Compilation techniques for functional and object-oriented languages

Xavier Leroy

INRIA Rocquencourt, projet Cristal

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

PLDI tutorial, june 1998

Techniques de compilation pour les langages fonctionnels et à objets

Xavier Leroy

INRIA Rocquencourt, projet Cristal

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

Tutoriel PLDI, juin 1998

Outline

1. Basic translations for high-level features.

   Intermediate language and back-end requirement.

2. Optimizing function applications and method dispatches.

3. Optimizing memory accesses.

Part 1

Basic compilation techniques and requirements from the back-end

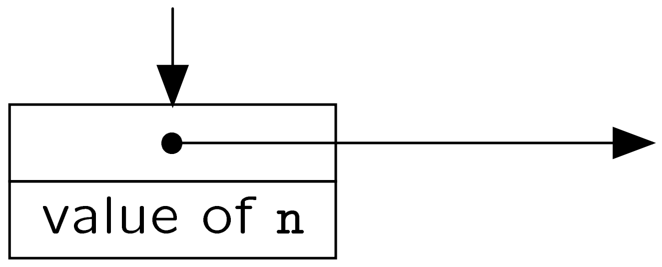
Strict functional languages

- Transformation of functions into closures.
- Layout of data types, heap allocation, pointer introduction.
- For ML: transformation of pattern-matchings into sequences of simple tests.
- For Scheme: introduction of run-time type tests.

Function closures

```
scale n = \x. x * n
```

evaluates to



(heap-allocated block)

```
int f(clos, x)
{ return x * clos->n; }
```

(fixed code block)

Closure application:

```
[[fun(arg)]] =
    closure * clos = [[fun]]; clos->code(clos, [[arg]])
```

Lazy functional languages

Same transformations as for strict languages, plus:

- Representation of delayed computations by heap-allocated data structures (thunks, graphs).

  When the result is needed, test if already computed and if not, perform the computation and overwrite the thunk with the result.

- Strictness analysis to determine when result can safely be computed immediately.

Object-oriented languages

- Layout of objects, heap allocation, pointer introduction.

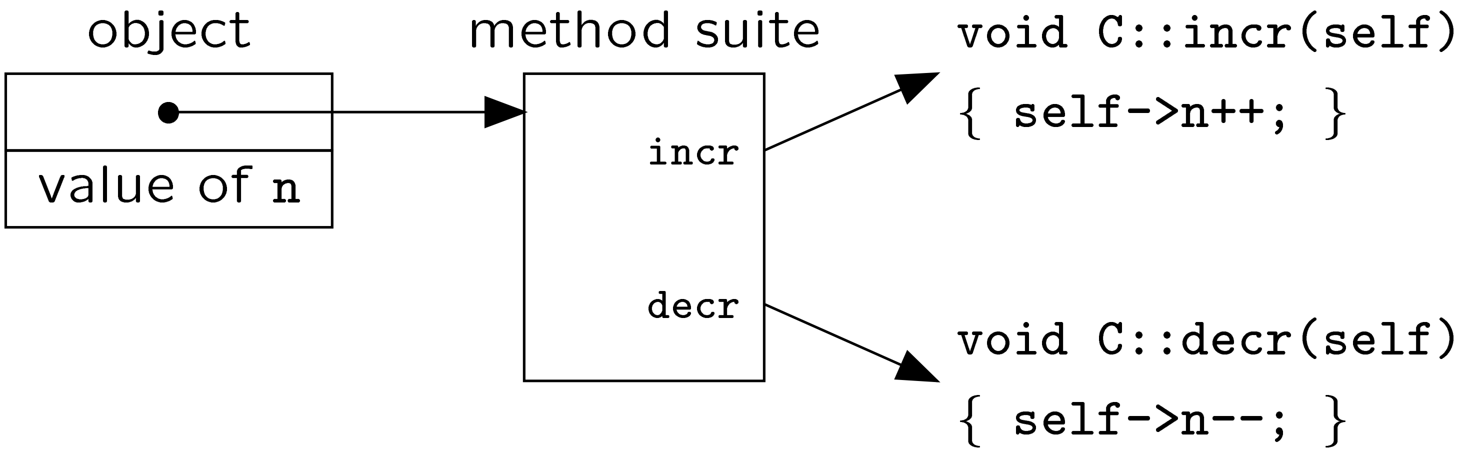
- Layout of method suites ("v-tables") attached to objects (based on inheritance and subtyping hierarchies).

- Transformation of method invocations into method lookups + calls.

- Introduction of run-time type tests if needed.

Objects and method suites

```
class C {
  int n;
  void incr() { n++; }
  void decr() { n--; }
}
```



Method invocation:

$[\![$`obj.meth(arg)`$]\!]$ =

  `object * o =` $[\![$`obj`$]\!]$`; lookup(o->vtable, meth)(o,` $[\![$`arg`$]\!]$`)`

The intermediate language

The target of the previous translations is a simple conventional language:

- Integer and floating-point operations.
- Loads and stores of words, bytes, single and double floats.
- C-style function calls (including function pointers).
- Classic control structures (`if then else`, loops, ...).

Actually: a subset of C.

Why not generate C source and use a C compiler as a back-end?

Or use a generic back-end (VPO, compiler infrastructure, ...)?

Why C is not an adequate target language

Crucial features missing from C and generic back-ends:

- Support for tail calls.
  (Caller deallocates its stack frame before jumping to callee.)

- Support exact (non-conservative) garbage collection.

Convenient features missing:

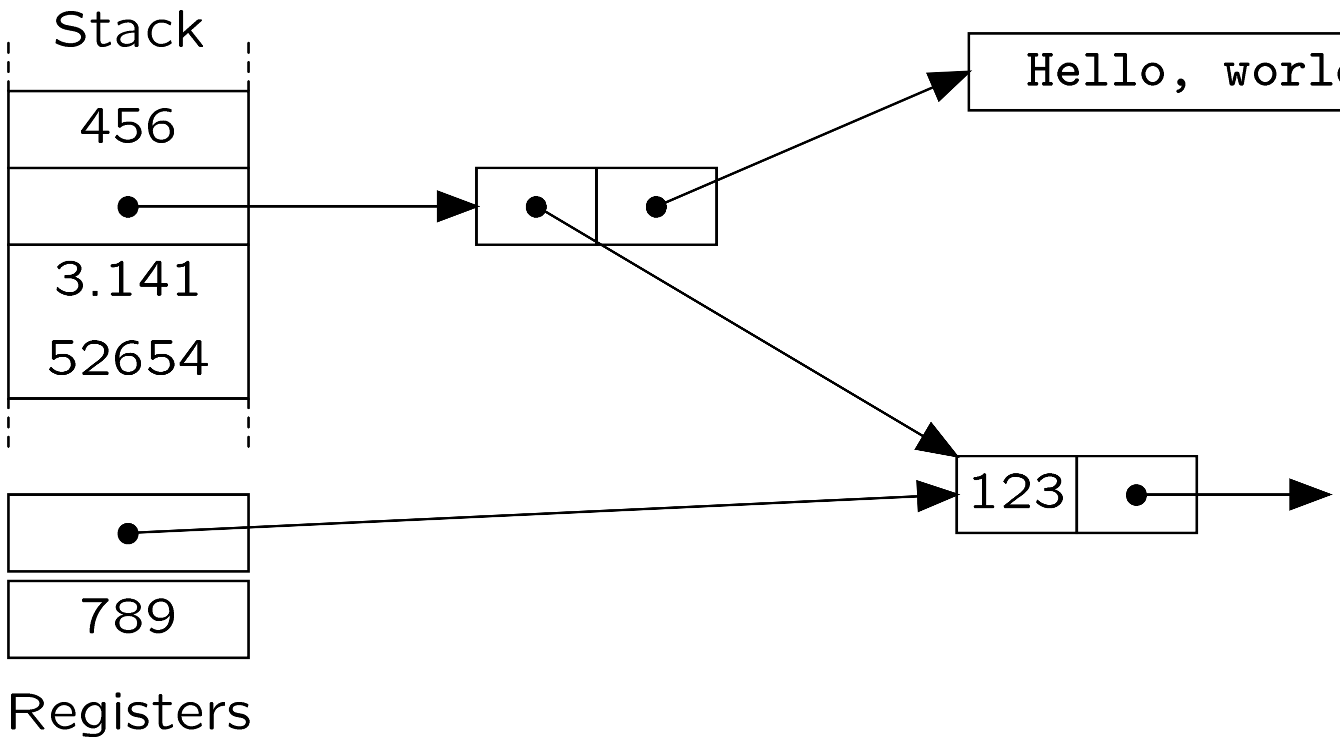
- Better arithmetic (overflow detection, multi-word arithmetic).

- Low-level idioms (using traps for array bounds checking).

The problem with exact garbage collection

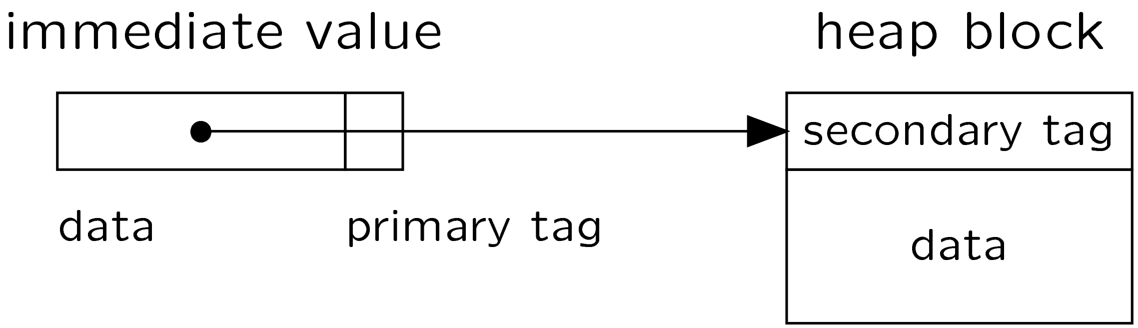
Exact garbage collectors need to distinguish pointers from immediate data in the registers, stack, and heap blocks.

(E.g. in order to relocate the pointers after moving heap blocks.)



Distinguishing pointers from immediate data

- In heap blocks: attach tags to data and to heap blocks.

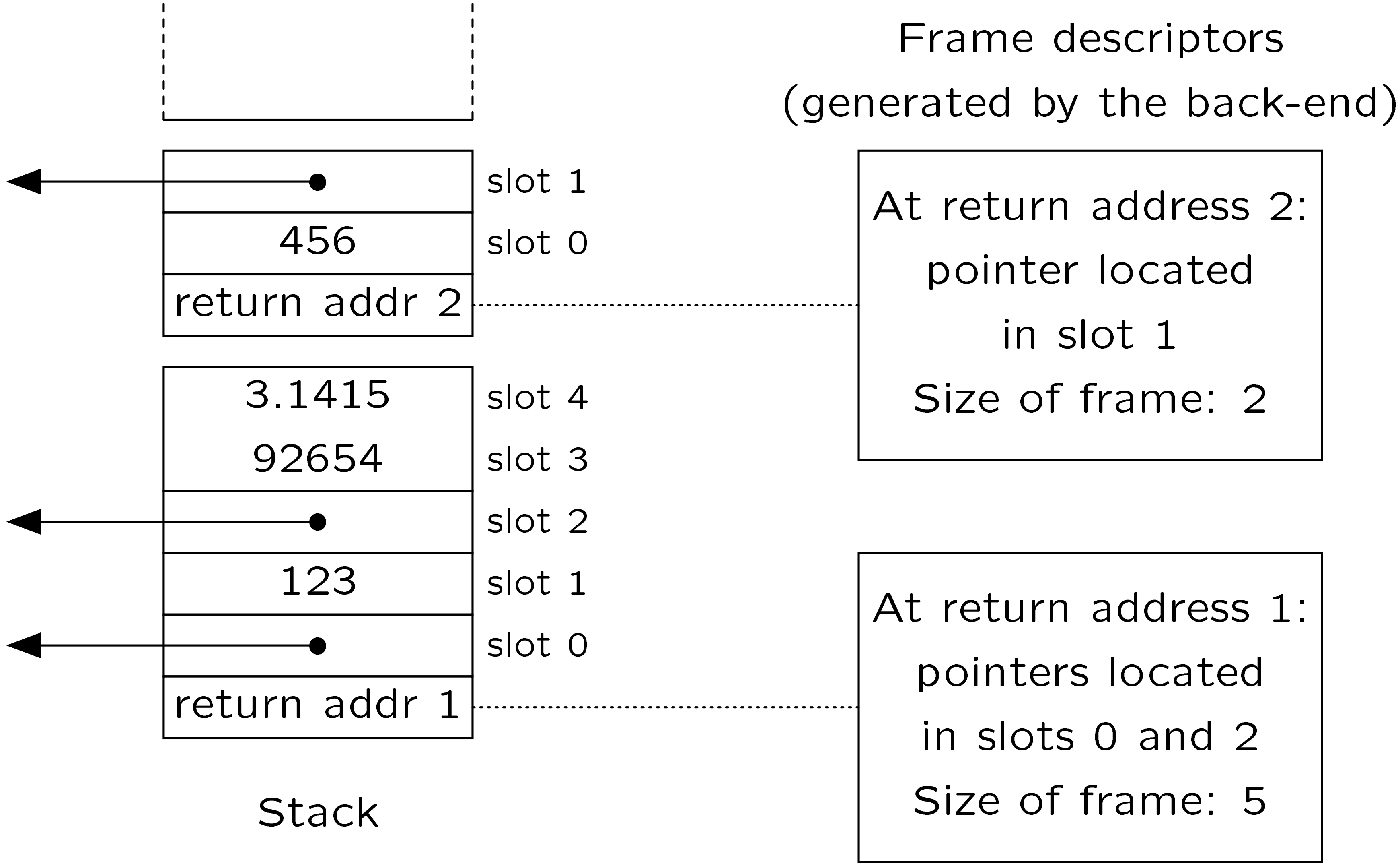


This can be done entirely by the front-end.

- In stack frames and in registers:
  - Have separate stacks and register files for pointers and non-pointers.
  - Or, generate frame descriptors telling the GC where the pointers are.

This requires cooperation from the register allocator.

Example of frame descriptors



Summary

- The high-level features of functional and OO languages are not hard to translate into a simple C-like intermediate language.

- The back-end technology required is mostly standard...

- ...except for:
  - tail calls;
  - transmission of type annotations from the front-end to the run-time system.

- It would be great if portable back-ends supported these.

Part 2

Optimizing function applications and method dispatch

The problem: computed branches

Examination of the code produced by naive functional or OO compilers reveal a high number of computed branches (calls and jumps to code addresses computed dynamically).

The naive compilation schemes produce one computed branch per function application or method invocation.

```
⟦fun(arg)⟧ =
    closure * clos = ⟦fun⟧; clos->code(clos, ⟦arg⟧)

⟦obj.meth(arg)⟧ =
    object * o = ⟦obj⟧; lookup(o->vtable, meth)(o, ⟦arg⟧)
```

The cost of computed branches

- On today's processors, computed branches stall the instruction pipeline for several (5 to 10) cycles.

  (Most processors perform dynamic prediction for conditional branches and function returns, but not for computed calls.)

- We do a lot of those.

  (Steenkiste and Hennesy: one function call or return every 11 instructions.)

- Since the target of the call is not known at compile-time, inlining of small functions and methods is not possible.

- For the same reason, most interprocedural analyses do not apply.

Example: invocation of a constant function or method

Type of invocation	Time	Instructions performed
Function inlined at point of call	0 cycles	load immediate
Direct call to known function	4 cycles	load immediate (argument), call to known address, load immediate (result), return
Generic call through function closure	11 cycles	load immediate, load code pointer, call to computed address, load immediate, return
Generic method invocation	18 cycles	lookup in two-level array (3 loads, 2 ALU), call to computed address, load immediate, return

(Measured on a PowerPC 603e with the Objective Caml 1.07 compiler.)

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 {`\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, required for later interprocedural optimizations.

Control-flow analyses in object-oriented terms

Similar analyses have been developed in the OO world (concrete type inference, interprocedural class analysis, ...).

The goal is to approximate at each method invocation point the set of classes to which the object may belong.

- If that set is a singleton $\{c\}$, we can generate a direct call to $c.m$, or inline it.
- Same if all elements in that set have the same implementation of method m (e.g. because they inherit it from the same superclass).
- Also gives an approximation of the call graph.

A high-level view of CFA

Since functions and objects are first-class values, CFA is actually a data-flow analysis that keeps track of the flow of functions and objects, 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, `\x -> e`, `new C`, cons, `+`, ...)		(function parameter, argument to cons, argument to `+`, ...)

Example of constraint generation rules

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

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

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

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

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

An example of CFA

```
apply_list l arg =
  case l of
    [] -> []
  | hd : tl -> hd(arg) : apply_list tl arg
```

```
apply_list ((\x -> x+1) : (\x -> x-1) : []) 0
```

```
apply_list l arg =
  case l of
    [] -> []
  | hd : tl -> hd(arg) : apply_list tl arg


apply_list ((\x -> x+1) : (\x -> x-1) : []) 0
```

```
apply_list l arg =
  case l of
    [] -> []
  | hd : tl -> hd(arg) : apply_list tl arg


apply_list ((\x -> x+1) : (\x -> x-1) : []) 0
```

```
apply_list l arg =
  case l of
    [] -> []
  | hd : tl -> hd(arg) : apply_list tl arg


apply_list ((\x -> x+1) : (\x -> x-1) : []) 0
```

```
apply_list l arg =
  case l of
    [] -> []
  | hd : tl -> hd(arg) : apply_list tl arg


apply_list ((\x -> x+1) : (\x -> x-1) : []) 0
```

Summary on CFA

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

This is a whole program analysis, but it can also 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 unnecessary run-time type tests in dynamically-typed languages.
- Eliminate redundant boxing of function arguments and results in polymorphic languages.

More precise analyses

- Polyvariant analyses (n-CFA, polymorphic splitting, ...): distinguish between different call sites of the same function.

`f x = x + 1`	monovariant:	`x` is `0` or `1`
`g y = ... f 0 ...`	polyvariant:	`x` is `0` when `f` called from `g`
`h z = ... f 1 ...`		`x` is `1` when `f` called from `h`

- Finer approximation of values (Heintze's set-based analysis): capture the shapes of data structures using grammars.

Less precise but faster analyses

- Coarser representations of sets of values:
  $\emptyset$ or $\{v\}$ (singletons) or $\top$ (all values).

- Do not iterate till fixpoint:
  start with $\top$ on all variables and do 1 or 2 iterations.

- Use equality constraints (unification) in addition to inclusion constraints.

- Exploit type information (class hierarchy analysis, rapid type analysis).

```
C obj; ... obj.m() ...
```

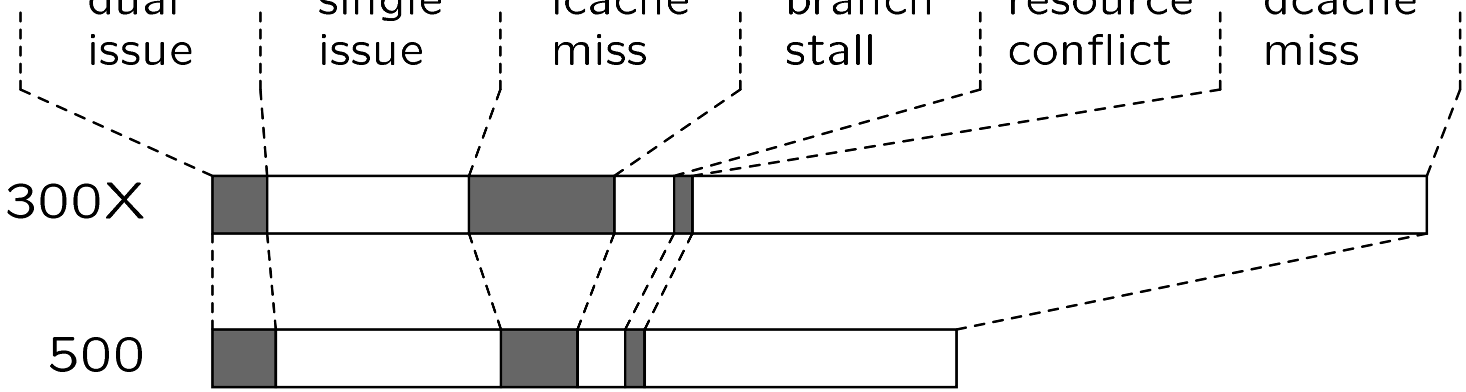
  Can only call methods `m` from `C` and its subclasses.

Part 3

Optimizing memory accesses

The Knuth-Bendix benchmark (in Objective Caml)

Machine	Dec 3000/300X	Dec 3000/500
Processor	21064, 175 MHz	21064, 150 MHz
Running time	4.85 s	3.00 s
Cycles per instruction	3.9 CPI	2.1 CPI



Primary cache miss:	17 cycles	9 cycles
Secondary cache miss:	85 cycles	35 cycles

Symbolic processing

Most functional programs and many OO programs perform symbolic processing as opposed to numerical computations.

Main characteristics of symbolic processing:

- Mostly loading data somewhere and storing it elsewhere. Few integer and floating-point computations.
- A lot of pointer chasing; irregular patterns of memory accesses; poor locality.
- High rate of heap allocation; frequent garbage collections.

Symbolic processing is memory-limited

For symbolic processing:

- Execution speed depends crucially on the memory subsystem. Other architectural features such as multiple ALUs have low impact on performances.

- Compiler optimizations that do not optimize loads and stores have little impact on overall performance.

- Shall we just wait for hardware designers to come up with better memory subsystems?

  (They do. Example: on a Pentium Pro 200 MHz, Knuth-Bendix runs at 1.1 CPI.)

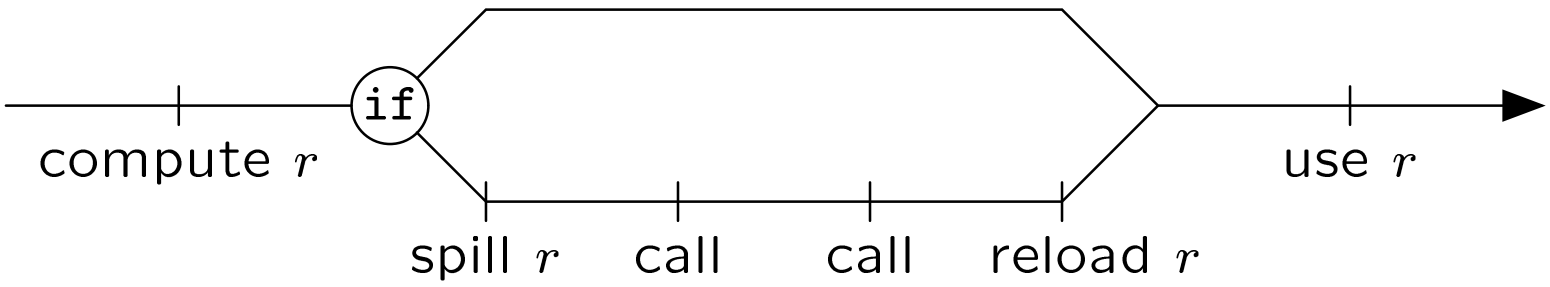
Optimizing memory accesses

- Optimizing control-related memory accesses:
  - Allocation of activation records.
  - Register spilling around function calls.
- Optimizing data-related memory accesses:
  - Unboxing data representations.
  - Aggressive scheduling of memory accesses.
- Cache behavior of (generational) heap allocation.

Register spilling around function calls

In functional and OO languages, the main source of spilling is function calls, not excessive register pressure.

- Avoid spills and restores in paths that do not call functions (1/3 procedure activations contain no function calls; 2/3 procedure activations perform no function calls).
- Spill only when a function call is inevitable.
- Reload only when a use of the variable is inevitable.
- Strict callee-save and strict caller-save are suboptimal.



Allocation of activation records

Frequent function calls $\rightarrow$ many allocations of activation records.

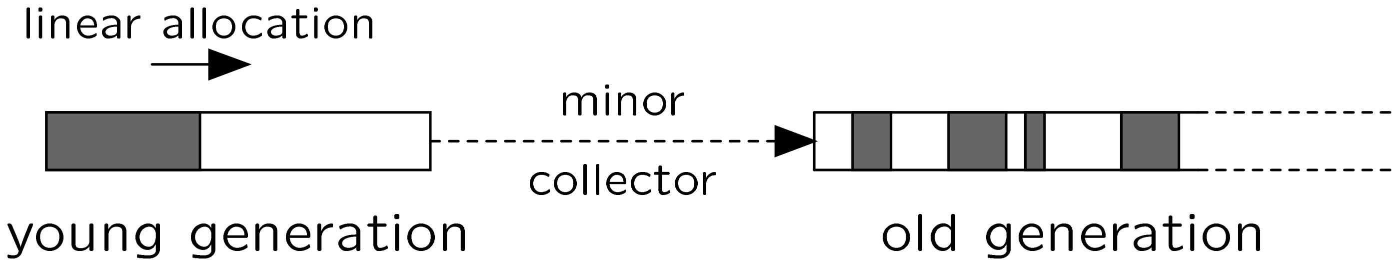
- Stack-allocated activation records:
  common wisdom; good cache behavior.

- Heap-allocated activation records:
  - Many nice applications: first-class continuations, lightweight threads, ...
  - Heap allocation can be almost as fast as stack allocation.
  - Heavy load on garbage collector.
  - Cache behavior is the main problem.

Cache behavior of linear heap allocation

- Reads to recently-allocated objects have good spatial locality and low miss rate.
- Writes to newly-allocated objects have $\approx$ 100% miss rate.
- Write misses can be absorbed by several architectural features (large write buffers, sub-block placement, ...).
- For better write behavior, can prefetch (or better preallocate) next block cache in young generation.

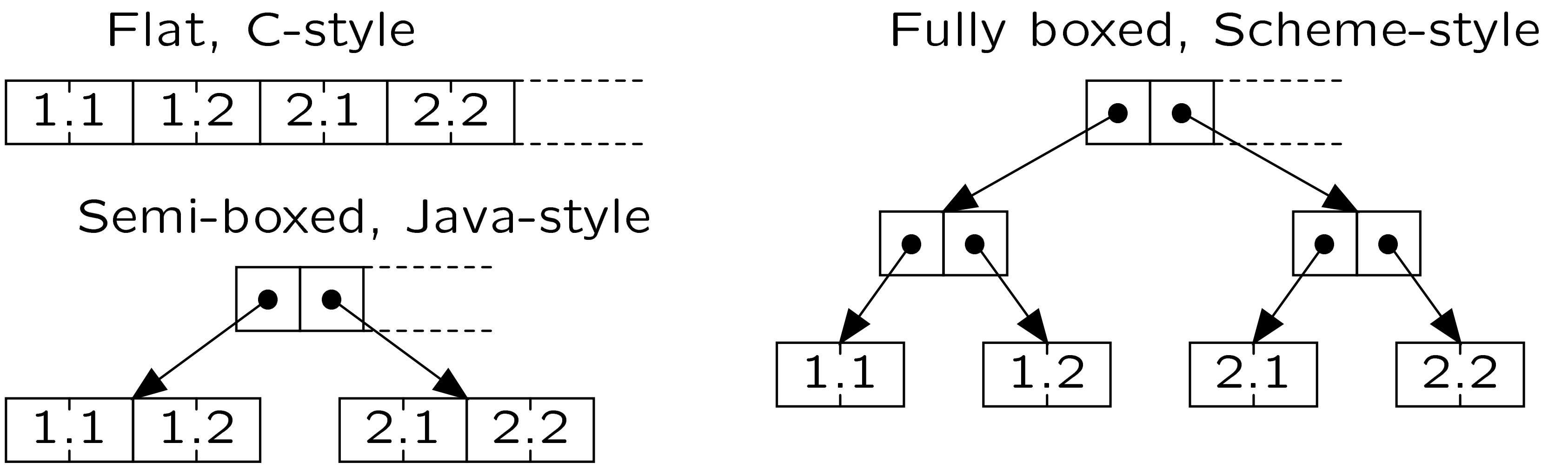
  (Objective Caml 1.07, PowerPC 603e, `dcbz` on next cache line at each allocation $\rightarrow$ 5–8% speedup.)


linear allocation
minor
collector
young generation
old generation

Unboxing data representations

Boxing consists in heap-allocating a piece of data and handling it through a pointer.

Example: an array of 2-D points (pairs of floats).



Boxed representations are expensive (heap allocation, extra loads) but facilitate the handling of important language features such as polymorphism.

Unboxing and polymorphism

Boxed data representations support *polymorphism* (the ability for a piece of code to operate uniformly on data of different types).

- In Java, all class instances can be viewed with type `Object`
  $\rightarrow$ all objects must be represented by pointers
  $\rightarrow$ no unboxed tuples or records
  $\rightarrow$ arrays of records are pointer arrays, not flat arrays.

- In Scheme, ML, Haskell, any language value can be passed to a polymorphic function or put in a list.
  $\rightarrow$ all data representations fit in one word
  $\rightarrow$ tuples, records, double-precision floats are boxed
  $\rightarrow$ no flat arrays either

Several approaches have been proposed to make monomorphic code more efficient, at the expense of slowing down polymorphic code:

- Use unboxed representations inside data structures (flat arrays, flat records, ...)
  In monomorphic code, rely on static types to compute offsets at accesses.
  In polymorphic code, rely on run-time type tests instead.

- Use unboxed representations in monomorphic functions, relying on static type information for size and layout determination.
  When entering polymorphic code, coerce data to fully boxed form, and unbox it back on return.

Aggressive scheduling of memory accesses

Exploit semantic guarantees of the language to schedule loads and stores aggressively:

- Pointers to heap objects of incompatible types cannot alias.
- Pointers to newly allocated objects do not alias with any other pointer.
- In ML and Haskell, most data structures are *immutable*: once initialized, they cannot be modified. Loads from immutable data structures cannot interfere with other memory accesses.
- In ML and Haskell, pointers to many data types (tuples, records, arrays) are never `NULL` and always point to valid memory space. Loads from pointers to such objects can be lifted above conditional branches safely.

Limitations to instruction scheduling

Instruction scheduling is limited by the following factors:

- Function bodies are relatively small (even after inlining).
- Many conditional branches; small basic blocks.
- Few loops (although tail-recursive functions are loops), hence software pipelining does not apply often.

  (But see Pouzet for an attempt to generalize software pipelining to arbitrary recursive functions.)

It is still unclear whether loads can be scheduled early enough to e.g. absorb the latency of a L1 cache miss.

Conclusions

Compilation technology for functional and OO languages has progressed tremendously in the last 15 years.

(Several compilers stay within a factor of 2 of optimizing C/C++ compilers on comparable programs.)

Those high-level languages do raise new, challenging compilation problems (flow analysis, representation analysis).

They also simplify some traditionally hard problems (aliasing, load speculation).

Strong type systems and other semantic guarantees are the compiler writer's friends, not enemies.