The effectiveness of type-based unboxing

Xavier Leroy

Vincent Foley-Bourgon
COMP-763 - Fall 2013
McGill University

September 2013
Plan

1. About the paper
2. The Big Idea
3. Type-directed unboxing
4. Type-directed unboxing overhead
5. Type-directed unboxing GC overhead
6. Untyped unboxing
7. Experimental results
About the paper
About the paper

- Written by Xavier Leroy, INRIA
- Published in 1997
- Presented at the “Types in Compilation” workshop of ICFP ’97
The Big Idea
Why do we need boxing?

C, Pascal

- All data types are known at compile-time
- Efficient memory layout
- Efficient calling conventions
Why do we need boxing?

Polymorphism and type abstraction

Compile-time type ≠ Run-time type

```ml
val triplicate : 'a -> 'a array
let triplicate x = [|x; x; x|]
```
Why do we need boxing?

ML

- Abandon C-style representation
- Revert to Lisp-style representation
- All data structures fit a common format (e.g. one word)
Boxing and unboxing

Explanation

**Boxing**: heap-allocating and handling through a pointer

**Unboxing**: getting at the primitive data through the pointer
Boxed values
So, what’s the problem?

- In tight loops, the constant boxing and unboxing is a major bottleneck
- Especially true in numerical applications
- Need a strategy to avoid unnecessary boxing/unboxing
- Some strategies rely on type information
- Others rely on program analysis, apply equally well to dynamically-typed languages
Monomorphisation

- Possible solution: monomorphisation
- Duplicate and specialize all generic functions for each type instanciation
- No major increase in code size
- Not viable for OCaml 😞
Type-directed unboxing
Coercions

- Coercions between boxed and unboxed representations inserted at type specialization points
- Generic code always operates on boxed values
- Monomorphic code can take advantage of unboxed representations
- 😊 Efficient register-based calling conventions
- 😞 Does not support deep unboxing (e.g. arrays of unboxed elements)
Run-time type inspection

- Run-time representation of static types maintained
  - Extra arguments for polymorphic functions
  - Extra fields for structures
- Generic code inspects those run-time type expressions
- 😊 Supports arbitrary unboxing in data structures
- 😞 Not very good with register-based calling conventions
Tag-based unboxing

- Used in dynamically-typed languages
- Type information is attached to the data structure
- Small set of base types, encoded at the bit level
Tag-based unboxing

OCaml

- 1-bit tagging
- Two kinds of arrays
  - Arrays of tagged ints or pointers
  - Arrays of unboxed floats
- Arrays with a concrete type: generate code for accessing arrays of pointers or floats
- Arrays with statically unknown type: test tag at run-time, and if float array, perform unboxing of floats
Type-directed unboxing overhead
Coercions

- Often, no overhead (boxing+unboxing would’ve happened anyway)
- Some examples show long sequence of successive unboxing+boxing before data is actually used
Run-time type inspection

Can anyone guess what the sources of overhead for RTTI are?
Run-time type inspection

- More arguments to pass
- Heap allocations to build tree of type expressions
- Testing the type expressions
- “Several techniques have been proposed to reduce overhead of type building or type inspection, but not both.”
Tag-based unboxing

- Shares some of the costs of RTTI, but not all
- In OCaml, tags are stored with GC information
- No overhead to function calls
- Run-time cost relatively small (one load, one compare)
- Extra conditional branches
- E.g.: OCaml 1.05: polymorphic array copy is 10x slower than int array copy, and 8x slower than float array copy
- In OCaml 4.00, naïve polymorphic array copy is \( \sim 2.5x \) slower than either int or float array copy
Type-directed unboxing GC overhead
Overhead in GC

What might be a source of overhead (and headaches) with an unboxing strategy?
Getting the roots in the stack

- Without unboxing, all values on the stack are either tagged ints or pointers
- With unboxing, some values are unboxed ints or floats
- Need to distinguish between boxed and unboxed values
- One possibility (used by OCaml): maintain a table of the pointers in the frame
Mixture of pointers and raw data in blocks

- With some unboxing strategies, heap blocks will contain pointers interleaved with unboxed values
- E.g. heap block containing a string * float * int list value
  - The string and list are boxed
  - The float is unboxed
- Maintain a table of the primitive types (pointer, int, float) in the block header
Untyped unboxing
Local unboxing

- Boxing and unboxing that cancel each other out in the same function body are eliminated by a dataflow analysis.

- How many boxing/unboxing operations in the following example?

```ocaml
let f (a: float array) (x: float) =
  let y = a.(0) *. x in
  y +. 1.0
```

- Simple and very effective on numerical code.

- Could be extended to inter-procedural analysis.
Known functions and partial inlining

- Functions in ML are usually curried
  
  ```ml
  let f a b c = a + b + c
  =>
  let f =
    fun a ->
      fun b ->
        fun c -> a+b+c
  ```

- Have two entry points: standard (curried) and quick (all arguments supplied)

- A control-flow analysis can determine if all arguments are supplied, and use the quick entry point

- In OCaml test suite, 80% to 100% of all function calls use the quick entry point
Experimental results
Match-ups

- Gallium 1 vs Gallium 1
  - One version is using coercion-based unboxing
  - The other is using fully boxed, tagged data representations

- Gallium 2 vs OCaml
  - Gallium 2: coercion-based, tag-based unboxing of float arrays
  - OCaml: mostly-tagged data representation, local unboxing of floats, multiple function entry points, tag-based unboxing of float arrays
## Gallium 1 vs Gallium 1

<table>
<thead>
<tr>
<th>Test</th>
<th>Unboxing</th>
<th>Boxed</th>
<th>Test type</th>
</tr>
</thead>
<tbody>
<tr>
<td>takeushii</td>
<td>3.00</td>
<td>5.09</td>
<td>fun calls, int arith</td>
</tr>
<tr>
<td>integral</td>
<td>0.80</td>
<td>2.83</td>
<td>float arith, loops</td>
</tr>
<tr>
<td>sumlist</td>
<td>3.60</td>
<td>3.45</td>
<td>lists, int arith</td>
</tr>
<tr>
<td>sieve</td>
<td>1.00</td>
<td>0.94</td>
<td>int arith, lists, polymorphism</td>
</tr>
<tr>
<td>boyer</td>
<td>1.80</td>
<td>2.76</td>
<td>fun calls, symbols</td>
</tr>
<tr>
<td>knuth-bendix</td>
<td>0.90</td>
<td>0.98</td>
<td>symbols, polymorphism</td>
</tr>
<tr>
<td>quad quad succ</td>
<td>6.58</td>
<td>2.40</td>
<td>Church numbers</td>
</tr>
</tbody>
</table>
Gallium 1 vs Gallium 1

![Gallium 1 vs Gallium 1 Graph](image)
Gallium 1 vs Gallium 1

- Unboxing strategy yields a noticeable performance boost in many tests.
- `quad quad succ` shows off one of the performance overhead of coercion-based unboxing.
## Gallium 2 vs OCaml

<table>
<thead>
<tr>
<th>Test</th>
<th>Gallium 2</th>
<th>OCaml</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bdd</td>
<td>19.0</td>
<td>12.3</td>
<td>term processing, hash tables</td>
</tr>
<tr>
<td>bdd *</td>
<td>17.8</td>
<td>11.0</td>
<td>bdd, bounds checking off</td>
</tr>
<tr>
<td>boyer</td>
<td>0.52</td>
<td>0.62</td>
<td>term processing, fun calls</td>
</tr>
<tr>
<td>fft</td>
<td>3.49</td>
<td>2.00</td>
<td>float arith, float arrays</td>
</tr>
<tr>
<td>fft *</td>
<td>2.02</td>
<td>1.58</td>
<td>fft, bounds checking off</td>
</tr>
<tr>
<td>fib</td>
<td>0.33</td>
<td>0.34</td>
<td>int arith, fun calls</td>
</tr>
<tr>
<td>genlex</td>
<td>0.69</td>
<td>0.76</td>
<td>lexing, parsing, symbols</td>
</tr>
<tr>
<td>knuth-bendix</td>
<td>3.00</td>
<td>2.47</td>
<td>term processing, fun calls</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>2.52</td>
<td>7.31</td>
<td>float arith, loops</td>
</tr>
<tr>
<td>nucleic</td>
<td>0.88</td>
<td>0.89</td>
<td>float arith, trees</td>
</tr>
<tr>
<td>quad quad succ</td>
<td>0.53</td>
<td>0.12</td>
<td>Church numerals, polymorphism</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.44</td>
<td>0.65</td>
<td>int arrays, loops</td>
</tr>
<tr>
<td>quicksort *</td>
<td>0.54</td>
<td>0.43</td>
<td>quicksort, bounds checking off</td>
</tr>
<tr>
<td>sieve</td>
<td>1.03</td>
<td>1.01</td>
<td>int arith, lists</td>
</tr>
<tr>
<td>solitaire</td>
<td>1.51</td>
<td>0.56</td>
<td>arrays, loops</td>
</tr>
<tr>
<td>solitaire *</td>
<td>0.41</td>
<td>0.38</td>
<td>solitaire, bounds checking off</td>
</tr>
<tr>
<td>takeushi</td>
<td>0.41</td>
<td>0.39</td>
<td>int arith, fun calls</td>
</tr>
</tbody>
</table>
Gallium 2 vs OCaml

- Despite less sophisticated unboxing strategy, OCaml matches and beats Gallium 2 in most tests.

- Floating-point tests (fft, nucleic) show that the local unboxing strategy of OCaml is just as effective as the more general strategy of Gallium 2.

- The only test (mandelbrot) where Gallium 2 is significantly faster is due to Gallium removing 2 levels of indirection while OCaml removes only 1.
</presentation>