An introduction to compiling functional languages

Xavier Leroy

INRIA Rocquencourt, projet Cristal

`http://pauillac.inria.fr/~xleroy/`

Prologue

What is a functional language?

Functions as first-class values

A functional language is a language where the notion of function is taken seriously.

- Functions are first-class objects in the language: just like any other language object, they can be
  - passed as parameters to other functions;
  - returned as function results;
  - stored in data structures.
- Functions can be computed (in some sense).

Examples

Derivative of a floating-point function:

```
# let deriv f h =
    (fun x -> (f(x +. h) -. f(x -. h)) /. (2.0 *. h));;
val deriv = <fun>

# let d_sin = deriv sin 1e-5;;
val d_sin = <fun>

# d_sin 0.0;;
- = 0.999999999983
```

Main uses of functions as first-class values

- Facilitates reuse of generic functions.

  Example: the `List.map` list transformer:

  $$\texttt{List.map}\ f\ [e_1;\ \ldots;\ e_n]\ \texttt{=}\ [f(e_1);\ \ldots;\ f(e_n)]$$

- Supports user-defined control structures (e.g. iterators).

- Allows mixing data and functions in data structures

Non-functional languages

- In C:
  - code pointers are first-class values;
  - but functions cannot depend on parameters to other functions.
- In Pascal:
  - nested functions are supported;
  - but functions cannot be returned as a result, nor put in data structures.

Declarative vs. imperative

Functional languages promote a more abstract (more “mathematical”) view of programming than imperative languages.

Example: the `let` construct names an intermediate result.

No need to declare a variable, then assign it with the result.

```
let f x =
  let cosx = cos(x) in
  cosx *. cosx
```

```
function f(x: real): real
begin
  var cosx: real;
  cosx := cos(x);
  f := cosx *. cosx;
end
```

Declarative vs. imperative, 2

Data structures are allocated and initialized in one construct.

No need to allocate space for the structure, then fill in the fields.

```
type point =
  {x: float; y: float}

let scale pt s =
  {x = pt.x *. s;
   y = pt.y *. s}
```

```
typedef struct {double x, y;} *point;

point scale(point pt, double s) {
   point res = malloc(sizeof(*pt));
   res->x = pt->x * s;
   res->y = pt->y * s;
   return res;
}
```

Also:

- pointers are implicit;
- automatic memory management (garbage collection);
- impossible to have partially initialized objects.

Purely functional or not?

- "Pure" functional languages prohibit entirely assignment to variables and in-place modification to data structures.

  Example: Haskell.

  Goes along with lazy evaluation (on-demand computation of expressions).

- Other functional languages are equipped with full imperative power as well.

  Example: ML, Scheme.

  Still, these languages encourage declarative programming.

Datatypes and pattern-matching

Most functional languages have high-level data structures and pattern-matching to operate over these:

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

let expr = Prod(Sum(Var, Const 1), Var)

let rec deriv e =
  match e with
    Const n -> Const 0
  | Var -> Const 1
  | Sum(e1,e2) -> Sum(deriv e1, deriv e2)
  | Prod(e1,e2) -> Sum(Prod(e1, deriv e2), Prod(deriv e1, e2))
```

Strong static typing with type inference (Hindley-Milner)

Most functional languages are statically strongly typed:

```
# "xyz" ^ 123
This expression has type int but is here used with type string
```

The compiler infers types (no type declarations on functions):

```
# let geom_mean a b = sqrt(a *. b)
val geom_mean : float -> float -> float
```

Generic functions receive polymorphic types:

`List.map: ('a -> 'b) -> 'a list -> 'b list` for all types `'a,'b`

```
# List.map (fun x -> x+1) [1;2;3]
# List.map string_of_int [1;2;3]
```

Summary

Compilers for functional languages need to deal with (or take advantage of) the following features:

	ML	Haskell	Scheme
Full functionality	⋆	⋆	⋆
Declarative style	⋆	⋆	⋆
Imperative style	⋆		⋆
Lazy evaluation		⋆	
High-level data structures	⋆	⋆	⋆
Automatic memory management	⋆	⋆	⋆
Pattern-matching	⋆	⋆	
Strong static typing	⋆	⋆	
Polymorphism	⋆	⋆	⋆

Outline

1. Compiling full functionality: closures
2. Optimizing full functionality: flow analyses
3. Data representation issues
4. Code generation issues

Not treated in this tutorial:

1. Lazy evaluation (see Peyton-Jones' book).
2. Compilation of pattern matching (same).
3. Garbage collection techniques (see Paul Wilson's surveys).

Part 1

Compiling full functionality: closures

A naive approach

When evaluating `fun x -> e`, generate a block of code at run-time and return a pointer to that code:

```
        let scale n = fun x -> x * n
```

```
let f = scale 2                 let f = scale 10


res <- arg                      res <- arg
mul res, 2                      mul res, 10
return                          return
```

Problems:

- run-time code generation is expensive;
- for big functions, this allocates large blocks of code.

Towards a better solution

Notice that all the generated blocks of code share the same "shape": they differ only in the values of the `n` variable at the time of evaluation of `scale n`.

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

Idea: share the common code and put the varying parts (i.e. the value of `n`) in some external data structure.

Closures (P. J. Landin, 1964)

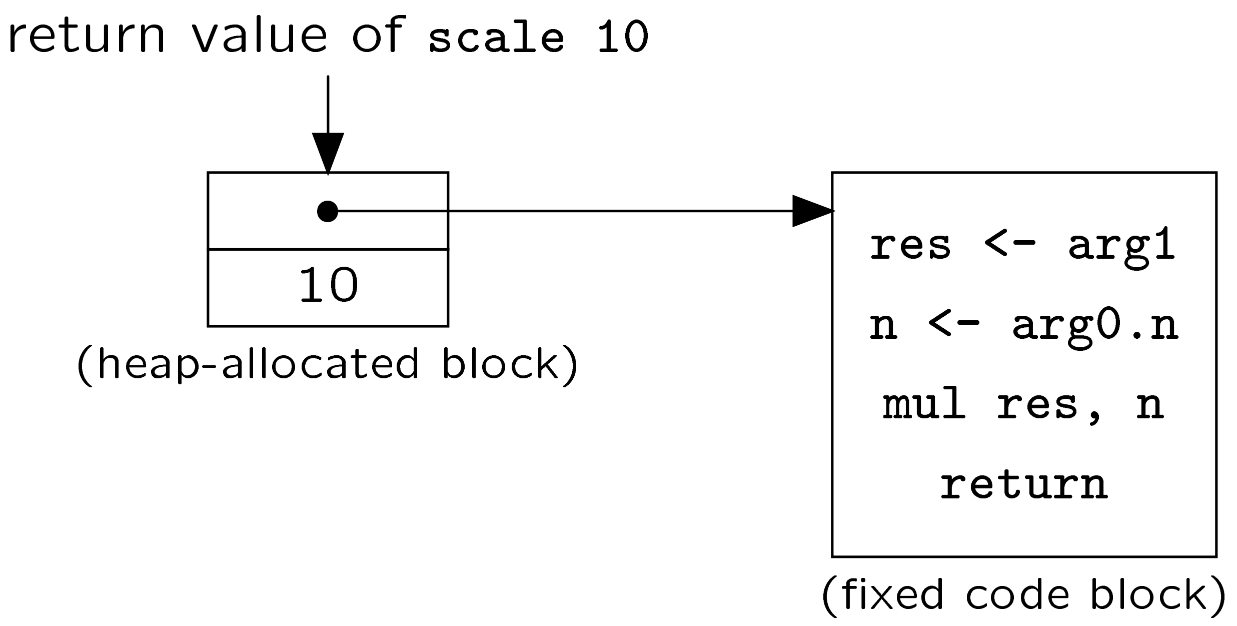
All functional values are represented by *closures*.

Closures are heap-allocated data structures containing:

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

(The free variables of the function are all variables appearing in the function that are neither parameters to the function, nor `let`-bound inside the function.)

Example of closure



The compilation scheme for functions and applications

```
⟦a(b)⟧ =
    let clos = ⟦a⟧ in clos.code clos ⟦b⟧

⟦fun x -> e⟧ =
    let code_f clos x =
        let v = clos.env_v and w = clos.env_w and ... in ⟦e⟧
    in
        { code = code_f; env_v = v; env_w = w; ... }
```

where `v`, `w`, ... are the variables free in **`fun x -> e`**.

Note: the function **`code_f`** above has no free variables and is represented by a pointer to its (fixed) code.

Translating functions into an object-oriented language

Closures are a special case of classes and objects having only one `apply` method.

For the `scale` example, we get:

```
class scale {
  private int env_n;
  int apply(int x) { return x * env_n; }
  scale(int n) { env_n = n; }
}
```

Build the closure = instantiate the class (`new scale(10)`).

Apply the closure = call its `apply` method (`s.apply(5)`).

Closure representation strategies

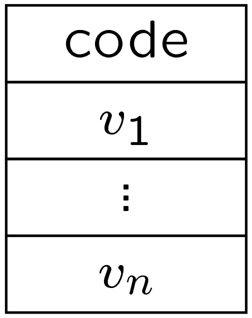
- When compiling an application, nothing is known about the closure being called (this can be the closure of any function in the program).

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

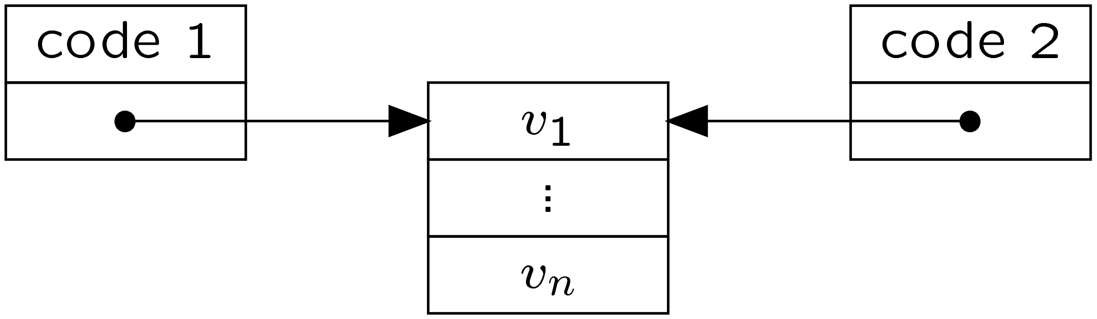
- The environment part of a closure is not accessed during application. The structure of the environment matters only to the code that builds the closure and the code for the function body.

  → Considerable flexibility in choosing the layout for the environment part.

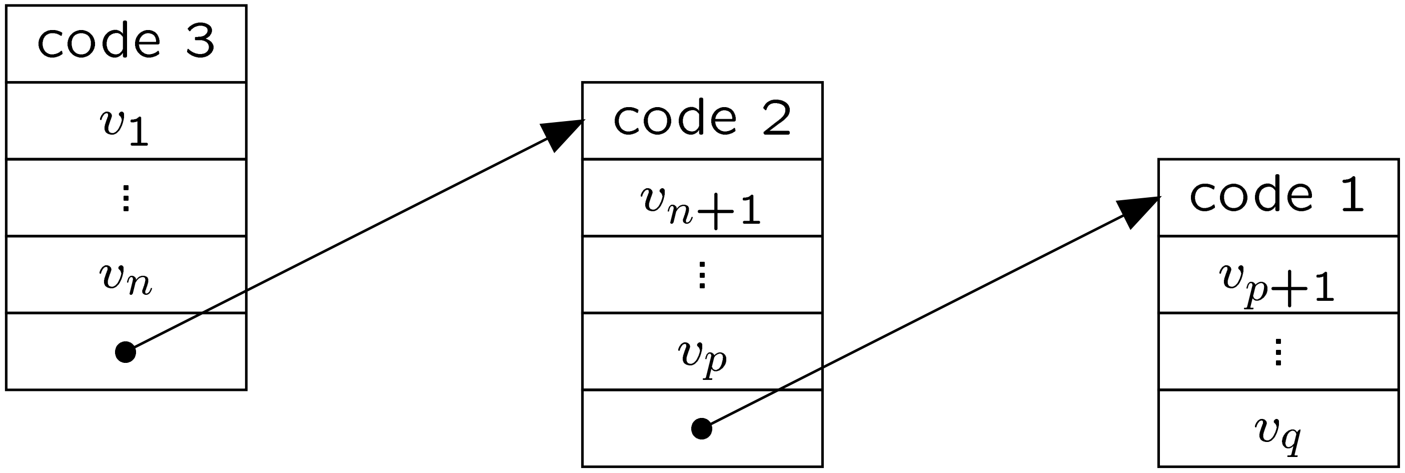
One-block closure



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 ...
```

That's easy because the code for the function receives the closure itself as extra argument:

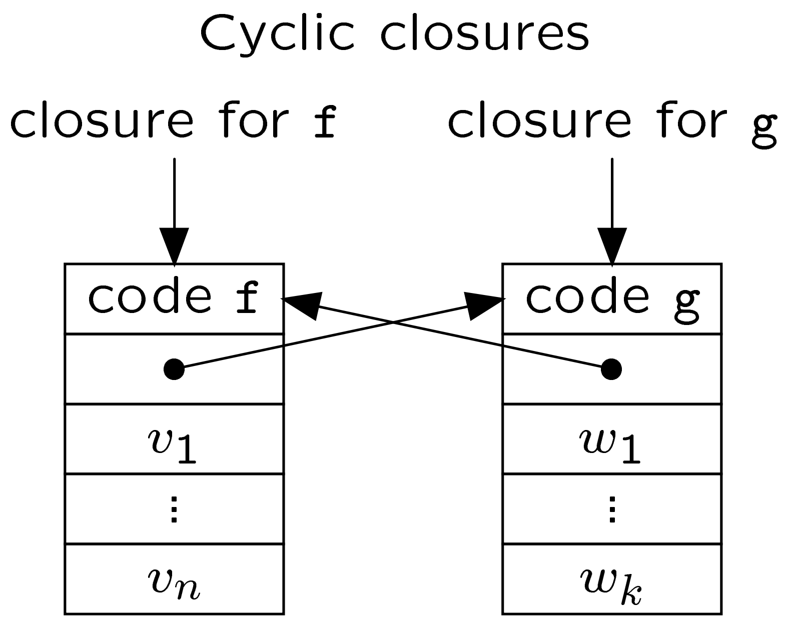
```
        let rec code_f clos x =
          let f = clos in
          let v = clos.env_v in
         ... List.map f l ... in

        let f = {code = code_f; env_v = v}
```

Mutually recursive functions

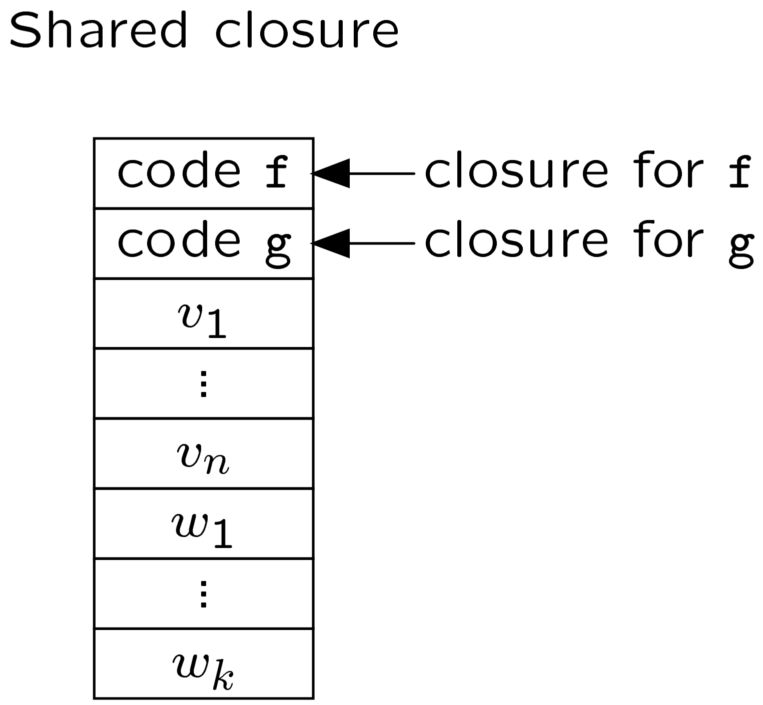
Mutually recursive functions need access to the closures of all functions in the mutual recursive definition:

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

- 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 (Appel, 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

Optimizing full functionality: flow analyses

The overhead of closure calling

The introduction of closures makes function applications more expensive:

- one load (to recover the code pointer from the closure)
- one call to a computed address (to call the code pointer)

The latter is expensive on modern processors (pipeline stall).

Contrast with a non-functional language:

- one call to a known address (no pipeline problems);
- or even no overhead at all if we inline expand the body of the called function (if it is small enough).

Opportunities for optimizations

The overhead of closures can be avoided in many practical situations:

- Many function applications always call the same function:

  ```
  let rec f x = ... f e ...
  ```

  `... Sort.list order l ...` *(no other call to* `Sort.list`*)*

  $\rightarrow$ generate calls to known code addresses.

- Many functions are small
  $\rightarrow$ good candidates for inline expansion.

More opportunities for optimizations

- Lightweight closure conversion (Wand and Steckler, 1992): sometimes the full power of closures is not needed and simpler representations can be used for functional values.

  Example: all functions that can be called at some point are closed $\rightarrow$ represent them by mere code pointers.

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

Control-flow analyses

Control-flow analyses (Shivers, 1991) 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 {`fun x -> e`}, we can generate a direct call to the code for `e`, or inline it.
- If all elements in that set are “simple” (e.g. closed), consider lightweight closures.
- In all cases, we 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:

producer point	$\longrightarrow$	consumer point
(constant, `fun x -> e`, cons, `+`, ...)		(function parameter, argument to cons, argument to `+`, ...)

Example of constraint generation rules

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

  $$\begin{array}{rcll} V(n) & \subseteq & V(\ell) & \text{(the \texttt{then} branch flows to the result)} \\ V(p) & \subseteq & V(\ell) & \text{(the \texttt{else} branch flows to the result)} \end{array}$$

- For `(a`^{m}`(b`^{n}`))`$^\ell$:
  for each function `fun x -> e`^{q} in $V(m)$, add the constraints

  $$\begin{array}{rcll} V(n) & \subseteq & V(\mathtt{x}) & \text{(the argument flows to the parameter)} \\ V(q) & \subseteq & V(\ell) & \text{(the function result flows to the application result)} \end{array}$$

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

An example of CFA

```
let rec apply_list l arg =
  match l with
    [] -> []
  | hd :: tl -> hd(arg) :: apply_list tl arg
```

```
apply_list ((fun x -> x+1) :: (fun x -> x-1) :: []) 1
```

Summary on CFA

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

Can be performed module per module with some loss of precision.

Main applications:

- Optimize function calls in functional languages.
- Optimize method dispatch in object-oriented languages.
- Eliminate run-time type tests in Scheme.
- 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, 1997) start with $\top$ on all variables and do 1 or 2 iterations.
- Use equality constraints (unification) in addition to inclusion constraints.

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 (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.

More connections

- Type inference algorithms for type systems with subtyping are based on inclusion constraints similar to those used by CFA. (Aiken and Wimmers, 1993; Smith et al, 1995.)
- Rich type systems (with intersection and union types) have been used to represent the results of flow analyses and exploit them for closure optimizations (Muller et al, 1997).

Part 3

Representing 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.
$\rightarrow$ clever representation tricks are feasible.
(Would be hard to do by hand in C.)

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

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

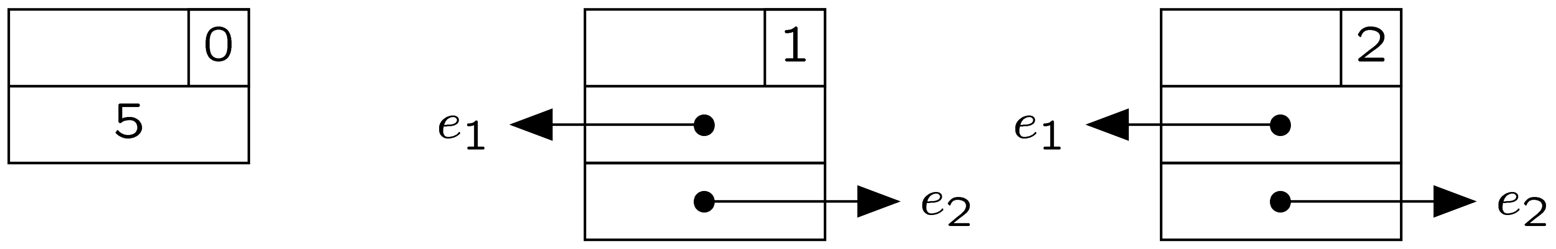
Constructors with arguments are represented by word-aligned pointers to heap blocks.
(Bit pattern: $\ldots xx00$)

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.

```
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)



Further optimizations

On a 64-bit processor, encode the tag byte in the pointer itself:

Constructor with argument	*tag*	*pointer* (56 bits)
Constant constructor	*tag*	0000 ...

Allows testing the tag without any memory access.

```
type expr =
    Const of int          (* pointer 00... *)
  | Var                   (* null pointer 010000... *)
  | Sum of expr * expr    (* pointer 02... *)
  | Prod of expr * expr   (* pointer 03... *)
```

Hard to do by hand in C (backward compatibility with 32-bit platforms).

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).
  $\rightarrow$ floats are boxed (heap-allocated);
  $\rightarrow$ records are boxed;
  $\rightarrow$ arrays are arrays of pointers to boxed elements.
- All functions must use the same calling conventions:
  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:

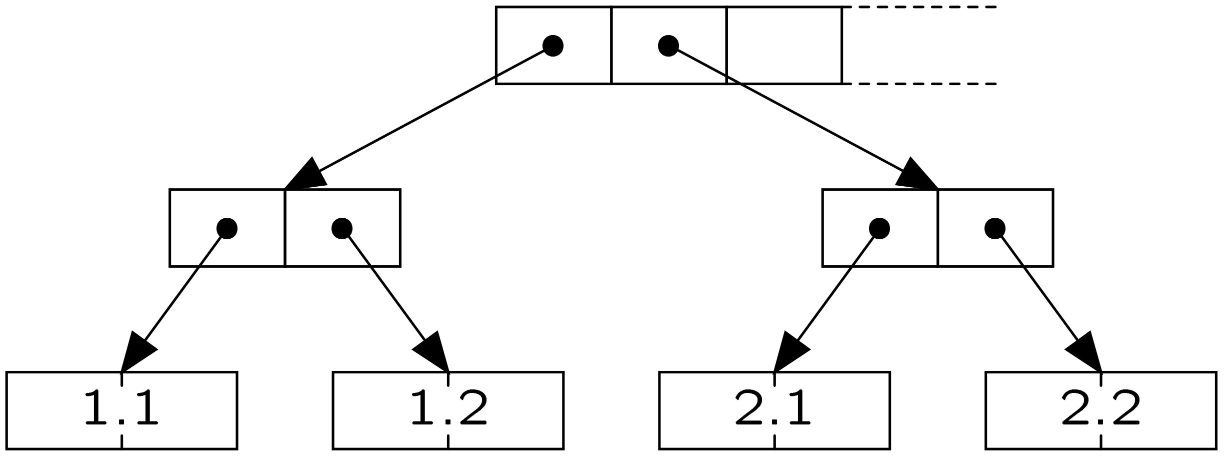
  $$|\texttt{int}| = 1 \text{ word} \qquad |\texttt{float}| = 2 \text{ words} \qquad |\tau \times \sigma| = |\tau| + |\sigma|$$

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

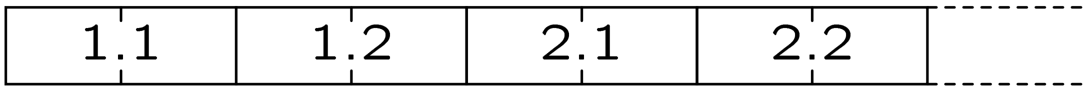
$\texttt{float} \rightarrow \texttt{float}$	argument in `FP0`, result in `FP0`
$\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$	argument in `R0` and `R1`, result in `R0`

Example: an array of 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` : $\forall\alpha.\ \alpha \to \alpha$

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 $\langle\beta \times \beta\rangle$ (x, x)
`g 5`	g $\langle$int$\rangle$ 5

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

Type-dependent data layout

The TIL approach (Harper, Morrisett, et al, 1994):

- 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: Generated code, TIL style:

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

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

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

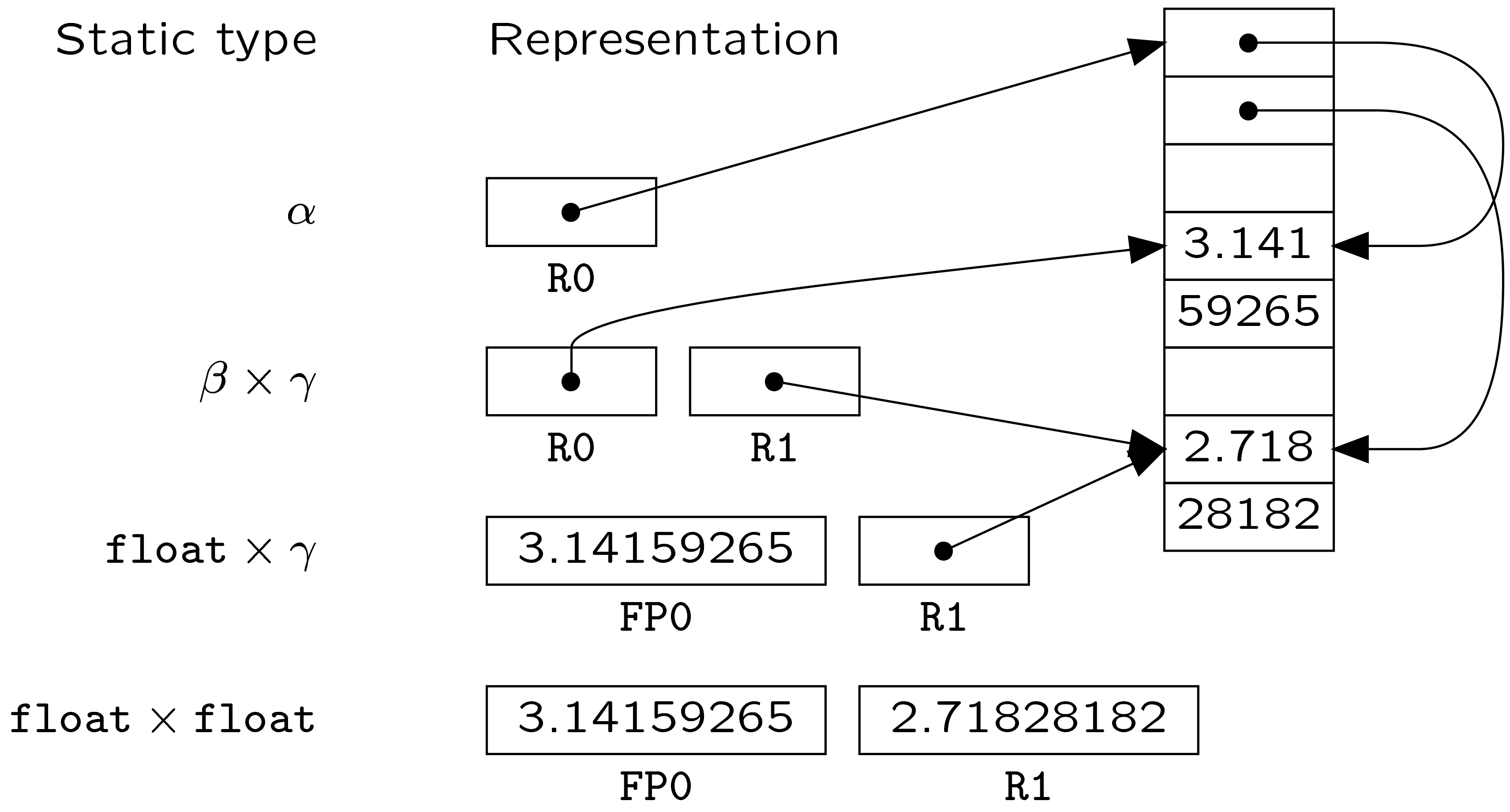
Mixed data representations

(Leroy, 1992; Shao and Appel, 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 $\times\ \gamma$
3.14159265
FP0
R1
float $\times$ float
3.14159265
FP0
2.71828182
R1
3.141
59265
2.718
28182

Source code:

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

Coercion diagram:

$$\begin{tikzcd}
\mathtt{R0} \arrow[r, "\mathtt{make\_pair}"] & \mathtt{R0,} \arrow[d, "\mathtt{unbox}"'] & \mathtt{R1} \arrow[d, "\mathtt{unbox}"] \\
\mathtt{FP4} \arrow[u, "\mathtt{box}"'] & \mathtt{FP2,} & \mathtt{FP3}
\end{tikzcd}$$

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

$$\begin{array}{rcl@{\qquad}rcl}
[\alpha \Rightarrow \texttt{int}] & = & \texttt{identity} & [\texttt{int} \Rightarrow \alpha] & = & \texttt{identity} \\
[\alpha \Rightarrow \texttt{float}] & = & \texttt{unbox\_float} & [\texttt{float} \Rightarrow \alpha] & = & \texttt{box\_float} \\
[\alpha \Rightarrow \beta \times \gamma] & = & \texttt{unbox\_pair} & [\beta \times \gamma \Rightarrow \alpha] & = & \texttt{box\_pair}
\end{array}$$

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

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

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, 1991; Goubault, 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 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 ...
```

Part 4

Code generation issues

Recapitulation

The front-end of a compiler for a functional language performs:

- closure introduction and optimization;
- compilation of pattern-matching into simple tests;
- making explicit data representations, heap allocation, and pointer dereferencing.

This done, we are back to a simple intermediate language (similar to a subset of C with support for garbage collection).

→ Generate assembly using conventional back-end technology.

→ Or use a virtual machine.

Virtual machines

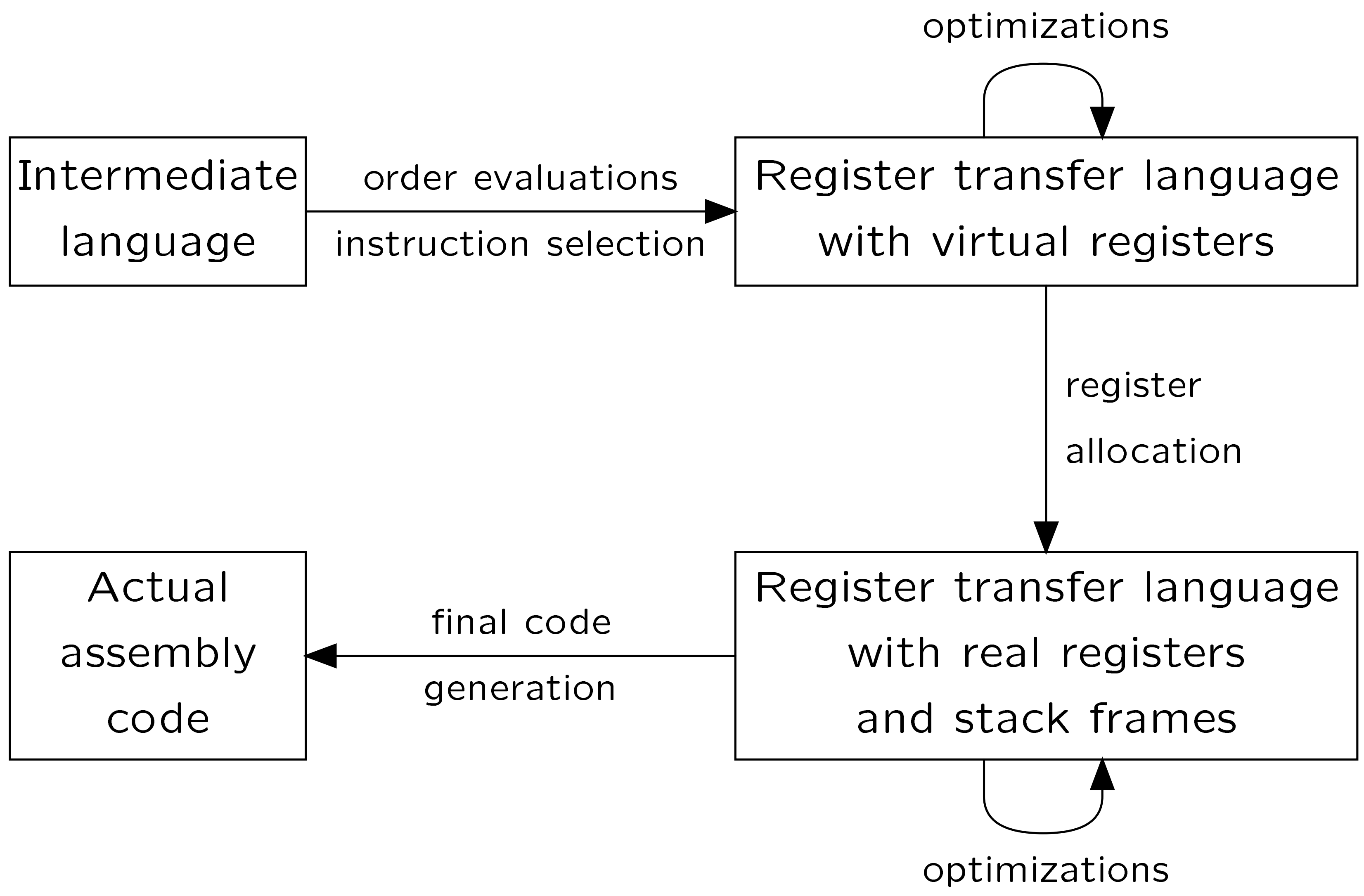
The code for a virtual machine is executed not by hardware, but by a a *bytecode interpreter* (usually written in C).
$\rightarrow$ high-portability; platform-independence.

Virtual machine instruction sets usually consist in:

- Simple, generic stack-based instructions for arithmetic, branches, etc (mostly independent of the source language).
- Complex instructions specialized to the source language:
  For functional languages: closure construction, closure application, allocation of data structures.
  For Java: method invocation, object allocation.

Many known tricks to make bytecode interpreters efficient (Piumarta, 1998).

Organization of a typical back-end



Relevance of conventional compiler optimizations

When compiling to assembly code, keep in mind that functional programs have unusual characteristics compared with Fortran or C programs (e.g. the SPEC benchmarks):

- Few floating-point operations
- Many memory accesses
- Few proper loops (but tail-recursive functions are loops)
- Small basic blocks
- Frequent function calls
- Lots of heap allocation and garbage collection.

$\rightarrow$ Many classic compiler optimizations are not very effective.

Most important optimizations

- Procedure optimizations (inlining, tail-call optimization, leaf routine optimization, lightweight calling conventions).
- Register allocation.
- Spill code optimization (early saves, lazy restores).
- Efficient heap allocation sequences.
- Efficient garbage collection interface.

Not so important optimizations

- Loop and array optimizations (e.g. strength reduction, induction variables).
- Classic intraprocedural optimizations (constant propagation, constant folding, copy propagation, common subexpression elimination).
- Instruction scheduling (processors with out-of-order execution do it dynamically very well, especially on small basic blocks).

Note however some attempts at extending software pipelining to recursive functions (Pouzet, 1995).

Optimizations that have not been tried yet

- Link-time optimizations based on whole program analysis:
  - Interprocedural register allocation.
  - Specialization of polymorphism and type abstraction.

  Linker technology lags behind.

- Optimization w.r.t. the memory hierarchy.

  Very little is known, and what is known applies only to arrays but not heap-allocated structures (tiling, loop interchange).

Interactions with the garbage collector

Garbage collectors walk the memory graph.

For this, they need to know the *memory roots*: the pointers into the heap contained in registers and in the stack.

Conservative garbage collectors can work even if some of the "roots" given to them are not actually heap pointers (such as return addresses, untagged integers, unboxed floats).
$\rightarrow$ Use the whole stack + all the registers as roots.

Exact garbage collectors (more efficient) must be given roots that are guaranteed to be well-formed heap pointers.
$\rightarrow$ We must know at GC-time which registers and which stack slots contain heap pointers.

Communicate type information to the garbage collector

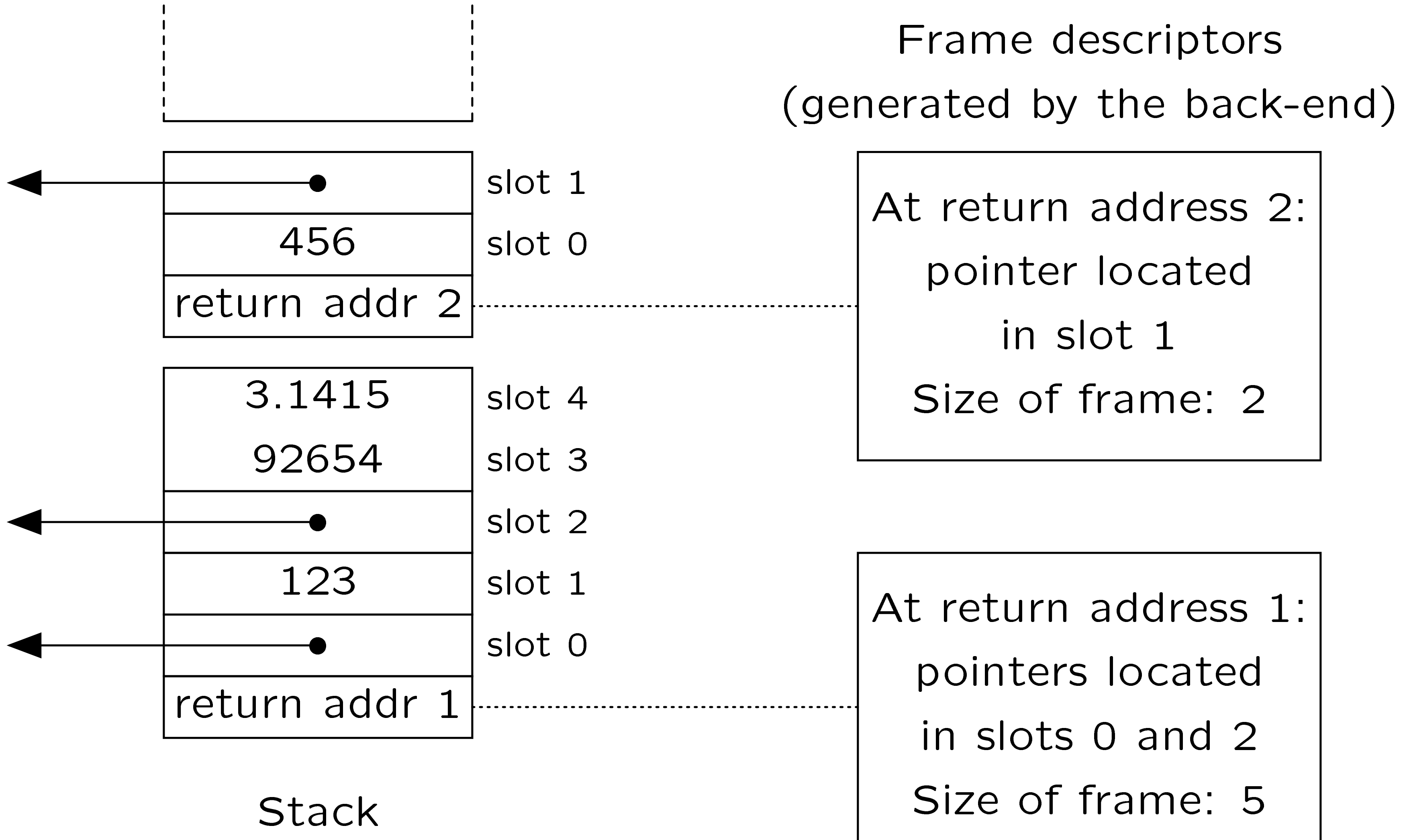
In the register transfer language, it is easy to annotate virtual registers with machine-level types:

heap pointer (or tagged integer)
untagged integer
unboxed float

This annotation is preserved during further passes (e.g. register allocation).

For each GC point (heap allocation, function call), generate a frame descriptor listing the actual locations of all virtual registers with type “heap pointer”.

Example



Frame descriptors
(generated by the back-end)
slot 1
456
slot 0
return addr 2
At return address 2:
pointer located
in slot 1
Size of frame: 2
3.1415
92654
slot 4
slot 3
slot 2
123
slot 1
slot 0
return addr 1
At return address 1:
pointers located
in slots 0 and 2
Size of frame: 5
Stack

Passing type information through the back-end

Generating the descriptors is easy, but requires that the back-end preserves and updates the type annotations.

Existing portable back-ends (e.g. C compilers, or VPO) don't do this.

C-- (Peyton-Jones et al, 1998): towards a GC-friendly intermediate language and portable back-end.

Conclusions

- Compilation technology for functional languages is relatively mature.

  On comparable programs, achieve at least 50% of the performance of optimizing C compilers (often more) (Hartel et al, 1996).

- Still needs work to keep up with the evolution of processors.

- Truly efficient functional programs still require programmers to be conscious of performance issues.