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?
ML

- Polymorphism and type abstraction
- Compile-time type $\neq$ Run-time type

```
val triplicate : 'a -> 'a array
let triplicate x = [|x; x; x|]
```

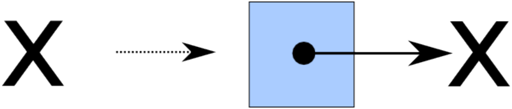
Why do we need boxing?
ML

- 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 ∼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?

```
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

```
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

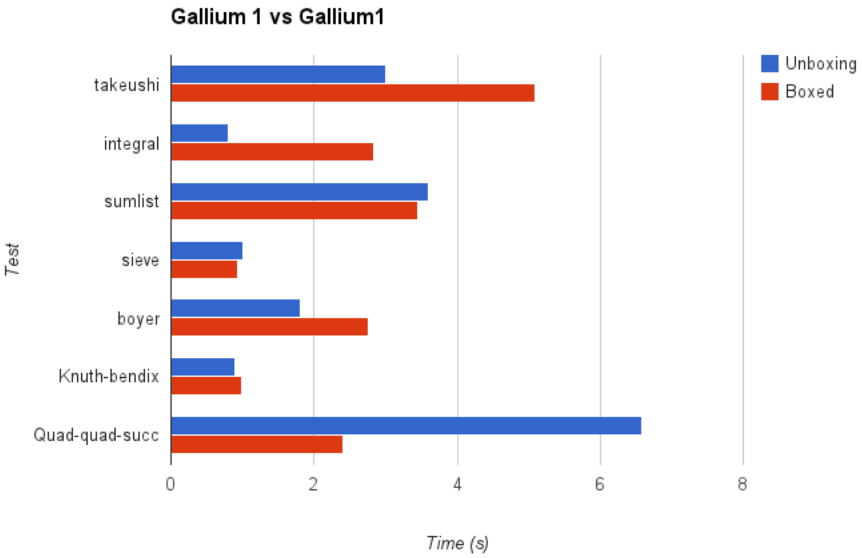
Match-ups

- 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

Test	Unboxing	Boxed	Test type
takeushi	**3.00**	5.09	fun calls, int arith
integral	**0.80**	2.83	float arith, loops
sumlist	3.60	**3.45**	lists, int arith
sieve	1.00	**0.94**	int arith, lists, polymorphism
boyer	**1.80**	2.76	fun calls, symbols
knuth-bendix	**0.90**	0.98	symbols, polymorphism
quad quad succ	6.58	**2.40**	Church numbers

Gallium 1 vs Gallium 1



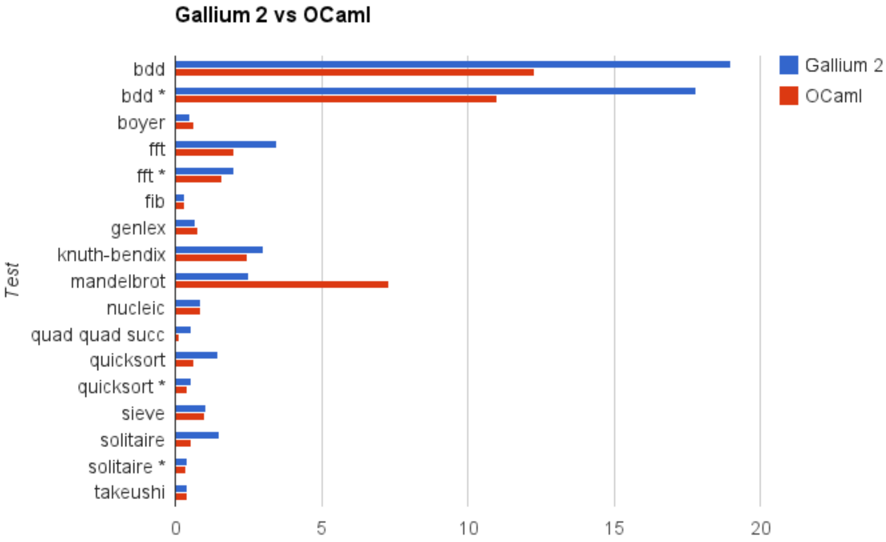
Gallium 1 vs Gallium 1

- 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

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

Gallium 2 vs OCaml



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⟩