THE EFFECTIVENESS of TYPE-BASED UNBOXING

X. Leroy

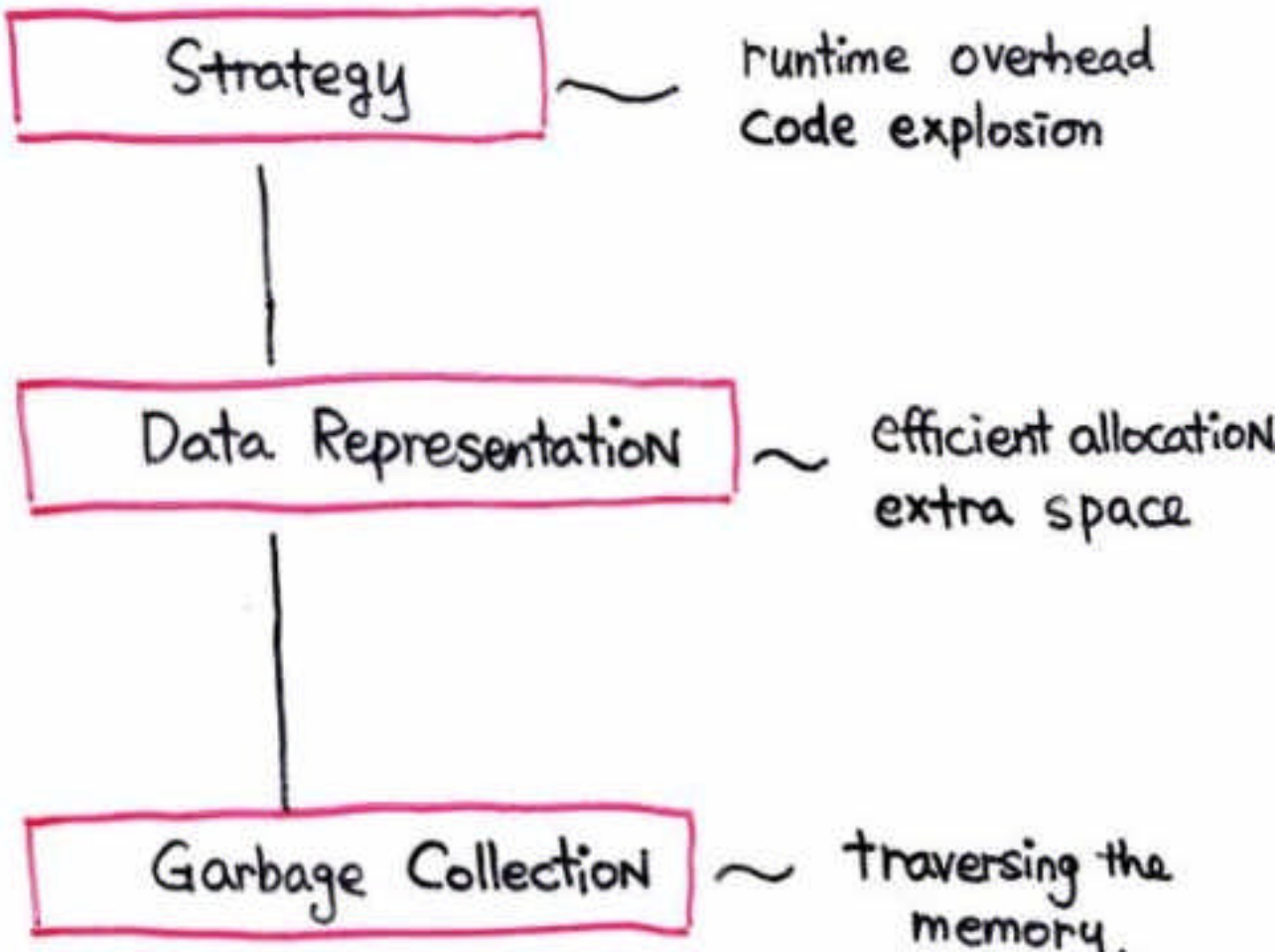
TIC 1997

Presented by Oukseh Lee
ROPAS, KAIST

Nov. 4, 2000

OCaml uses tag-based unboxing &
coercion-based unboxing of floats.

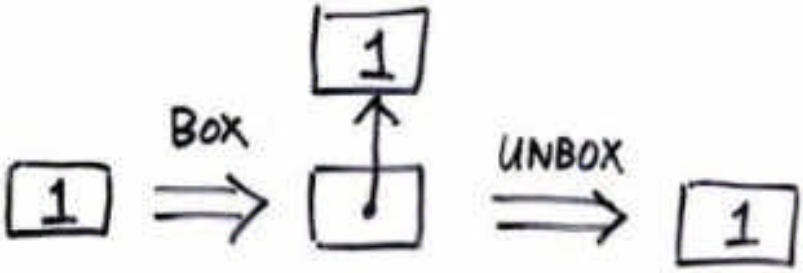
Consideration for Compiling Polymorphic Values



Present 3 method informally

Coercions , type-passing , tag-based

Coercions [Leroy 1992]

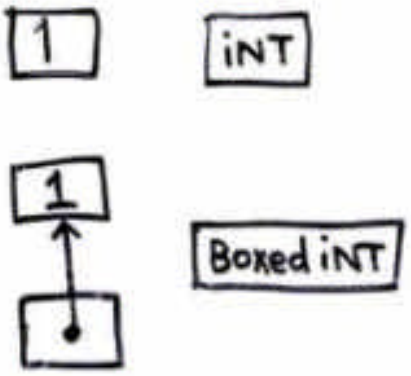


- insert explicit Box/UNBOX code
  - Monomorphic function: unbox → unbox
  - Polymorphic function: box → box

- Coersion on fns involve extra fn calls.
- Unnecessary sequence of Box/UNBOX codes.

Type-Passing [Harper and Morrisett 1995]

- type is passed to polymorphic fn.



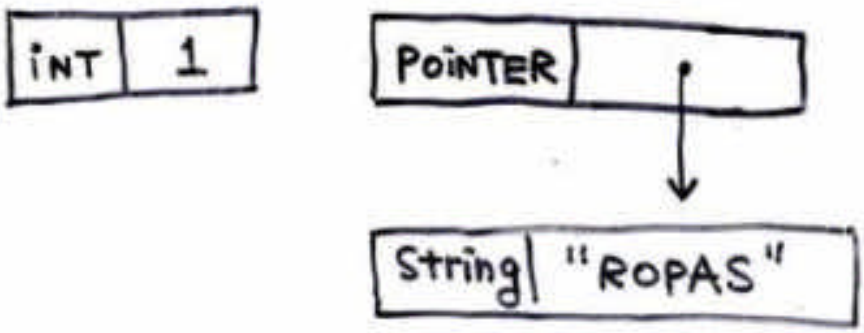
e.g.

```
sub = Λα. typecase α of
                INT ⇒ sub_int
                real ⇒ sub_real
                _ ⇒ sub_boxed [α]
```

- pass more arguments.
  maintain complex type.
  test complex type.

Tag-based Unboxing

- replace type by "tag".
  attach tag to data.



- Store tag ↛ extra space?
  ↘ merge with tag for GC.

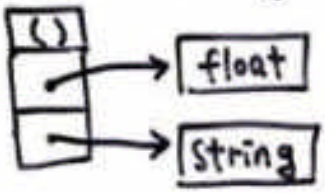
  tag size → encoded at bit-level
  (OCaml-3.0 use 8 bit for tag)

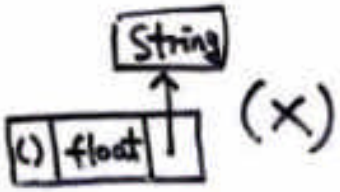
  runtime tag test → relatively inexpensive

GC Overhead

- Most strategies complicate GC's traversal of the memory graph.
  - how to distinguish whether it is a **POINTER** or not?
- Some Solutions (used in OCaml)
  - In stack frame, use frame descriptors which list the locations of the pointers
  - In Heap, does not mix pointers and data.

e.g float * String





(X)

- Alternatives
  - type-directed GC

Some BOX/UNBOX Optimization

- local unboxing

  eliminate BOX/UNBOX within the fn Body.

- partial inlining

  inline KNOWN fn's prelude & postlude.
  (unbox) (box)

Experimental results.

- Galium 2

  coercion-based unboxing
  tag-based unboxing of float arrays

- OCaml

  tag-based unboxing
  local unboxing of floats

Test	Gallium 2	Obj. Caml	What is tested
`bdd`	19.0	12.3	term processing, hash tables
`bdd` *	17.8	11.0	same as `bdd`, without array bounds checking
`boyer`	0.52	0.62	term processing, function calls
`fft`	3.49	2.00	floating-point arithmetic, float arrays
`fft` *	2.02	1.58	same as `fft`, without array bounds checking
`fib`	0.33	0.34	integer arithmetic, function calls (1 argument)
`genlex`	0.69	0.76	lexing, parsing, symbolic processing
`knuth-bendix`	3.00	2.47	term processing, function calls, functionals
`mandelbrot`	2.52	7.31	floating-point arithmetic, loops
`nucleic`	0.88	0.89	floating-point arithmetic, tree searching
`quad quad succ`	0.53	0.12	Church numerals, functionals, polymorphism
`quicksort`	1.44	0.65	integer arrays, loops
`quicksort` *	0.54	0.43	same as `quicksort`, without array bounds checking
`sieve`	1.03	1.01	integer arithmetic, list processing, functionals
`solitaire`	1.51	0.56	arrays, loops
`solitaire` *	0.41	0.38	same as `solitaire`, without array bounds checking
`takeushi`	0.41	0.39	integer arithmetic, function calls (3 arguments)

Times are given in seconds, averaged on three runs. The tests were conducted on an Alpha 21064-based Decstation 3000/400 running Digital Unix.

Figure 2: Performance comparison between Gallium 2 and Objective Caml 1.05