Workshop "Types in Compilation", Amsterdam, June 1997

The effectiveness of type-based unboxing

Xavier Leroy*
INRIA Rocquencourt

Abstract

We compare the efficiency of type-based unboxing strategies with that of simpler, untyped unboxing optimizations, building on our practical experience with the Gallium and Objective Caml compilers. We find the untyped optimizations to perform as well on the best case and significantly better in the worst case.

1 Introduction

In Pascal or C, the actual types of all data are always known at compile-time, allowing the compilers to base data representation decisions on this typing information, thus supporting efficient memory layout of data structures as well as efficient calling conventions for functions.

This is no longer true for languages featuring polymorphism and type abstraction, such as ML: there, the static, compile-time type information does not always determine the actual, run-time type of a data (e.g. when the static type contains type variables or abstract type identifiers).

Hence, compilers for these languages often abandon C-style type-based data representations and revert to uniform, Lisp-style data representations, where all data structures fit a common format (usually, one word), if necessary by boxing (i.e. heap-allocating and handling through a pointer) data that does not naturally fit the common format.

However, the extra boxing involved can be quite expensive in terms of performance, and is a bottleneck in certain applications, especially numerical computation. To address this issue, a number of unboxing strategies for polymorphically-typed languages have been proposed: some rely on static typing information, just like C-style representation algorithms, but extended to cope with polymorphism and abstract types [9, 12, 6, 18, 16]; others rely on program analyses distinct from typing and apply equally well to untyped or dynamically-typed languages [3, 13].

If only core ML polymorphism is considered, a simpler alternative to these unboxing strategies is monomorphisation (duplicating polymorphic functions once for each instantiation type to obtain a monomorphic program). Experimental evidence [4, 11] suggests that monomorphisation does not result in major increase in code size, even though it remains delicate to implement efficiently in a separate compilation context. But the real challenge is with the SML module system, especially functors and type abstraction in structures, which results in large quantities of generic code (code that manipulates values whose representation types are statically unknown). Here, monomorphisation does not appear viable, leaving the unboxing strategies mentioned above as the only alternatives.

In this position paper, we build on our practical experience with unboxing strategies in the Gallium and Objective Caml compilers to assess the efficiency of type-based unboxing. We claim that while type-based unboxing strategies can be very effective on some monomorphic programs (e.g. numerical applications), they also add significant overhead to polymorphic programs and some monomorphic programs as well (e.g. symbolic computation). On the other hand, we found that untyped unboxing strategies can also achieve good performances on numerical applications, without penalizing symbolic computations. In other words, the best case for untyped strategies is almost as good as for type-based strategies, but the worst case is significantly better.

The remainder of this paper is organized as follows. Section 2 recalls the main type-based unboxing strategies proposed so far. Section 3 analyzes some of the overheads incurred by these strategies. Section 4 presents untyped unboxing optimizations that avoid these overheads. Section 5 discusses experimental results obtained with our Caml compilers, followed by concluding remarks in section 6.

2 Type-directed unboxing

Type-directed techniques for avoiding unnecessary boxing fall in three classes:

*Authors' address: projet Cristal, B.P.105, 78153 Le Chesnay, France. E-mail: `Xavier.Leroy@inria.fr`.

Coercions: In this approach, coercions between boxed and unboxed data representations are inserted at type specialization points, so that generic code always operates on boxed representations, while monomorphic code can take advantage of unboxed representations [6, 15]. This approach is particularly effective for supporting efficient, register-based calling conventions (with tuple arguments flattened and float arguments passed in float registers). Its main weakness is that it does not support deep unboxing inside generic data structures (e.g. lists or arrays with unboxed elements).

Run-time type inspection: Here, run-time representations of static typing information are maintained in the program, as extra arguments to polymorphic functions and extra components of structures defining abstract types; generic code then inspects those run-time type expressions to determine the locations and sizes of values with statically unknown types [18, 8]. Earlier proposals in the context of stack-based abstract machines [9, 10] pass only size information as extra parameters, instead of full type expressions. Unlike the coercion-based approach, this approach supports arbitrary unboxing inside data structures, but does not accommodate very well efficient register-based calling conventions for generic functions.

Tag-based unboxing: Tagging is a well-known technique for implementing dynamically-typed languages (Lisp, Smalltalk). It can be used as a special case of run-time type inspection, where type information is attached to data structures instead of being passed separately, and type expressions are mapped to a small set of base types, efficiently encoded at the bit level. Tagging supports only type inspection over types of existing values, and is best performed on large data structures, where the space overhead of storing the tag is negligible.

As an example of tag-based unboxing, we show how arrays are handled in the Gallium 2 and Objective Caml compilers. The run-time system supports two kinds of arrays: arrays of pointers and tagged integers, and arrays of unboxed floats. The two kinds have different tag bytes in the array header. Operations on arrays with a known type (τ `array` where τ is neither a type variable nor an abstract type) generate directly the correct code for accessing arrays of pointers or floats. Array operations with statically unknown type test the array tag at run-time, and if it is a float array, perform the required boxing and unboxing of floating-point numbers. This scheme supports fast operations over float arrays with known type, without the expense of extra type parameters, but at the cost of slower operations over generic arrays.

3 Overheads of type-directed unboxing

The overall goal of unboxing is to make program run faster by reducing the number of heap allocations and pointer dereferences. However, the unboxing techniques presented above also add extra run-time operations. The overheads of unboxing techniques must therefore be weighted carefully against the benefits. It must be kept in mind that the `load` operations eliminated by unboxing are relatively inexpensive operations on a modern processor with good memory hierarchy and possibly several load/store units. Heap allocations eliminated by unboxing represent more significant savings. Still, even small overheads can result in unboxing techniques being globally less efficient than no unboxing at all.

3.1 Extra operations introduced by unboxing

The first source of overhead is the extra operations introduced in the program code to implement the unboxing strategy.

Coercions: The extra coercions introduced by the unboxing strategy often introduce no overhead (the boxing and unboxing steps performed by the coercions would also be performed – at different times – by a systematic boxing strategy), but not always. In particular, coercions on functions involve extra function calls. Worse, some examples demonstrate a long sequence of successive unboxing and boxing of the same data before it is actually used [6].

Run-time type inspection: Propagating type information at run-time adds some overhead to polymorphic function calls: there are more arguments to pass, and more importantly some heap allocation is often performed to build the tree-shaped structures representing types at run-time. Even constant type expressions, entirely built at compile-time, entail the overhead of loading constant pointers in registers[1].

The second source of overhead is actually testing the run-time type information inside generic code. This can involve complex pattern-matching on the type expressions, resulting in additional loads and conditional branches, as well as a general increase in code size.

Several techniques have been proposed to reduce the overhead of type building or type inspection, but not both. Tolmach [19] uses an indirect representation of type expressions, reminiscent of explicit substitutions,

[1] With position-independent code, as is now standard on the Alpha and PowerPC, loading a constant pointer is not as cheap as it seems, since it is turned into a `load` and requires the global pointer to be properly set up.

so that only the variable parts of type expressions need to be passed at run-time. This greatly reduces run-time heap allocations of type expressions, but still requires arbitrary pattern-matching for type discrimination. Shao [16] uses hash-consing to reduce most type tests to simple pointer equality tests, but this makes type construction even more expensive.

Little experimental data on the actual cost of run-time type passing and type inspection has been published. The figures in [18] are not conclusive, since they are given for small test programs where all run-time type handling has been eliminated through aggressive inlining. Morrisett [8] reports slow-downs of 10% to 350% between single-module test programs (hence completely monomorphised by the compiler) and the same programs split in several separately-compiled modules (hence still containing run-time type tests). However, the slow-downs reported also include the cost of calling unknown functions instead of using pre-defined inlined operations. Even if its precise cost has not yet been determined, we believe that run-time type inspection can have a fairly high price on modern processors: it increases code size and introduces lots of conditional branches, which are difficult to schedule well and consume extra entries in the processor's branch prediction tables. Both phenomena favor code stalls, which are very expensive on modern processors.

Tag-based unboxing: Tagging shares some of the costs of run-time type inspection, but not all. Storing tags in newly-allocated blocks often has zero cost, since the tag can often be merged with GC information, which has to be maintained in any case. In Objective Caml, for instance, each heap block has a one-byte tag stored in the one-word header containing the block size and GC marking bits. There is no overhead on function calls, since no extra type parameters are introduced. Run-time tag tests are relatively inexpensive (one load and one integer comparison), never requiring pattern-matching on arbitrary trees. However, some of the drawbacks of run-time type inspection still apply to tag tests: increased code size and extra conditional branches. For instance, in Objective Caml 1.05, a polymorphic array copy function runs 10 times slower than the same function specialized to integer arrays, and 8 times slower than the same function specialized to float arrays.

3.2 GC overhead

The sources of overhead discussed so far apply only to generic code: fully monomorphic code pays no performance penalty. This is not the case for the GC-related overhead, discussed below, which affects all programs.

Unboxing strategies can add significant overhead to the traversal of the memory graph performed by the garbage collector. With a conventional, fully-boxed data representation strategy, walking the memory graph is relatively efficient: typically, one bit needs to be tested in every value to distinguish pointers from integers; then, for each pointer, one word of header must be consulted to determine the size of the block and whether it contains other pointers or just raw data. Most unboxing strategies complicate this traversal of the memory graph:

Getting the roots in the stack With unboxing, stack frames usually contain a mixture of valid pointers (or well-tagged integers) and raw, unboxed integers and floats. A non-conservative garbage collector needs to distinguish the pointers from the raw data. One possibility, used in Gallium and Objective Caml, is to associate frame descriptors, listing the locations of the pointers in the frame, to return addresses. Finding the descriptors associated with the return addresses in the stack frames and interpreting them adds some overhead compared with a fully boxed model where all stack words are valid pointers or tagged integers.

Handling mixtures of pointers and raw data in heap blocks Some unboxing strategies result in heap blocks that contain pointers intermingled with unboxed integers or floats. For instance, in Gallium, a heap block containing a value of type `string * float * int list` contains two pointers at byte offsets 0 and 12, separated by 8 bytes of raw data for the unboxed float. In this case, finding all the pointers contained in a block is no longer a yes/no question. The Gallium runtime system stores, in the block header, a pointer to a block descriptor enumerating the machine types (address, integer or float) of all block fields. The garbage collector then decodes that information to follow the pointers contained in the block. Despite various optimizations for frequent special cases (no pointers, all pointers, etc.), we found that this decoding of block descriptors accounts for a fairly large part of the time spent in garbage collection.

Type-directed garbage collection Some garbage collectors abandon tags and header words altogether, and base their traversal of the memory graph on static type information, using either run-time type parameters for polymorphic code [19] or GC-time type reconstruction [2, 1]. Here, determining where pointers lie inside blocks is even more expensive than in the previous case, since instead of reading pre-digested block descriptors, the garbage collector must interpret full type expressions. In addition, since type information is not attached to data, a Cheney-style breadth-first traversal of the memory graph is no longer feasible and must be replaced by a depth-first traversal [2, 19] or allocate extra heap memory for storing types. Tolmach [19] reports execution times ranging from 0.6 to 2.6 relative to a Gallium-style garbage collector on small programs. (Some of the extra cost of garbage collection is compensated by the fact that heap blocks are smaller – no extra header word is required). We believe more important slow-downs would be observed relative to a conventional, fully-tagged garbage collector, especially on larger programs.

In conclusion, the GC overhead of an unboxing strategy can be significant, and affects not only generic code, but fully monomorphic programs as well. This is especially bad for heavy symbolic processing (e.g. theorem proving), which is GC-intensive (it is not unusual to spend more than 30% of total running time in garbage collection), and does not benefit much from unboxing optimizations: most of the computation is performed on tree-shaped datatypes representing expressions, which remain fully boxed with all existing unboxing strategies. Symbolic processing often runs slower with aggressive unboxing optimizations than with a conventional, fully-boxed data representation, since the main code is identical but the garbage collector runs slower. In our opinion, this is not acceptable: symbolic processing, which is ML's bread and butter, should never run slower due to optimizations targeted towards hypothetical numerical or byte-oriented applications.

Recovering the efficiency of a conventional, fully-tagged garbage collector can be done in two directions. The first is to restrict the unboxing strategy so that it never produces heap blocks containing both pointers and unboxed data. For instance, a heap block containing a value of type `float * float` has the two floats unboxed (and is marked as "raw data" for the garbage collector), but a block containing a `float * string` holds two pointers, the float being allocated separately. Both SML/NJ and Objective Caml go even further and flatten only records of floating-point numbers, keeping everything else boxed inside heap blocks [15, 5].

The second direction is to allow mixed heap blocks, but group all pointers at the beginning of the block. The garbage collector is then instructed to follow the first N fields as pointers, with N possibly null or smaller than the actual size of the block. This greatly complicates access to block fields: a block of type τ_1 `*` τ_2, no longer contains a value of type τ_1 followed by a value of type τ_2; for instance, if $\tau_1 =$ `string * int` and $\tau_2 =$ `string`, the first component of the pair is composed of fields 0 and 2 of the block, while the second component is field 1. Also, there are practical difficulties with storing both a size and a pointer count in a one-word block header.

4 Untyped unboxing techniques

In addition to the type-directed unboxing techniques recalled in section 2, there also exists several unboxing techniques that use no or very little typing information, yet achieve most of the performance of type-based techniques, usually with a better worst-case behavior.

4.1 Local unboxing

Boxing and unboxing operations that cancel each other in the same function body are easily eliminated by a straightforward dataflow analysis. For instance, the following Objective Caml code

```
let f a x =
  let y = a.(0) *. x in y +. 1.0
```

performs only one float unboxing (on the `x` argument) and one float boxing (on the function result); the intermediate results remain unboxed. Also, the access to the array `a` performs neither unboxing nor type testing, since `a` is statically known to be a float array.

Trivial as it may seem, local unboxing is already very effective on numerical code, provided loops are not represented as tail-recursive functions in the intermediate language, but kept as part of the current function using special loop constructs in the intermediate language. For instance, the core of our FFT benchmark (see section 5) is composed of one fairly large function with four nested loops; local unboxing succeeds in eliminating *all* floating-point boxing and unboxing in this function, resulting in assembly code that looks very much like the one produced by a good C compiler.

Like all dataflow analyses, local unboxing can be extended to an inter-function analysis operating on whole compilation units, by combining it with a control-flow analysis (to determine the call graph) and an escape analysis (to determine data structures for which all creation and use sites are known). The Bigloo Scheme compiler performs unboxing of floats and float arrays this way, and achieves respectable performance on numerical code [13].

4.2 Known functions and partial inlining

A standard trick for making function calls with multiple arguments efficient in ML is to have two entry points per function: a standard entry point, using the regular calling conventions (take a heap-allocated tuple of arguments for uncurried function, or take one argument and return a closure for a curried function), and a fast entry point, taking all the arguments in registers. A direct call to the fast entry point is generated when the caller "knows" which function is being called (i.e. a control-flow analysis has determined that only one function flows to the call site) and provides exactly the expected number of arguments (no partial application). In all other cases (call to an unknown function, partial application, etc.), a regular call through the function's closure is generated, and the closure points to the standard entry point. The standard entry point can be a code prelude sequence, which dispatches the arguments to registers before falling through the fast entry point, or (to save code space) a shared combinator, which dispatches the arguments before tail-calling the fast entry point, stored in a conventional field of the closure.

For calling known functions taking a tuple of arguments, this scheme is essentially as efficient as unboxed tuples in a coercion-based unboxing scheme [6]. Unboxed tuples work better for calls to unknown functions with known types, but the multiple entry point scheme deals with curried functions equally well, while unboxing schemes are ineffective against currying.

Peyton-Jones and Launchbury [12] and independently Goubault [3] proposed an elegant reformulation of the multiple entry point trick as a partial inlining problem, which allows not only tuples of arguments to be unboxed, but also tuples of results, floating-point arguments and results, and possibly more. It is obvious that inlining a function at point of call and applying a local unboxing optimization gets rid of all unnecessary boxing of the function arguments and results. However, most functions are too large to be inlined. The solution is to decompose functions into three parts:

- a prelude that unboxes those arguments that need to be unboxed (as determined by the local unboxing analysis);
- a body that takes unboxed arguments and computes unboxed results;
- a postlude that boxes the results.

Then, the function is partially inlined at call sites where it can be determined that the function is the only one that flows to these call sites: the prelude and postlude are inlined, hopefully canceling the boxing and unboxing operations around the call site; the function body is not inlined, but simply called.

No experimental results have been published for the partial inlining approach to unboxing, but, based on our experience with multiple entry points for curried and uncurried functions in Objective Caml, we expect this scheme to be very effective for removing boxing and unboxing operations around function calls. The only potential problem is a certain growth in code size when the inlined preludes and postludes do not cancel cleanly with other operations around the call site. Also, inlining a postlude can prevent tail call optimization.

Of course, both multiple entry points and partial inlining apply only to calls to known functions. On our test suite for the Objective Caml compiler, 80% to 100% of all dynamically executed function calls are statically turned into direct calls to known functions. However, Objective Caml uses a very simple-minded control-flow analysis, comparable to the first iteration of the 0CFA algorithm [17, 14]; we expect that better control-flow analyses would lead to even better figures. Objective Caml's simple-minded control-flow analysis works quite well not only on the core ML language, but across structures and functors as well. We have not yet extended it to the object-oriented features of Objective Caml, however. It is likely that more sophisticated control-flow analyses are needed to recognize invocations of known methods.

5 Experimental results

We now discuss some experimental results obtained with the Gallium 1, Gallium 2 and Objective Caml compilers. Gallium 1 was the first implementation of the coercion-based type-directed unboxing presented in [6]. It generated code for the MIPS processor and had a simple, one-generation copying collector. Gallium 2, briefly described in [19], also uses coercion-based unboxing, but adds a better, more portable code generator, a two-generation copying collector, as well as tag-based unboxing of floats in arrays (as described in section 2). The Objective Caml native-code compiler [5, 7] abandons coercion-based unboxing and uses conventional, mostly-tagged data representations in combination with local unboxing of floats (as described in section 4.1), multiple entry points to uncurried and curried functions (section 4.2), and tag-based handling of unboxed float arrays (section 2). The garbage collector has two generations, using an incremental mark-and-sweep collector on the old generation. The main reason coercion-based unboxing was abandoned in Objective Caml is because of the GC overhead discussed in section 3.2, and also to allow more code sharing with the Objective Caml bytecode compiler.

The first series of experimental results are shown in figure 1. They compare the Gallium 1 compiler with type-directed unboxing versus a simple variant of the same compiler using conventional, fully boxed or tagged data representations. The intent was to compare unboxed and boxed representation strategies with all other things (code generator, garbage collector, etc) being equal. The compiler using boxed representations did not implement any kind of local unboxing nor optimizations for multiple-argument functions, though. The results are taken from [6].

As figure 1 shows, unboxing is most effective on programs that perform a lot of floating-point computation, such as `integral`, achieving speedups of 3 to 4. Integer computations (`sieve`, `sumlist`) run at about the

Test	Gallium 1 with unboxing	Gallium 1 no unboxing	What is tested
`takeushi`	3.00	5.09	function calls, integer arithmetic
`integral`	0.80	2.83	floating-point arithmetic, loops
`sumlist`	3.60	3.45	list processing, integer arithmetic
`sieve`	1.00	0.94	integer arithmetic, lists, functionals, polymorphism
`boyer`	1.80	2.76	term processing, function calls
`knuth-bendix`	0.90	0.98	term processing, functionals, polymorphism
`quad quad succ`	6.58	2.40	Church numerals, functionals, polymorphism

Times are given in seconds, averaged on three runs. The tests were conducted on a MIPS R3000-based Decstation 5000/200 running Ultrix 4.0.

Figure 1: Performance comparison between Gallium 1 with and without coercion-based type-directed unboxing

same speed, even though one compiler uses native 32-bit integer arithmetic, while the other uses tagged 31-bit integers (with n being represented as $2n+1$). Clearly, the overhead of maintaining the tag bit on integers is low, and probably even lower on a more modern processor with multiple integer units. The symbolic processing tests (`boyer`, `knuth-bendix`) show a slight performance advantage for the unboxing compiler, which we attribute to the fact that the calling conventions for uncurried functions with several arguments are more efficient in the unboxing compiler. This intuition is confirmed by the `takeushi` test, which measures essentially the speed of function calls with three arguments. The `quad quad succ` test, based on Church numerals, is the one known case where the unboxing and boxing coercions wrapped around polymorphic function actually perform a lot of unnecessary work, causing performances much worse than those of a fully-boxed implementation.

The second set of results (figure 2) pit the Gallium 2 compiler, with coercion-based type-directed unboxing, against the Objective Caml 1.05 native-code compiler, which uses mostly standard (tagged or boxed) data representations combined with a number of tricks for floats, float arrays, and multiple-argument functions. The comparison is not completely fair, since both compilers use slightly different code generators and garbage collectors.

As figure 2 shows, despite its inferior unboxing technology, Objective Caml matches the performances of Gallium 2 on most tests. Objective Caml is even slightly faster than Gallium 2 on some symbolic processing tests (`knuth-bendix`, `bdd`), a fact we attribute to the simpler heap traversal in the Objective Caml garbage collector, which, unlike Gallium's, does not have to deal with mixed pointers and raw data in heap blocks. On the other hand, the Objective Caml garbage collector is handicapped by the fact that the major heap is not contiguous (it grows on demand without copying), making it more expensive to determine which pointers point to the heap than in Gallium; this accounts for Gallium's better performance on `boyer`.

On floating-point tests (`fft`, `nucleic`), local unboxing of floats as in Objective Caml is just as effective as the more general unboxing strategy of Gallium 2. Integer tests (`fib`, `takeushi`, `sieve`, `solitaire`) show no significant differences, thus confirming that 63-bit tagged arithmetic is essentially as fast as 64-bit native arithmetic.

Tests involving arrays (`fft`, `quicksort`, `solitaire`, `bdd`) show a large performance advantage for Objective Caml. This is a consequence of much more efficient array bounds checking in Objective Caml. To compensate for this, we also give measurements with array bounds checking turned off (the starred tests in figure 2).

The only test where Objective Caml is noticeably slower is `mandelbrot`, which operates on references to floats. The Gallium compiler gets rid of the two levels of indirection (the reference, then the float), while the local unboxing algorithm of Objective Caml 1.05 eliminates only one level. This is to be construed as a bug in Objective Caml 1.05, which we expect to fix shortly.

6 Conclusions

Like all typing analyses, type-directed unboxing is highly systematic: all data having the same type must have the same representation. This leads to unboxing strategies that either unbox very little, as in SML/NJ, or unbox quite a lot but slow down the garbage collector and other parts of the runtime system, as in Gallium. We believe unboxing is best viewed as an optimization, in the classic compiler sense of the term: a transformation that can be applied or not on a case-by-case basis, without compromising correctness. Following this approach, we have found that a modest amount of type-directed unboxing (tag-based handling of unboxed float arrays and records of unboxed floats) combined with mostly-standard, untyped optimizations (local unbox-

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

ing, special calling protocols for known functions) performs just as well as, and even slightly better than aggressive coercion-based type-directed unboxing.

We conclude that unboxing is not the "killer app" for type-based compilation: good unboxing can be achieved without propagating type information through the whole compilation chain[2]. This is not to say that it's a bad idea to propagate types throughout an ML compiler; just that there must be other motivations to do so besides unboxing optimizations.

References

[1] S. Aditya, C. H. Flood, and J. E. Hick. Garbage collection for strongly-typed languages using run-time type reconstruction. In *Lisp and Functional Programming 1994*, pages 12–23. ACM Press, 1994.

[2] A. W. Appel. Run-time tags aren't necessary. *Lisp and Symbolic Computation*, 2(2), 1989.

[3] J. Goubault. Generalized boxing, congruences and partial inlining. In *Static Analysis Symposium '94*, number 864 in Lecture Notes in Computer Science, pages 147–161. Springer-Verlag, 1994.

[4] M. P. Jones. Partial evaluation for dictionary-free overloading. Technical Report YALEU/DCS/RR-959, Yale University, Dept. of Computer Science, Apr. 1993.

[5] X. Leroy, J. Vouillon, and D. Doligez. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[6] X. Leroy. Unboxed objects and polymorphic typing. In *19th symposium Principles of Programming Languages*, pages 177–188. ACM Press, 1992.

[7] X. Leroy. Le système Caml Special Light: modules et compilation efficace en Caml. Research report 2721, INRIA, Nov. 1995.

[8] G. Morrisett. *Compiling with types.* PhD thesis, Carnegie Mellon University, Dec. 1995.

[9] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Trans. Prog. Lang. Syst.*, 13(3), 1991.

[10] A. Ohori and T. Takamizawa. An unboxed operational semantics for ML polymorphism. *Lisp and Symbolic Computation*, 1997. To appear.

[11] D. P. Oliva and A. Tolmach. From ML to Ada(!?!): strongly-typed language interoperability via source

[2] The Objective Caml compiler exploits type information in the first compilation pass only – the one that goes from ML abstract syntax to the `lambda` intermediate code. All type-directed transformations are performed there. This greatly simplifies further compilation passes, which do not have to keep typing information up to date.

translation. Draft, available electronically, Nov. 1996.

[12] S. L. Peyton-Jones and J. Launchbury. Unboxed values as first-class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, 1991.

[13] M. Serrano and M. Feeley. Storage use analysis and its applications. In *International Conference on Functional Programming 1996*, pages 50–61. ACM Press, 1996.

[14] M. Serrano. Control flow analysis: a functional language compilation paradigm. In *Symposium on Applied Computing SAC '95*. ACM Press, 1995.

[15] Z. Shao and A. Appel. A type-based compiler for Standard ML. In *Programming Language Design and Implementation 1995*, pages 116–129. ACM Press, 1995.

[16] Z. Shao. Flexible representation analysis. In *International Conference on Functional Programming 1997*. ACM Press, 1997.

[17] O. Shivers. Control-flow analysis in Scheme. *SIGPLAN Notices*, 23(7):164–174, July 1988.

[18] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *Programming Language Design and Implementation 1996*, pages 181–192. ACM Press, 1996.

[19] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *Lisp and Functional Programming 1994*, pages 1–11. ACM Press, 1994.