms:

1. Source-level evaluation with environments:
   $a[e] \xrightarrow{*} v$ or $e \vdash a \Rightarrow v$.

2. Compilation, then execution by the abstract machine:

$$\begin{pmatrix} c = \mathcal{C}(a) \\ e = \varepsilon \\ s = \varepsilon \end{pmatrix} \xrightarrow{*} \begin{pmatrix} c = \varepsilon \\ e = \ldots \\ s = v \ldots \end{pmatrix}$$

Do these two notions agree? Does the abstract machine compute the right results?

Partial correctness w.r.t. the big-step semantics

The compilation scheme is compositional: each sub-term is compiled to code that evaluates it and leaves its value on the top of the stack.

This parallels exactly a derivation of $e \vdash a \Rightarrow v$ in the big-step semantics, which contains sub-derivations $e' \vdash a' \Rightarrow v'$ for each sub-expression a'.

Theorem: if $e \vdash a \Rightarrow v$, then

$$\begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{pmatrix} \quad \xrightarrow{*} \quad \begin{pmatrix} k \\ \mathcal{C}(e) \\ \mathcal{C}(v).s \end{pmatrix}$$

The compilation scheme $\mathcal{C}(\cdot)$ is extended to values and environments as follows:

$$
\begin{aligned}
\mathcal{C}(cst) &= cst \\
\mathcal{C}((\lambda a)[e]) &= [(\mathcal{C}(a); \texttt{RETURN}), \mathcal{C}(e)] \\
\mathcal{C}(v_1 \ldots v_n.\varepsilon) &= \mathcal{C}(v_1) \ldots \mathcal{C}(v_n).\varepsilon
\end{aligned}
$$

The theorem is proved by induction on the derivation of $e \vdash a \Rightarrow v$. We show the most interesting case: function application.

$$
\frac{e \vdash a \Rightarrow (\lambda c)[e'] \qquad e \vdash b \Rightarrow v' \qquad v'.e' \vdash c \Rightarrow v}{e \vdash a\ b \Rightarrow v}
$$

$$(\; \mathcal{C}(a);\mathcal{C}(b);\texttt{APPLY};k \;\mid\; \mathcal{C}(e) \;\mid\; s \;)$$

$\downarrow *$ ind. hyp. on first premise

$$(\; \mathcal{C}(b);\texttt{APPLY};k \;\mid\; \mathcal{C}(e) \;\mid\; [(\mathcal{C}(c);\texttt{RETURN}),\mathcal{C}(e')].s \;)$$

$\downarrow *$ ind. hyp. on second premise

$$(\; \texttt{APPLY};k \;\mid\; \mathcal{C}(e) \;\mid\; \mathcal{C}(v').[(\mathcal{C}(c);\texttt{RETURN}),\mathcal{C}(e')].s \;)$$

$\downarrow$ APPLY transition

$$(\; \mathcal{C}(c);\texttt{RETURN} \;\mid\; \mathcal{C}(v'.e') \;\mid\; k.\mathcal{C}(e).s \;)$$

$\downarrow *$ ind. hyp. on third premise

$$(\; \texttt{RETURN} \;\mid\; \mathcal{C}(v'.e') \;\mid\; \mathcal{C}(v).k.\mathcal{C}(e).s \;)$$

$\downarrow$ RETURN transition

$$(\; k \;\mid\; \mathcal{C}(e) \;\mid\; \mathcal{C}(v).s \;)$$

Towards full correctness

The previous theorem shows the correctness of the abstract machine for terminating terms. However, if the term a does not terminate, $e \vdash a \Rightarrow v$ does not hold, and we do not know anything about the execution of the compiled code.

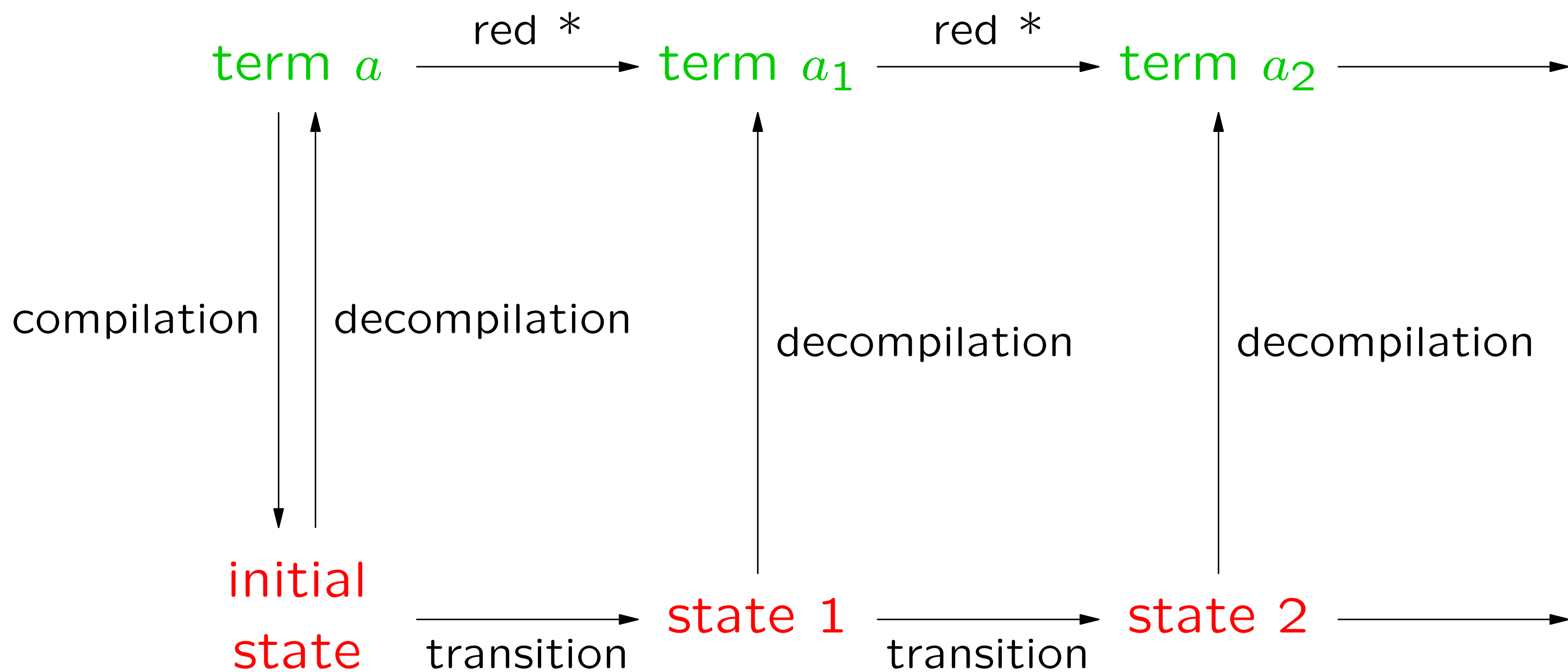
(It might loop, but it might just as well stop and reply "42".)

To show correctness for all terms (terminating or not), we need to establish a simulation between machine transitions and source-level reductions:

> Each transition of the machine corresponds to zero, one, or several source-level reductions.

See *Functional Runtimes within the Lambda-Sigma Calculus*, T. Hardin, L. Maranget, B. Pagano, *J. Func. Prog* 8(2), 1998.

The simulation



Problem: not all intermediate states of the machine correspond to the compilation of a source term.

Solution: define a decompilation function $\mathcal{D}$: States $\rightarrow$ Terms that is defined on all intermediate states and is left inverse of the compilation function.

The decompilation function

Idea: decompilation is a symbolic variant of the abstract machine: it reconstructs source terms rather than performing the computations.

Decompilation of values:

$$\mathcal{D}(cst) = cst \qquad \mathcal{D}([c, e]) = (\lambda a)[\mathcal{D}(e)] \text{ if } c = \mathcal{C}(a); \mathtt{RETURN}$$

Decompilation of environments and stacks:

$$\mathcal{D}(v_1 \dots v_n.\varepsilon) = \mathcal{D}(v_1) \dots \mathcal{D}(v_n).\varepsilon$$
$$\mathcal{D}(\dots v \dots c.e \dots) = \dots \mathcal{D}(v) \dots c.\mathcal{D}(e) \dots$$

Decompilation of concrete states:

$$\mathcal{D}(c \mid e \mid s) = \mathcal{D}(c \mid \mathcal{D}(e) \mid \mathcal{D}(s))$$

Decompilation, continued

Decompilation of abstract states (E, S already decompiled):

$$
\begin{array}{rcl}
\mathcal{D}(\varepsilon \mid E \mid a.S) & = & a \\
\mathcal{D}(\mathtt{CONST}(cst); c \mid E \mid S) & = & \mathcal{D}(c \mid E \mid cst.S) \\
\mathcal{D}(\mathtt{ACCESS}(n); c \mid E \mid S) & = & \mathcal{D}(c \mid E \mid E(n).S) \\
\mathcal{D}(\mathtt{CLOSURE}(c'); c \mid E \mid S) & = & \mathcal{D}(c \mid E \mid (\lambda a)[E].S) \\
 & & \text{if } c' = \mathcal{C}(a); \mathtt{RETURN} \\
\mathcal{D}(\mathtt{RETURN}; c \mid E \mid a.c'.E'.S) & = & \mathcal{D}(c' \mid E' \mid a.S) \\
\mathcal{D}(\mathtt{APPLY}; c \mid E \mid b.a.S) & = & \mathcal{D}(c \mid E \mid (a\ b).S) \\
\mathcal{D}(I(op); c \mid E \mid a_n \ldots a_1.S) & = & \mathcal{D}(c \mid E \mid (op(a_1, \ldots, a_n)).S)
\end{array}
$$

Correctness lemmas

Simulation: if $\mathcal{D}(S)$ is defined, and the machine performs a transition from S to S', then $\mathcal{D}(S')$ is defined and $\mathcal{D}(S) \xrightarrow{*} \mathcal{D}(S')$.

Progress: if S is not a final state, and $\mathcal{D}(S)$ is defined and reduces, then the machine can perform a transition from S.

No stuttering: there exists a non-negative measure $|S|$ on machine states such that if the machine does a silent transition from S to S' (i.e. $\mathcal{D}(S) = \mathcal{D}(S')$), then $|S| > |S'|$.

Initial states: $\mathcal{D}(\mathcal{C}(a) \mid \varepsilon \mid \varepsilon) = a$ if a is closed.

Final states: $\mathcal{D}(\varepsilon \mid e \mid v.s) = \mathcal{D}(v)$.

Correctness theorem

Theorem: Let a be a closed term and $S = (\mathcal{C}(a) \mid \varepsilon \mid \varepsilon)$.

- If $a \xrightarrow{*} v$, then the abstract machine started in state S terminates and returns the value $\mathcal{C}(v)$.
- If a reduces infinitely, the machine started in state S performs an infinite number of transitions.

Part 3

Optimized compilation of functions: flow analysis

Compilation to optimized machine code

Step 1: Replace functions by closures; make explicit the construction, passing, and accessing of the environments.

This brings us to a conventional intermediate language that manipulates code pointers (i.e. closed functions).

(Think of the intermediate language as a subset of C with support for dynamic allocation and garbage collection.)

Step 2: Optimize and generate machine code from the intermediate language.

- Write your own code generator using conventional compiler technology (OCaml, SML/NJ)
- Or use a C compiler (GHC, Bigloo).

Replacement of functions by closures

$[\![$x$]\!]$ = x $\qquad$ $[\![$cst$]\!]$ = cst

$[\![$op(a$_1$,...,a$_n$)$]\!]$ = op($[\![$a$_1$$]\!]$,...,$[\![a_n$$]\!]$)

$[\![$a b$]\!]$ = let clos = $[\![$a$]\!]$ in clos[0] (clos, $[\![$b$]\!]$)

$[\![$$\lambda$x. a$]\!]$ =

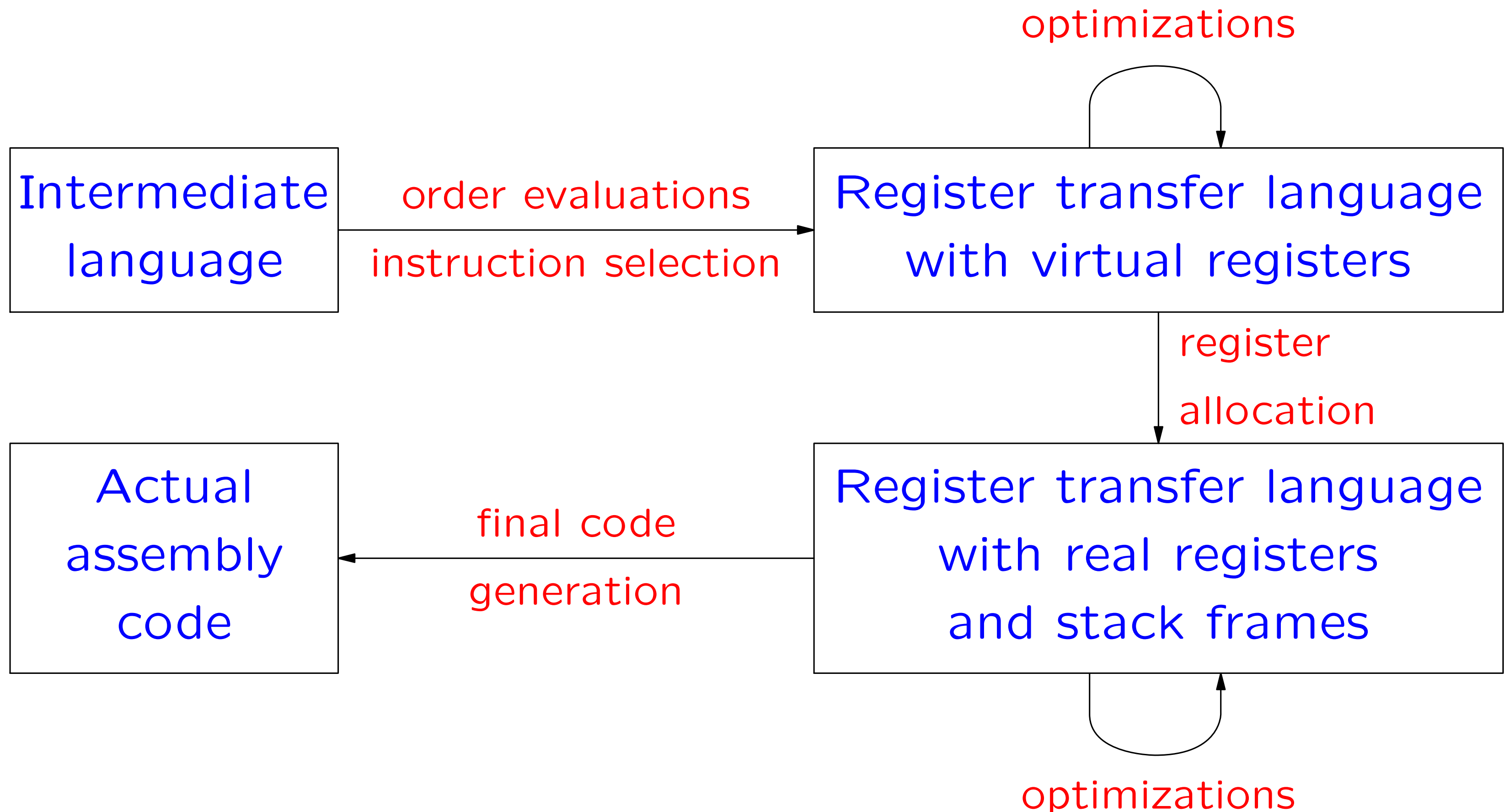
```
    let code_fn (clos, x) =
      let v1 = clos[1] and ... and vn = clos[n] in [[a]]
    in
      makeblock(code_fn, v1, ..., vn)
```

where $v_1,\ldots,v_n$ are the variables free in $\lambda x.a$.

Note: the function `code_fn` above has no free variables. The occurrence of `code_fn` in `makeblock(code_fn, \ldots)` denotes a pointer to its (fixed) code.

From intermediate language to machine code

Standard compiler technology applies largely unchanged.



See any good compiler textbook, e.g. *Modern Compiler Implementation in ML*, Andrew W. Appel, Cambridge Univ. Press, 1998.

The overhead of closure invocation

```
⟦a b⟧ = let clos = ⟦a⟧ in clos[0] (clos, ⟦b⟧)
```

Our translation scheme transforms every function application into

- one load `clos[0]`
- one call to a computed address `clos[0](...)`

Calls to a computed address are expensive on modern processors:

- the destination address is usually not predicted in advance;
- this stalls the pipeline (cannot fetch and start executing instructions from the called function while earlier instructions complete).

Typically, a factor of 10 more expensive than a call to a statically known address.

Opportunities for generating static calls

The overhead of calls to computed addresses can be avoided in many practical situations:

```
let succ = λx. x + 1 in succ (succ 2)
```

The two applications of `succ` "obviously" call the code for $\lambda x.x + 1$, and no other code.

```
let rec f = λx. ... f arg ...
```

The application of `f` always calls the code for the current function.

```
let sort_list = λordering. λlist. ... in
... sort_list (λx.λy. compare(x,y)) some_list ...
```

If there are no other calls to `Sort.list` in the program, all applications of `ordering` in the body of `sort_list` call the code for $\lambda x.\lambda y.\mathrm{compare}(x, y)$.

Opportunities for generating static calls

In all of these cases:

- application of a function in the static scope of its definition
- recursive calls
- higher-order functions applied only once

we could (and should):

- Generate calls to statically-known code addresses.

- Or if the called function is small, perform inline expansion (compile-time β-reduction) of its body, e.g.

`succ (succ 2)` $\Rightarrow$ `(2 + 1) + 1`

Control-flow analyses (CFA)

A program analysis is needed to discover those opportunities for closure optimization.

Control-flow analyses (Shivers, PLDI 1988) approximate at each application point the set of functions that can be called here (in other terms, the set of function values that can flow to this application point).

If that set is a singleton $\{\lambda x.a\}$, we can generate a direct call to the code for a, or inline it if a is small enough.

In all cases, we also get an approximation of the call graph for the program (who calls who?), required for later interprocedural optimizations (e.g. global register allocation).

A high-level view of CFA

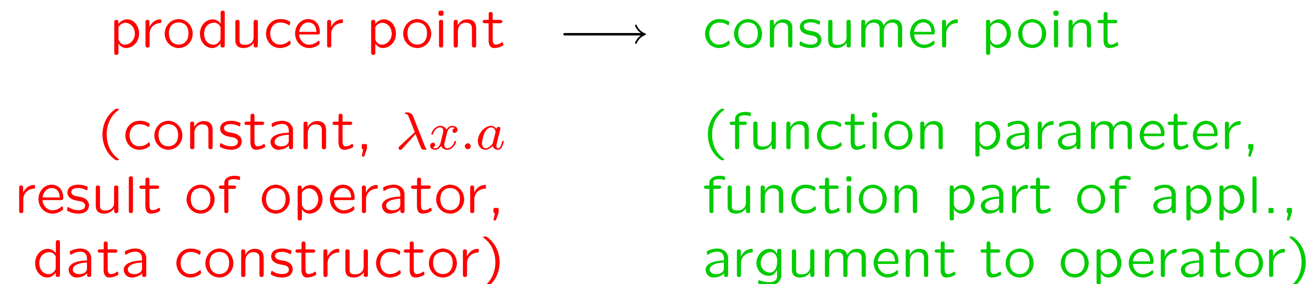
Since functions are first-class values, CFA is actually a data-flow analysis that keeps track of the flow of functional values and determines control-flow along the way.

CFA sets up a system of constraints of the form

$$V(\ell_1) \subseteq V(\ell_2)$$

meaning that all values at program point ℓ_1 can flow to point ℓ_2.

Solve that system into a flow graph:



Example of constraint generation rules

For `(if a`^{m} `then b`^{n} `else c`^{p}`)`$^\ell$: add the constraints

$$V(n) \subseteq V(\ell) \quad \text{(the then branch flows to the result)}$$
$$V(p) \subseteq V(\ell) \quad \text{(the else branch flows to the result)}$$

For `(let x = a`^{m} `in b`^{n}`)`$^\ell$: add

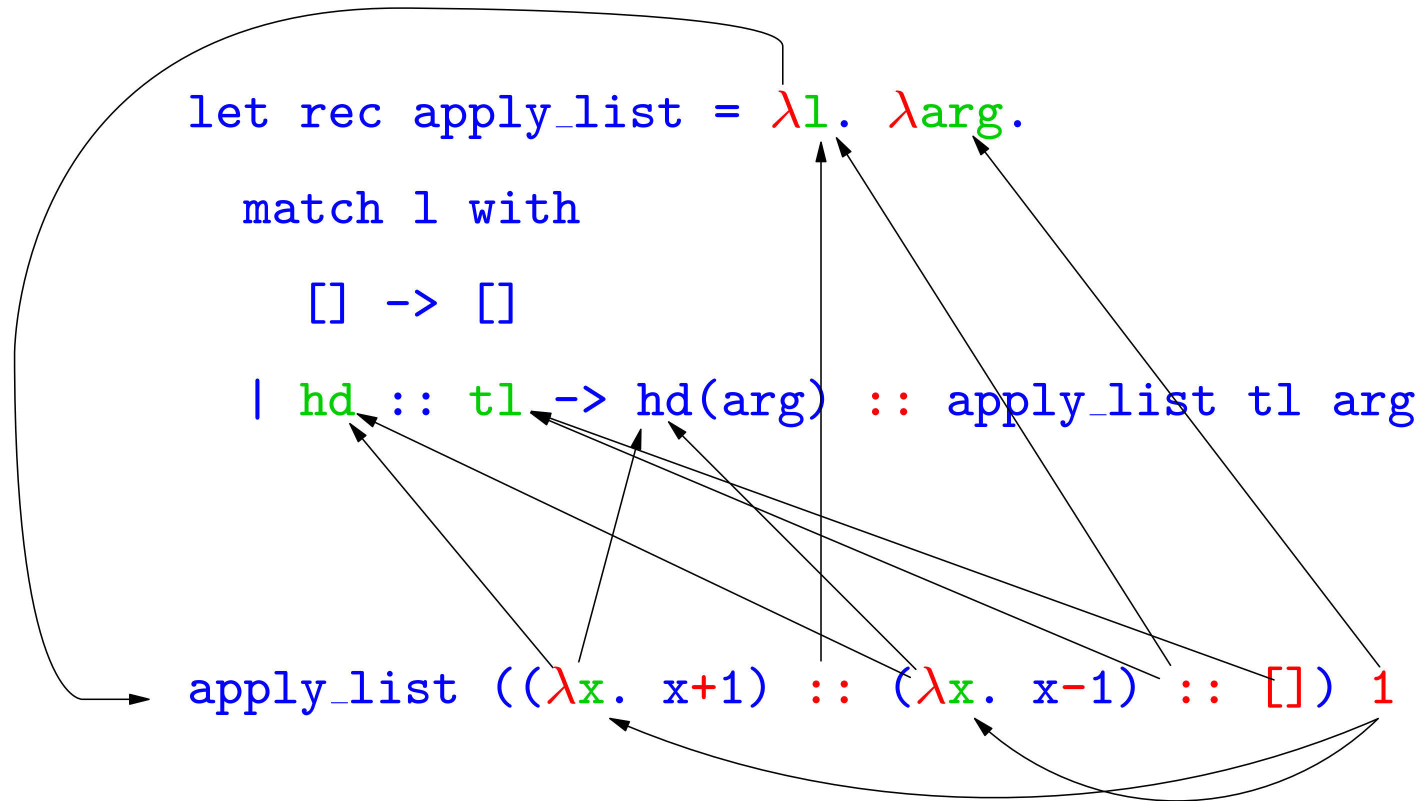
$$V(m) \subseteq V(x) \qquad\qquad V(n) \subseteq V(\ell)$$

For `(a`^{m}`(b`^{n}`))`$^\ell$:
for each function $\lambda x.\ c^q$ in $V(m)$, add the constraints

$$V(n) \subseteq V(x) \quad \text{(the argument flows to the parameter)}$$
$$V(q) \subseteq V(\ell) \quad \text{(the function result flows to the application result)}$$

Note: need to interlace constraint building and constraint solving, and iterate till fixpoint is reached.

An example of CFA



(Not all edges are shown.)

We have determined the two functions called from `apply_list`, and also that they are always applied to 1.

Summary on CFA

Basic algorithm (0-CFA) is $O(n^3)$ (n is the size of the program).

Main applications:

- Optimize function calls in functional languages.

  (As seen before.)

- Optimize method dispatch in object-oriented languages.

  (Similar problems to that of closures. Roughly, an object is a closure with multiple entry points.)

- Eliminate run-time type tests in dynamically-typed languages such as Scheme.

  (E.g. if all values flowing to + are integers, + does not need to check the type of its arguments.)

- More applications later...

Variants of 0-CFA

More precise analyses:

- Polyvariant analyses (n-CFA, polymorphic splitting, ...): distinguish between different call sites of the same function.
- Finer approximation of values (Heintze's set-based analysis): capture the shapes of data structures using grammars.

Less precise (faster) analyses:

- Coarser representations of sets of values: $\emptyset$ or $\{v\}$ (singletons) or $\top$ (all values).
- Do not iterate till fixpoint: (Ashley, ICFP 1997) start with $\top$ on all variables and do 1 or 2 iterations.
- Use equality constraints (unification) in addition to inclusion constraints.

Digression on inlining: why not compile-time β-reductions?

An analysis like CFA might seem overkill for function inlining: the effect of inlining can also be achieved by β-reductions at compile-time.

$$
\begin{array}{l}
\texttt{let succ = }\lambda\texttt{x. x + 1 in succ (succ 2)} \\
\xrightarrow{\beta} \texttt{(}\lambda\texttt{x. x + 1) ((}\lambda\texttt{x. x + 1) 2)} \\
\xrightarrow{\beta} \texttt{(}\lambda\texttt{x. x + 1) (2 + 1)} \\
\xrightarrow{\beta} \texttt{(2 + 1) + 1}
\end{array}
$$

Unrestricted compile-time reductions may cause program size explosion (or even execution of the whole program at compile-time!); various size-based heuristics control when to perform them.

The hardness of compile-time β-reductions

Problem 1: these heuristics are difficult to calibrate.

$\texttt{let p = } (\lambda\texttt{x.}\textit{small}\texttt{), } (\lambda\texttt{y.}\textit{HUGE}\texttt{) in fst(p)(1)}$

$\xrightarrow{\beta}$ $\texttt{fst((}\lambda\texttt{x.}\textit{small}\texttt{), (}\lambda\texttt{y.}\textit{HUGE}\texttt{))(1)}$

$\xrightarrow{\beta}$ $\texttt{(}\lambda\texttt{x.}\textit{small}\texttt{)(1)}$ $\xrightarrow{\beta}$ $\textit{small}\{\texttt{x}\leftarrow 1\}$

An intermediate reduct can be huge, then collapse to a much smaller term; shall the compiler lose the opportunity for inlining, or risk explosion?

Problem 2: β-reduction is not selective enough

$\texttt{let p = } (\lambda\texttt{x.}\textit{small}\texttt{), } (\lambda\texttt{y.}\textit{HUGE}\texttt{) in}$
$\texttt{fst(p)(1), snd(p)(2), snd(p)(3)}$

$\xrightarrow{\beta *}$ $\textit{small}\{\texttt{x}\leftarrow 1\}\texttt{, }\textit{HUGE}\{\texttt{y}\leftarrow 2\}\texttt{, }\textit{HUGE}\{\texttt{y}\leftarrow 3\}$

There is no way to inline *small* without duplicating *HUGE*.

CFA avoids both problems by simulating what β-reduction would do without actually performing it.

Connections between CFA and type systems

CFA can be used as a type system if enriched with safety checks (e.g. fail if an integer flows to an application site).

Conversely, many type systems (and type inference algorithms) can be viewed as checking / approximating the flow of data in a program.

Palsberg and O'Keefe (TOPLAS 1995) show equivalence between:

- 0-CFA with safety checks;
- the Amadio-Cardelli type system (subtyping + recursive types).

Provides an efficient type inference algorithm for that system.

Also: type inference algorithms for type systems with subtyping are based on inclusion constraints similar to those used by CFA. (Aiken and Wimmers, FPCA 1993; Smith et al, MFPS 1995; Pottier, ICFP 1998.)

Part 4

Optimizing the representation of data

Representations for high-level data structures

High-level data structures (such as ML's datatypes) leave considerable flexibility to the compiler in deciding a data representation.
$\longrightarrow$ clever representation tricks are feasible.

(Would be hard to do by hand in C, at least portably.)

Examples:

- For dynamically-typed languages (Scheme): clever tagging scheme (to embed the type of an object in its bit pattern).

- For ML's datatypes: clever encodings of the constructor.

Example: representation of datatypes in Objective Caml

```
type expr = Const of int | Var
          | Sum of expr * expr | Prod of expr * expr
```

Constant constructors are represented by odd integers 1, 3, ...
(Bit pattern: $\ldots xxx1$)

Constructors with arguments are represented by word-aligned pointers to heap blocks.
(Bit pattern: $\ldots xx00$)

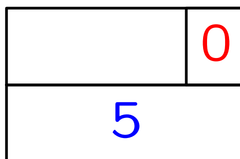
The heap block contains one byte (the “tag” byte) representing the number of the constructor.

(This byte is stored at no extra cost in the header word required by the garbage collector.)

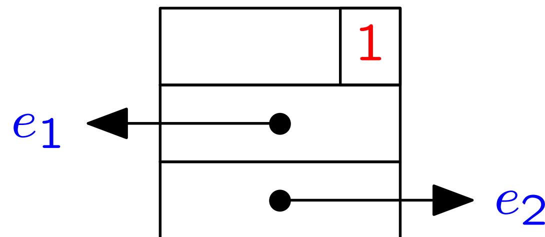
Example of datatype representation

```
type expr =
    Const of int              (* pointer to block with tag 0 *)
  | Var                       (* integer 1 *)
  | Sum of expr * expr        (* pointer to block with tag 1 *)
  | Prod of expr * expr       (* pointer to block with tag 2 *)
```

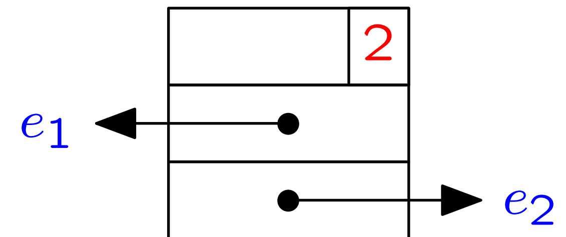
Const(5)



Sum(e_1, e_2)



Prod(e_1, e_2)



Data representation and static typing

Without static typing (Scheme):

- Need tagging to implement run-time type tests.
- All data types must fit a common format (usually one word).
  $\longrightarrow$ floats are boxed (heap-allocated);
  $\longrightarrow$ records are boxed;
  $\longrightarrow$ arrays are arrays of pointers to boxed elements.
- All functions must use the same calling conventions: e.g. argument in `R0`; result in `R0`.

With monomorphic static typing (Pascal, C)

- No need to support run-time type tests.

- Different data types can have different sizes.
  $\rightarrow$ unboxed floats
  $\rightarrow$ unboxed records (if small enough)
  $\rightarrow$ flat arrays

  The compiler determines the size from the static type:

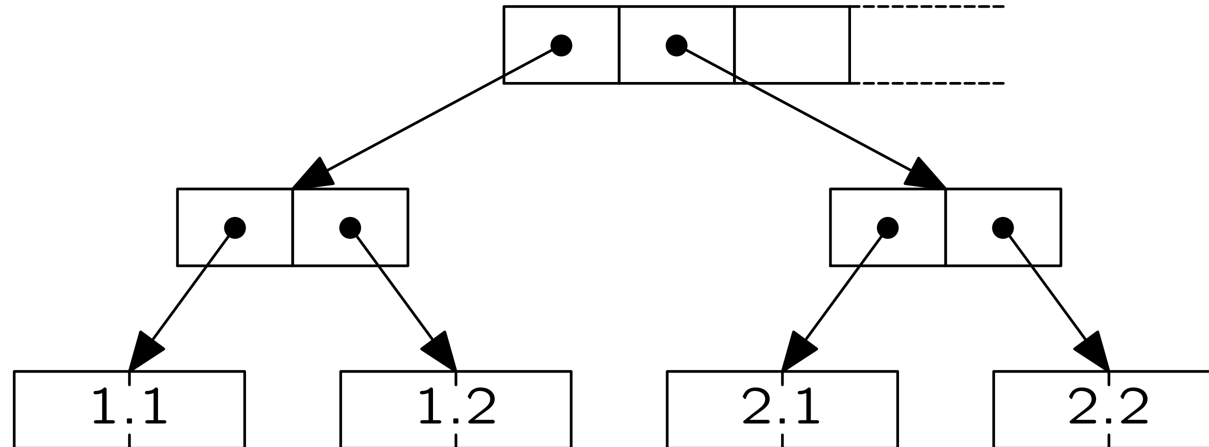
  $|\texttt{int}| = 1$ word $\qquad |\texttt{float}| = 2$ words $\qquad |\tau \times \sigma| = |\tau| + |\sigma|$

- Functions of different types can use different calling conventions. E.g. use floating-point registers for float arguments and results.

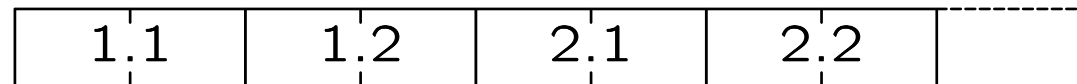
$\texttt{float} \rightarrow \texttt{float}$	argument in `FP0`, result in `FP0`
$\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$	argument in `R0` and `R1`, result in `R0`

Example: an array of 2D points

In Scheme:



In C:



The problem with polymorphic typing

The type system guarantees type safety, but does not assign a unique type to every value at compile-time:

Polymorphism:

`fun x -> x` : $\forall\alpha.\ \alpha \rightarrow \alpha$

Actual type of `x`: any

Size of `x`: variable

Calling conventions: variable

Type abstraction:

```
type t
val x : t
val f : t -> t
```

Actual type of `x`: unknown

Size of `x`: unknown

Calling conventions: unknown

Simple solutions

- Restrict polymorphism and type abstraction.

  Modula: abstract types must be pointer types.
  Java: cannot coerce integers and floats to/from type `Object`.

  Problem: unnatural.

- Code replication.

  Ada, C++: compile a specialized version of a generic function for each type it is used with.

  Problem: code size explosion; link-time code generation.

- Revert to Scheme-style representations.

  Problem: inefficient; lots of boxing and unboxing.

More interesting solutions

- Use run-time type inspection:
  pass type information at run-time to polymorphic code;
  use this information to determine sizes and layouts at run-time.

- Mix C-style representations for monomorphic code and Scheme-style representations for polymorphic code.

- Combine Scheme-style representations with local unboxing, partial inlining, and special treatment of arrays.

The type-passing interpretation of polymorphism

In order to reconstruct exact types of data structures at run-time, polymorphic function must receive as extra arguments the types to which they are specialized.

```
let f x = x                    let f α x = x

let g x = f (x, x)             let g β x = f ⟨β × β⟩ (x, x)

g 5                            g ⟨int⟩ 5
```

In this example, this allows `f` to determine at run-time that its `x` parameter has actual type $\texttt{int} \times \texttt{int}$.

Type-dependent data layout

The TIL approach (Harper, Morrisett, et al, PLDI 1996):

- Use C-style, “flat”, multi-word representations of data structures (just like in a monomorphic type system).
- In polymorphic code, compute size information, data layout, and calling conventions from the run-time type information.
- (In monomorphic code, this information is computed at compile-time.)

Example

Source code:

```
let assign_array a b i = b.(i) <- a.(i)
```

Generated code, Scheme style:

```
assign_array(a, b, i) {
  load one word from a + i * 4;
  store this word at b + i * 4;
}
```

Generated code, TIL style:

```
assign_array(α, a, b, i) {
    s = size_of_type(α);
    copy s bytes
    from a + i * s
    to b + i * s;
}
```

Variant (Ohori, Lisp.Symb.Comp.1993): pass only the size of types at run-time, not representations of whole type expressions.

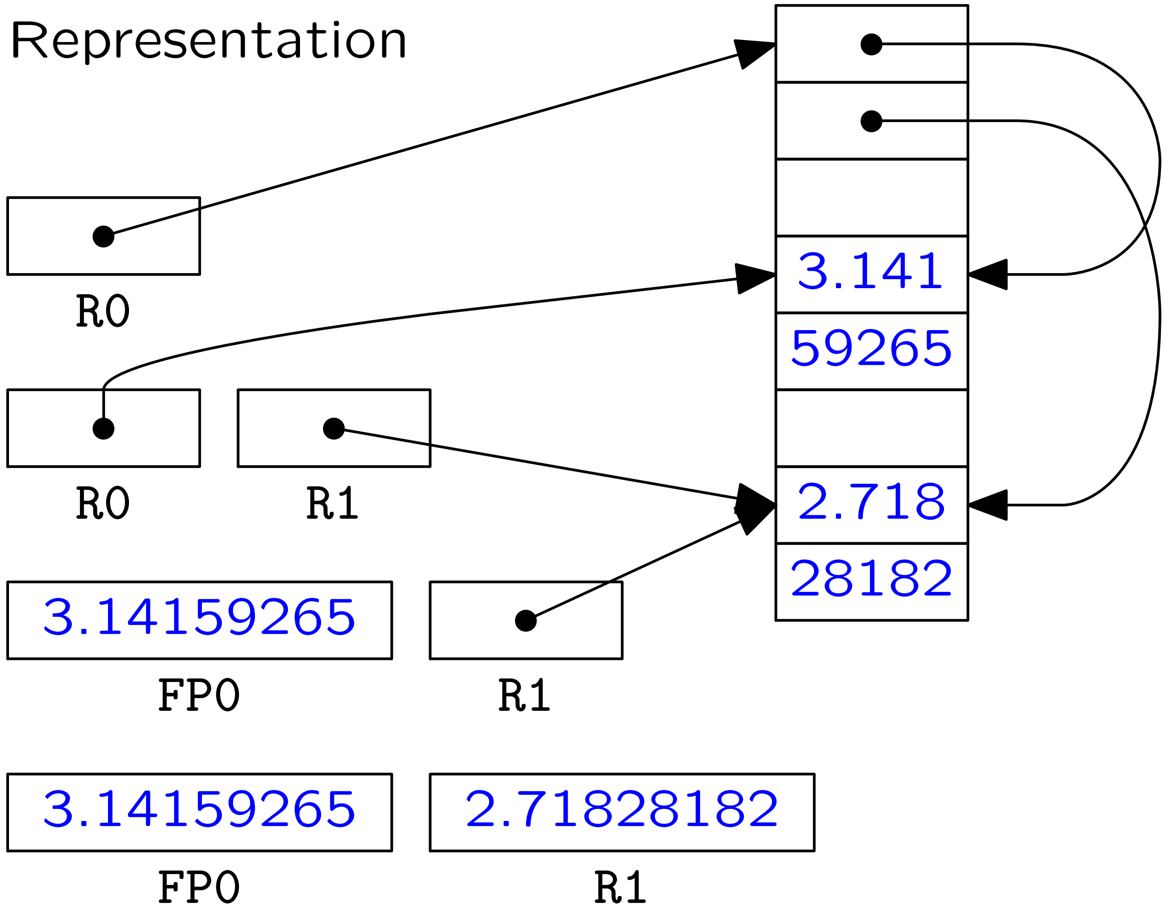
Mixed data representations

(Leroy, POPL 1992; Shao and Appel, PLDI 1995; the SML/NJ compiler)

Use C-style representations for data whose exact type is known at compile-time (i.e. inside monomorphic code).

Revert to Scheme-style representations for manipulating data whose type is not completely known at compile-time (i.e. inside polymorphic code).

Insert coercions between the two representations at interface points.

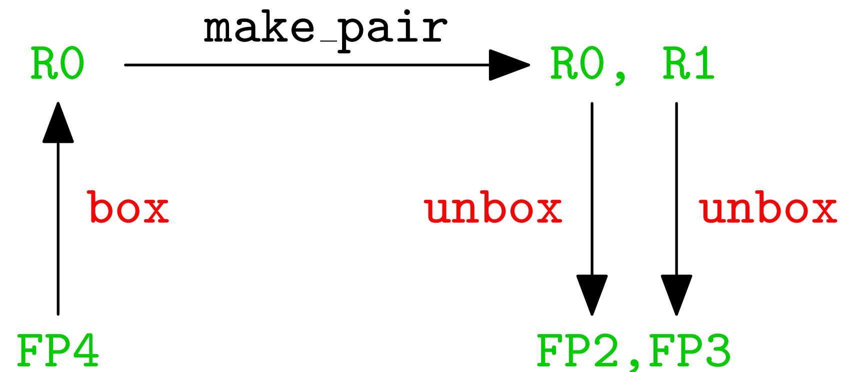


Static type
Representation
α
R0
$\beta \times \gamma$
R0
R1
float $\times$ γ
3.14159265
FP0
R1
float $\times$ float
3.14159265
2.71828182
FP0
R1
3.141
59265
2.718
28182

Source code:

```
let make_pair x = (x, x) in ... make_pair 3.41519
```

Coercion diagram:



Generated code:

```
let make_pair x = (x, x) in ...
let (fst, snd) = make_pair(box_float(3.14159)) in
  (unbox_float(fst), unbox_float(snd))
```

Defining the coercions

$$
\begin{array}{rclcrcl}
[\alpha \Rightarrow \texttt{int}] & = & \texttt{identity} & \quad & [\texttt{int} \Rightarrow \alpha] & = & \texttt{identity} \\
[\alpha \Rightarrow \texttt{float}] & = & \texttt{unbox_float} & & [\texttt{float} \Rightarrow \alpha] & = & \texttt{box_float} \\
[\alpha \Rightarrow \beta \times \gamma] & = & \texttt{unbox_pair} & & [\beta \times \gamma \Rightarrow \alpha] & = & \texttt{box_pair}
\end{array}
$$

$$
\begin{array}{rcl}
[(\tau \times \sigma) \Rightarrow (\tau' \times \sigma')] & = & \lambda(x, y).\ ([\tau \Rightarrow \tau'](x), [\sigma \Rightarrow \sigma'](y)) \\
[(\tau \rightarrow \sigma) \Rightarrow (\tau' \rightarrow \sigma')] & = & \lambda f.\ [\sigma \Rightarrow \sigma'] \circ f \circ [\tau' \Rightarrow \tau]
\end{array}
$$

When using a value v of type $\forall\alpha.\tau$ with type $\tau' = \tau\{\alpha \leftarrow \sigma\}$, insert the coercion $[\tau \Rightarrow \tau'](v)$.

When implementing an abstract type $\exists\alpha.\tau$ by a value v of type $\tau' = \tau\{\alpha \leftarrow \sigma\}$, insert the coercion $[\tau' \Rightarrow \tau](v)$.

Untyped unboxing techniques

(Objective Caml; Glasgow Haskell; Bigloo Scheme.)

Instead of basing the data representations on the types, use Scheme-style representations by default, plus:

- Perform intra-function unboxing by standard dataflow analysis:

```
let x = box(f) in                        ⇒      let x = f in
  ... unbox(x) ... unbox(x) ...                   ... x ... x ...
```

- Extend it to inter-function unboxing using control-flow analysis or partial inlining.

- Use simple tagging schemes and tag testing to support important special cases of generic data structures (e.g. float arrays).

Partial inlining (a.k.a. the worker-wrapper technique)

(Peyton-Jones and Lauchbury, FPCA 1991; Goubault, SAS 1994.)

Split a function into:

- a worker function taking and returning unboxed data;
- a wrapper function performing the boxing and unboxing around the worker.

At call sites, try to inline the wrapper function (typically small) and hope its boxing and unboxing cancel out with those of the call context.

Example

```
let worker_f a b =
  (* a and b are unboxed floats *)
  (* compute result *)
  (* return unboxed float result *)

let f a b = box(worker_f (unbox a) (unbox b))

... unbox(f (box 3.14) (box 2.71)) ...
```

After inlining of `f` and simplifications:

```
... worker_f 3.14 2.71 ...
```

(Crucially depends on the availability of a good inlining pass in the compiler.)

Conclusions and perspectives

Other relevant topics not addressed here

Memory management and garbage collection.

Uniprocessor Garbage Collection Techniques, P. Wilson,
`ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`.

Garbage collection support in intermediate code and code generation.

The `C--` *intermediate language*, N. Ramsey and S. Peyton Jones.

Optimized compilation of pattern-matching.

Two techniques for Compiling Lazy Pattern Matching, L. Maranget, INRIA research report 2385.

Relevance and adaptation of classic compiler optimizations.

Automatic parallelization.

Conclusions from an engineering standpoint

Compilation technology for functional languages is relatively mature:

- On comparable programs, achieve at least 50% of the performance of optimizing C compilers.
- Match or exceed the performance of C on allocation-intensive programs.

Still more work to do:

- Getting rid of the last factor of 2 is difficult.
- Truly efficient functional programs still require programmers to be conscious of performance issues while writing their code.

Conclusions from a research standpoint

Functional languages promote software reliability:

clean semantics $\Rightarrow$ formal methods $\Rightarrow$ reliable programs

But: proving the correctness of source code is useless if the compiler is incorrect.

$\longrightarrow$ Certified compilers

$$\mathit{Proof} \vdash \forall \mathit{Prog}.\quad \mathit{Comp}(\mathit{Prog}) \equiv \mathit{Prog}$$

$\longrightarrow$ Certifying compilers (proof-carrying code)

$$\forall \mathit{Prog}.\quad \mathit{Certif}(\mathit{Prog}) \vdash \mathit{Comp}(\mathit{Prog}) \equiv \mathit{Prog}$$

Today: we are able to certify realistic bytecode compilers and abstract machines.

Tomorrow: certification of optimizing native-code compilers?