Unboxed objects and polymorphic typing

Xavier Leroy

Proc. 19th Symp. Principles of Programming Languages, 1992, pages 177–188.

Unboxed objects and polymorphic typing

Xavier Leroy

Ecole Normale Supérieure and INRIA*

Abstract

This paper presents a program transformation that allows languages with polymorphic typing (e.g. ML) to be implemented with unboxed, multi-word data representations, more efficient than the conventional boxed representations. The transformation introduces coercions between various representations, based on a typing derivation. A prototype ML compiler utilizing this transformation demonstrates important speedups.

1 Introduction

It is common saying that statically-typed programs can be compiled more efficiently than dynamically-typed programs. A number of run-time type tests become unnecessary, for instance. In this paper, we study some compilation techniques that rely on the availability of typing information at compile-time. These techniques are connected to the data representation problem: how high-level objects manipulated by the language are mapped onto machine-level objects.

1.1 Static typing and data representation

There are many ways in which knowing type information at compile-time can help in selecting better representations for data. First of all, a compiler needs to know the size (required amount of memory) of all objects manipulated by the program, in order to allocate enough space for variable values and intermediate results, and to move the right number of bits from one location to another when performing bindings or assignments. Without static typing, a default size (usually one word) must be assumed for all objects in the program, and all representations must fit in this size. Objects that do not fit naturally in one word, such as records and 64-bit floating-point numbers, have to be boxed (allocated in the heap and handled through a pointer). With static typing, objects belonging to different types can have different sizes, as long as all objects of the same type have the same size. The compiler infers the size of an object from its type, and therefore knows how to allocate a variable of that type, or move a value of that type. This makes it possible to have unallocated objects larger than one word: unallocated 64-bit floating-point numbers, and unallocated records, in particular. The former are crucial for numerical-intensive programs. The latter are crucial for languages based on the λ-calculus, such as ML, where functions have only one argument and one result, and functions with several arguments or several results are encoded as functions taking or returning records.

A compiler may also use static typing information to indicate which register class is best suited for a given object. Most architectures distinguish floating-point registers from general-purpose registers; floating-point operations can only be performed between floating-point registers. When an object is statically known to have type `float`, it is possible to keep this object in a floating-point register, instead of a general-purpose register, so that computation on this object will not require register moves. The typical example of this type-based targeting is the use of different calling conventions for functions of different types: a function with result type `float` can be compiled to return its result in a given floating-point register, ready to be used by the caller; a function that returns an object of type `int` will use a given integer register instead. Static typing guarantees that the caller and the callee agree on the types of the arguments and the results, and therefore on their locations.

1.2 The problem with polymorphism

Type-directed compilation, as exemplified above, requires that each object manipulated by the program have one, unique, statically-known type. This holds with simple, monomorphic type systems such as Pascal. But this is not the case with more advanced type systems, notably those providing type abstraction or polymorphism [7]. With type abstraction, the concrete type of an object can remain unknown at compile-time.

*Author's address: INRIA Rocquencourt, projet Formel, B.P. 105, 78153 Le Chesnay, France. E-mail: `Xavier.Leroy@inria.fr`.

With polymorphism, an object can belong to several different types at the same time; again, its actual type is not available at compile-time. This paper focuses on polymorphism, and, more specifically, on the polymorphic type discipline of the ML language [12]. For instance, the ML identity function `function x → x` belongs to type $\alpha \to \alpha$ for all type expressions α. This type allows the function to be applied to objects of any type. Therefore, when compiling this function, we know neither the size of the argument nor the correct calling convention.

Several solutions to this issue have been considered. (See [13] for a survey.) The first one is to defer the compilation of a polymorphic object until it is actually used. At that time, we can compile a version of this object specialized to the type it is used with. This technique is often used for generics in Ada. It supports efficient data representations, but results in code duplication and loss of separate compilation.

A simpler solution is to assume a default size, common to all objects, and default calling conventions, common to all functions, just as if the language was not statically typed. Most existing ML implementations have taken this approach: they use one-word representations and uniform calling conventions[1] [6, 9, 4, 3]. This approach solves the problem of polymorphism, but results in a serious loss of efficiency. For instance, tuples are always heap-allocated, making passing several arguments to a function quite expensive. This efficiency loss is unfortunate, especially when large parts of a program are monomorphic (types are known at compile-time), as it is the case with most realistic ML programs.

1.3 Mixed representations

In this paper, we propose an alternate solution, mixed data representations, that relies on using different representations, boxed as well as unboxed, for the same high-level object, depending on the amount of type information available at compile-time. This solution is both efficient (monomorphic pieces of code are compiled with optimal data representations) and practical (polymorphic functions are compiled once and for all). It relies on introducing coercions between various data representations, based on a typing derivation for the program.

Some recent papers also consider mixing boxed and unboxed representations in the implementation of ML-like languages. Peyton-Jones [14] expresses many optimizations on boxing and unboxing as source-to-source transformations in this setting. To cope with polymorphism, a simple typing restriction is proposed: type variables in a polymorphic type can only be instantiated by "boxed" types (types whose values are boxed). The work presented here is complementary: our program transformation can be viewed as a translation from ML (unrestricted polymorphism) into Peyton-Jones' language (restricted polymorphism).

Morrison *et al.* have also used coercions between uniform and specialized representations in the implementation of their Napier88 language [13]. The coercion mechanism we use is similar to theirs, but offers one distinct advantage: whereas coercions in Napier88 involve interpreting type tags at run-time, ours may be entirely compiled, eliminating type information passing at run-time. We also provide a more formal framework to reason about coercion-based compilation techniques, and prove their correctness.

The remainder of this paper is organized as follows: section 2 informally explains mixed data representations and demonstrates the use of coercions. Section 3 formalizes these ideas, as a translation from the core ML language to the core ML language with restricted polymorphism and explicit coercions. We prove that the translation preserves the type and the semantics of the original program. Section 4 shows how the results above extend to concrete data types. Section 5 reports on an implementation of this technique in the Gallium high-performance ML compiler. We give a few concluding remarks in section 6.

2 Presentation

The approach taken in this paper is to mix two styles of data representation: specialized representations (multi-word objects and special calling conventions) when the static types are monomorphic; and uniform representations (one-word objects and default calling conventions) when the static types are polymorphic. Coercions between the two representation styles are performed when a polymorphic object is used with a more specific type. In the case of a polymorphic function, for instance, coercions take place just before the function call and just after the function return.

Polymorphic terms are compiled with the assumption that all terms whose static type is a type variable will be represented at run-time with a uniform representation. Hence the compiler knows their size (one word) and their calling protocol (the default one). Consider the following polymorphic function:

$$\texttt{let make\_pair} = \lambda \texttt{x}.\,(\texttt{x}, \texttt{x})$$

Its type is $\forall\alpha.\,\alpha \to \alpha \times \alpha$. Since x has static type α (a type variable), the compiler assumes the value of x fits in one word, and is passed in the default location:

[1] The New Jersey compiler utilizes better calling conventions when applying a known function [3]. However, this optimization does not work when calling functions that are passed as arguments, or defined in another compilation unit.

typically the first integer register. The returned value has static type $\alpha \times \alpha$. The compiler knows it is a pair, hence it produces code that returns an unallocated pair of one-word, uniformly represented values in the first two integer registers.

Consider now the application `make_pair`(3.14). Here, function `make_pair` is used with the more specific type `float` → `float` × `float`. The compiled code for this application evaluates the argument 3.14 with the specialized representation for objects of type `float`: an unallocated, two-word floating-point number in a floating-point register. Similarly, we expect as a result two unallocated floating-point numbers in two floating-point registers. These choices are not compatible with the hypotheses made when compiling `make_pair`. Therefore, it is not possible to call the code for function `make_pair` directly. The argument 3.14 must be first coerced to the uniform representation for floating-point numbers: the number is boxed, and a pointer to the box is passed to function `make_pair`. The value returned, an unallocated pair of two boxed floating-point numbers, must be coerced back to the specialized representation for pairs of floating-point numbers, by unboxing the two components of the pair.

To express this translation more formally, we introduce two operators: `wrap`(τ), the coercion from the specialized representation for objects of type τ to the uniform representation; and `unwrap`(τ), the reverse coercion, from the uniform representation to the specialized representation. Often, `wrap`(τ) will be implemented by boxing, and `unwrap`(τ) by unboxing. Better implementations can be considered for certain types τ, however; hence we stick to the more general terminology `wrap`-`unwrap`. We will often say that an object is "in the wrapped state", or "in the unwrapped state", to indicate how it is represented. With this notation, the compilation of `make_pair`(3.14) can be expressed as a translation to an expression with explicit coercions:

```
let x = make_pair(wrap(float)(3.14)) in
  (unwrap(float)(fst(x)), unwrap(float)(snd(x)))
```

followed by a conventional Pascal-like compilation that infers size and calling convention informations from the types.

The next example involves higher-order functions:

```
let map_pair = λf. λx. (f(fst(x)), f(snd(x)))
in map_pair(int_of_float)(3.14, 2.718)
```

The `map_pair` functional has type $\forall\alpha, \beta.\, (\alpha \to \beta) \to \alpha \times \alpha \to \beta \times \beta$. As explained above, it was compiled with the assumption that its parameter `f` is a function whose argument and result are in the wrapped state. The `int_of_float` primitive has type `float` → `int`, and therefore operates on unwrapped integer and floating-point numbers. Hence, the `map_pair` function cannot be applied directly to the `int_of_float` function: it must be given a version of `int_of_float` that operates on wrapped representations. This version is obtained by composing `int_of_float` with the right coercions:

```
λx. wrap(int)(int_of_float(unwrap(float)(x)))
```

This function is a suitable argument to the `map_pair` functional. The rest of the translation proceeds as in the previous example, resulting in:

```
let y = map_pair
  (λx. wrap(int)(int_of_float(unwrap(float)(x))))
  (wrap(float)(3.14), wrap(float)(2.718))
in (unwrap(int)(fst(y)), unwrap(int)(snd(y)))
```

The important point is that higher-order functions may require their functional arguments to be transformed to accommodate uniform representations instead of specialized representations. This transformation does not require recompilation of the functional, nor of the functional argument. It suffices to put some "stub code" around the functional argument, to perform the required coercions.

3 Formalization

In this section, we formally define the translation outlined above, in the context of the core ML language. We show that the translated program, evaluated with specialized data representations, computes the same thing as the original program, evaluated with uniform data representations.

3.1 The languages

The source language is the core ML language: λ-calculus with constants and the `let` construct. The only data structures are pairs. The target language is core ML extended with the two constructs `wrap`(τ) and `unwrap`(τ). The syntax for source terms (ranged over by a), target terms (a'), type expressions (τ), and type schemes (σ) is as follows:

$$
\begin{array}{rcl}
a & ::= & i \mid f \mid x \mid \lambda x.\, a \mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2 \mid a_1(a_2) \\
 & & \mid (a_1, a_2) \mid \texttt{fst}(a) \mid \texttt{snd}(a) \\
a' & ::= & i \mid f \mid x \mid \lambda x.\, a' \mid \texttt{let}\ x = a'_1\ \texttt{in}\ a'_2 \mid a'_1(a'_2) \\
 & & \mid (a'_1, a'_2) \mid \texttt{fst}(a') \mid \texttt{snd}(a') \\
 & & \mid \texttt{wrap}(\tau)(a') \mid \texttt{unwrap}(\tau)(a') \\
\tau & ::= & \alpha \mid \texttt{int} \mid \texttt{float} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \\
\sigma & ::= & \forall\alpha_1 \ldots \alpha_n.\, \tau
\end{array}
$$

Here and elsewhere, we write x for an identifier, i for an integer constant, f for a floating-point constant, and α for a type variable. Primitives are presented as predefined identifiers such as `add_float`.

To the source language we apply Milner's type discipline [10, 16]. We recall the typing rules below. They define the familiar predicate $E \vdash a : \tau$ ("under assumptions E, term a has type τ"). Here, E is a finite mapping from identifiers x to type schemes σ.

$$\frac{E(x) = \forall \alpha_1 \ldots \alpha_n . \tau \quad Dom(\rho) \subseteq \{\alpha_1 \ldots \alpha_n\}}{E \vdash x : \rho(\tau)}$$

$$E \vdash i : \texttt{int} \qquad E \vdash f : \texttt{float}$$

$$\frac{E + x : \tau_1 \vdash a : \tau_2}{E \vdash \lambda x.\, a : \tau_1 \to \tau_2}$$

$$\frac{E \vdash a_2 : \tau_1 \to \tau_2 \quad E \vdash a_1 : \tau_1}{E \vdash a_2(a_1) : \tau_2}$$

$$\frac{E \vdash a_1 : \tau_1 \quad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2}$$

$$\frac{E \vdash a : \tau_1 \times \tau_2}{E \vdash \texttt{fst}(a) : \tau_1} \qquad \frac{E \vdash a : \tau_1 \times \tau_2 \Rightarrow a'}{E \vdash \texttt{snd}(a) : \tau_2}$$

$$\frac{E \vdash a_1 : \tau_1 \quad E + x : Gen(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

In the last rule above, we write $Gen(\tau, E)$ for the type scheme generalizing τ in environment E. It is defined by

$$Gen(\tau, E) = \forall \alpha_1 \ldots \alpha_n . \tau$$

where $\alpha_1 \ldots \alpha_n$ are the type variables free in τ but not in E.

3.2 The translation

The translation of a term from the source language to the target language is based on the types given to the term and to its subterms. More precisely, we define the translation on a typing derivation for the given term. (In an actual compiler, this derivation would be the principal typing derivation for the term.) The translation is presented as the predicate $E \vdash a : \tau \Rightarrow a'$, where $E \vdash a : \tau$ is the typing predicate defined above, and the fourth component a' (a term of the target calculus) is the translation for a. This proposition is defined by a set of inference rules, with the same structure as the typing rules. The rules are given in figure 1.

Most rules propagate the translated terms in the obvious way. The hard work is performed by the S_ρ function (ρ is a substitution of types for type variables) in the rule for variable specialization. Given a target term a' represented according to type τ, function S_ρ is responsible for inserting the right `wrap` and `unwrap` coercions to transform it into a term represented according to type $\rho(\tau)$. The S transformation is defined as follows:

$$\begin{array}{rcl}
S_\rho(a' : \alpha) & = & \texttt{unwrap}(\rho(\alpha))(a') \\
S_\rho(a' : \texttt{int}) & = & a' \\
S_\rho(a' : \texttt{float}) & = & a' \\
S_\rho(a' : \tau_1 \times \tau_2) & = & \texttt{let x} = a' \texttt{ in} \\
 & & (S_\rho(\texttt{fst(x)} : \tau_1), S_\rho(\texttt{snd(x)} : \tau_2)) \\
S_\rho(a' : \tau_1 \to \tau_2) & = & \lambda x.\, S_\rho(a'(G_\rho(x : \tau_1)) : \tau_2) \\
 & & \text{where } x \text{ is not free in } a'
\end{array}$$

We also need to define the dual transformation G:

$$\begin{array}{rcl}
G_\rho(a' : \alpha) & = & \texttt{wrap}(\rho(\alpha))(a') \\
G_\rho(a' : \texttt{int}) & = & a' \\
G_\rho(a' : \texttt{float}) & = & a' \\
G_\rho(a' : \tau_1 \times \tau_2) & = & \texttt{let x} = a' \texttt{ in} \\
 & & (G_\rho(\texttt{fst(x)} : \tau_1), G_\rho(\texttt{snd(x)} : \tau_2)) \\
G_\rho(a' : \tau_1 \to \tau_2) & = & \lambda x.\, G_\rho(a'(S_\rho(x : \tau_1)) : \tau_2) \\
 & & \text{where } x \text{ is not free in } a'
\end{array}$$

The term a' given to the S transformation has been compiled assuming uniform representations for all data of static type α, for each type variable α. When a' is considered with type $\rho(\tau)$, the context expects these data of static type α to have the same (specialized) representations as data of static type $\rho(\alpha)$. Therefore, the goal of transformation S is to locate all data of static type α in term a', and apply the $\texttt{unwrap}(\rho(\alpha))$ coercion to them.

The transformation proceeds recursively over τ, the principal type for a'. When τ is a type variable α, it simply applies $\texttt{unwrap}(\rho(\alpha))$ to a'. When τ is an atomic type nothing needs to be done, since $\rho(\tau) = \tau$. When τ is a product type $\tau_1 \times \tau_2$, the two components of a', $\texttt{fst}(a')$ and $\texttt{snd}(a')$, are recursively transformed, and the two results are paired together. Finally, when τ is a function type $\tau_1 \to \tau_2$, the transformation returns a function that translates its argument y with type τ_1, applies a' to it, and translates the result with type τ_2. The processing of the argument requires a different transformation, G, instead of S, because of the contravariance of the arrow type constructor. In other words, the translated function $S_\rho(a' : \tau_1 \to \tau_2)$ should be applicable to data of type $\rho(\tau_1)$, using the specialized representation for components of type α; before applying a' to it, it is necessary to switch to uniform representation for these components. This is performed by the dual transformation G, defined exactly as S, except that in case $\tau = \alpha$, the coercion $\texttt{wrap}(\rho(\alpha))$ is used instead of $\texttt{unwrap}(\rho(\alpha))$.

Working out the example `make_pair`(3.14) above, we get the following derivation, where E is the typing environment $\texttt{make\_pair} \leftarrow \forall \alpha.\, \alpha \to \alpha \times \alpha$ and ρ is the sub-

$$\frac{E(x) = \forall\alpha_1 \ldots \alpha_n.\tau \quad Dom(\rho) \subseteq \{\alpha_1 \ldots \alpha_n\}}{E \vdash x : \rho(\tau) \Rightarrow S_\rho(x : \tau)} \qquad E \vdash i : \texttt{int} \Rightarrow i \qquad E \vdash f : \texttt{float} \Rightarrow f$$

$$\frac{E + x : \tau_1 \vdash a : \tau_2 \Rightarrow a'}{E \vdash \lambda x.\, a : \tau_1 \to \tau_2 \Rightarrow \lambda x.\, a'} \qquad \frac{E \vdash a_2 : \tau_1 \to \tau_2 \Rightarrow a_2' \quad E \vdash a_1 : \tau_1 \Rightarrow a_1'}{E \vdash a_2(a_1) : \tau_2 \Rightarrow a_2'(a_1')}$$

$$\frac{E \vdash a_1 : \tau_1 \Rightarrow a_1' \quad E + x : Gen(\tau_1, E) \vdash a_2 : \tau_2 \Rightarrow a_2'}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2 \Rightarrow \texttt{let } x = a_1' \texttt{ in } a_2'}$$

$$\frac{E \vdash a_1 : \tau_1 \Rightarrow a_1' \quad E \vdash a_2 : \tau_2 \Rightarrow a_2'}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2 \Rightarrow (a_1', a_2')} \qquad \frac{E \vdash a : \tau_1 \times \tau_2 \Rightarrow a'}{E \vdash \texttt{fst}(a) : \tau_1 \Rightarrow \texttt{fst}(a')} \qquad \frac{E \vdash a : \tau_1 \times \tau_2 \Rightarrow a'}{E \vdash \texttt{snd}(a) : \tau_2 \Rightarrow \texttt{snd}(a')}$$

Figure 1: The translation rules

stitution $\alpha \leftarrow \texttt{float}$.

$$\frac{\begin{array}{l} E \vdash \texttt{make\_pair} : \texttt{float} \to \texttt{float} \times \texttt{float} \\ \qquad\qquad \Rightarrow S_\rho(\texttt{make\_pair} : \alpha \to \alpha \times \alpha) \\ E \vdash 3.14 : \texttt{float} \Rightarrow 3.14 \end{array}}{\begin{array}{l} E \vdash \texttt{make\_pair}(3.14) : \texttt{float} \times \texttt{float} \\ \qquad\qquad \Rightarrow S_\rho(\texttt{make\_pair} : \alpha \to \alpha \times \alpha)(3.14) \end{array}}$$

By definition of S, we have:

$$\begin{array}{ll} & S_\rho(\texttt{make\_pair} : \alpha \to \alpha \times \alpha) \\ = & \lambda\texttt{x}.\, S_\rho(\texttt{make\_pair}(G_\rho(\texttt{x} : \texttt{float})) \\ & \qquad\qquad : \texttt{float} \times \texttt{float}) \\ = & \lambda\texttt{x}.\ \texttt{let y} = \texttt{make\_pair}(\texttt{wrap}(\texttt{float})(\texttt{x}))\ \texttt{in} \\ & \qquad (\texttt{unwrap}(\texttt{float})(\texttt{fst}(\texttt{y})), \\ & \qquad\ \texttt{unwrap}(\texttt{float})(\texttt{snd}(\texttt{y}))) \end{array}$$

After performing the β-reduction $\texttt{x} = 3.14$ at compile-time, we get the intuitive translation given in section 2. The translation often introduces many redexes that can be reduced at compile-time.

3.3 Type correctness of the translation

In this section, we show that the translation defined above does not introduce type errors: the resulting target term is well-typed. The target language is equipped with a variant of the type system for the source language. (We write $\vdash'$ for the typing judgements of the target language, instead of $\vdash$.) There are two differences. The first one is the explicit mention of wrapping and unwrapping at the level of types: we introduce a new kind of type expression, $[\tau]$, that represents the type of all wrapped values of type τ. Conversely, other kinds of type expressions, such as $\texttt{float}$ or $\tau_1 \times \tau_2$, now stand for unwrapped values of these types. We write τ' for the extended type expressions:

$$\tau' ::= \alpha \mid \texttt{int} \mid \texttt{float} \mid \tau_1' \to \tau_2' \mid \tau_1' \times \tau_2' \mid [\tau']$$

Then, the typing rules for the `wrap` and `unwrap` constructs are, obviously:

$$\frac{E' \vdash' a' : \tau}{E' \vdash' \texttt{wrap}(\tau)(a') : [\tau]} \qquad \frac{E' \vdash' a' : [\tau]}{E' \vdash' \texttt{unwrap}(\tau)(a') : \tau}$$

The second difference is the restriction of polymorphism. In the source language, a type variable α universally quantified in a type scheme can be substituted by any type expression. In the target language, we only allow substitution by a "wrapped type", that is, a type of the form $[\tau]$. This restriction reflects the fact that, at compile-time, objects whose type is a type variable are assumed to be in the wrapped state. If type variables can only be instantiated by wrapped types, then the assumption above holds for all well-typed target terms. To implement this restriction on polymorphism, we change the typing rule for variables to:

$$\frac{E'(x) = \forall\alpha_1 \ldots \alpha_n.\tau \quad Dom(\rho) \subseteq \{\alpha_1 \ldots \alpha_n\}}{E' \vdash' x : [\rho](\tau)}$$

Here, the substitution $[\rho]$ is the substitution defined by $[\rho](\alpha) = [\rho(\alpha)]$ for all $\alpha \in Dom(\rho)$. Any substitution of wrapped types for type variables has the the form $[\rho]$ for some ρ.

We can now state the correctness of the translation with respect to the type systems.

Proposition 1 *If $E \vdash a : \tau \Rightarrow a'$, then $E \vdash' a' : \tau$.*

Proof: the proof requires the following lemma.

Lemma 1

1. *If $E' \vdash' a' : [\rho](\tau')$, then $E' \vdash' S_\rho(a' : \tau') : \rho(\tau')$.*
2. *If $E' \vdash' a' : \rho(\tau')$, then $E' \vdash' G_\rho(a' : \tau') : [\rho](\tau')$.*

Proof: by induction over τ'. In case $\tau' = \alpha$, (1) is the typing rule for `unwrap`, and (2) is the typing rule for `wrap`. □

Proposition 1 follows from a simple inductive argument on the translation derivation. The only interesting case is $a = x$. Then, the translation is:

$$\frac{E(x) = \forall\alpha_1 \ldots \alpha_n.\tau \quad Dom(\rho) \subseteq \{\alpha_1 \ldots \alpha_n\}}{E \vdash x : \rho(\tau) \Rightarrow S_\rho(x : \tau)}$$

In the type system for the target language, we have $E \vdash' x : [\rho](\tau)$. By case 1 of lemma 1, we conclude that $E \vdash' S_\rho(x : \tau) : \rho(\tau)$, as expected. □

3.4 Operational semantics

In this section, we give operational semantics for the source language and for the target language, in preparation for a proof of the semantic correctness of the translation. We define two evaluation predicates, mapping terms to values, in the style of [12, 16]. The syntax for values is:

$$\begin{array}{rcl} v & ::= & i \mid f \mid \langle v \rangle \mid v_1, v_2 \mid \texttt{clos}(k, \lambda x.\, a, e) \mid op \\ op & ::= & \texttt{add} \mid \ldots \end{array}$$

A value is either a constant; a pointer $\langle v \rangle$ to a heap cell containing value v; an (unallocated) pair of two values v_1, v_2; a primitive operation op; or a closure $\texttt{clos}(k, \lambda x.\, a, e)$ of function $\lambda x.\, a$ by evaluation environment e, with k being the expected size of the function argument (see below). Evaluation environments e are finite mappings from variables to values. This definition for values makes the boxing steps explicit (they are usually left implicit in this kind of semantics), and provides for the fact that values may have different sizes. To be more specific, we assume the following typical size assignment:

$$\begin{array}{cccc} \|i\| = 1 & \|f\| = 2 & \|\langle v \rangle\| = 1 & \|op\| = 2 \\ \|v_1, v_2\| = \|v_1\| + \|v_2\| & & \|\texttt{clos}(k, \lambda x.\, a, e)\| = 2. & \end{array}$$

(We consider closures as two pointers, one to the code part, one to the environment part). We also associate a size to type expressions accordingly:

$$\begin{array}{cccc} \|\texttt{int}\| = 1 & \|\texttt{float}\| = 2 & \|\tau_1' \to \tau_2'\| = 2 & \\ \|\tau_1' \times \tau_2'\| = \|\tau_1'\| + \|\tau_2'\| & \|[\tau']\| = 1 & \|\alpha\| = 1. & \end{array}$$

The semantics are given in figure 2. The semantics for the source language uses uniform data representations: all terms are mapped onto values of size one. For instance, the floating-point number f is represented as $\langle f \rangle$; the pair of two terms, the first evaluating to v_1, the other to v_2, is represented as $\langle v_1, v_2 \rangle$; and closures are boxed, too.

For the target language, we use specialized representations: floating-point numbers, pairs, and closures are left unallocated. Notice the appearance of a new kind of run-time type errors: applying a closure to a value of the wrong size. Coercions $\texttt{wrap}(\tau)$ and $\texttt{unwrap}(\tau)$ are implemented as boxing and unboxing for types of size greater than one, and as no-ops for types of size one.

3.5 Semantic correctness of the translation

In this section, we show that the translation preserves semantics: the translated program (evaluated with mixed representations) computes the same thing as the original program (evaluated with uniform representations).

There is a slight difficulty here: the semantics may assign different values to the two programs, because one object may have different representations in the two semantics. For instance, in the case of a term with type `float`, the translation is correct if and only if whenever the term evaluates to $\langle f \rangle$, then its translation evaluates to f.

Hence we need to define a notion of equivalence between two values, one corresponding to uniform representations, the other corresponding to mixed representations. Actually, the equivalence is defined between typed values (value-type pairs). We write it $\Gamma \models v : \tau \approx v' : \tau'$. Types are needed here to correctly interpret the values, and to ensure the well-foundedness of the definition.

The environment Γ provides an interpretation for type variables in τ and τ'. Legal interpretations for type variables are non-empty sets $\mathcal{V}$ of pairs of values, such that for all $(v, v') \in \mathcal{V}$, we have $\|v'\| = 1$. This restriction over the size of v' captures the fact that whenever values are considered with type α, they must use default representations.

The definition of the semantic equivalence is modeled after the "semantic typing" relations used in some proofs of soundness for type systems [16].

- $\Gamma \models v : \alpha \approx v' : \alpha$ iff $(v, v') \in \Gamma(\alpha)$
- $\Gamma \models i : \texttt{int} \approx i : \texttt{int}$
- $\Gamma \models \langle f \rangle : \texttt{float} \approx f : \texttt{float}$
- $\Gamma \models v : \tau \approx v' : [\tau']$, where $\|\tau'\| = 1$, iff $\Gamma \models v : \tau \approx v' : \tau'$
- $\Gamma \models v : \tau \approx \langle v' \rangle : [\tau']$, where $\|\tau'\| > 1$, iff $\Gamma \models v : \tau \approx v' : \tau'$
- $\Gamma \models \langle v_1, v_2 \rangle : \tau_1 \times \tau_2 \approx v_1, v_2 : \tau_1' \times \tau_2'$ iff $\Gamma \models v_1 : \tau_1 \approx v_1' : \tau_1'$ and $\Gamma \models v_2 : \tau_2 \approx v_2' : \tau_2'$
- $\Gamma \models \langle \texttt{clos}(1, \lambda x.a, e) \rangle : \tau_1 \to \tau_2 \approx \texttt{clos}(k, \lambda x'.a', e') : \tau_1' \to \tau_2'$ iff $\|\tau_1'\| = k$, and for all values v_1, v_2, v_1' such that $\Gamma \models v_1 : \tau_1 \approx v_1' : \tau_1'$ and $e + x \leftarrow v_1 \vdash a \xrightarrow{u} v_2$, there exists a value v_2' such that $e' + x' \leftarrow v_1' \vdash a' \xrightarrow{m} v_2'$ and $\Gamma \models v_2 : \tau_2 \approx v_2' : \tau_2'$.

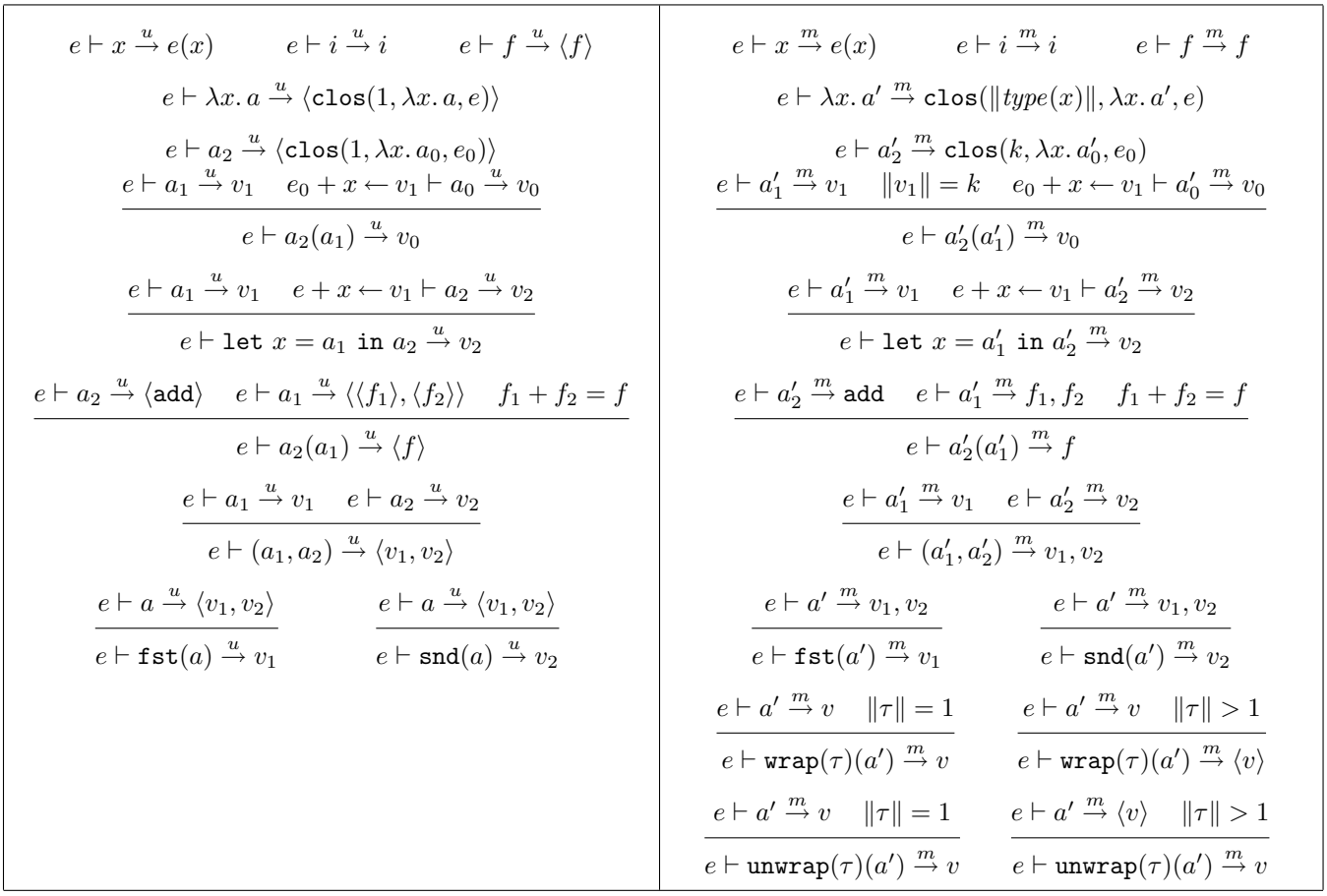


Figure 2: Operational semantics (left: uniform representations; right: mixed representations)

The equivalence relation extends to type schemes, and to environments:

- $\Gamma \models v : \forall\alpha_1 \ldots \alpha_n.\, \tau \approx v' : \forall\alpha_1 \ldots \alpha_n.\, \tau'$ iff for all legal interpretations $\mathcal{V}_1 \ldots \mathcal{V}_n$ for type variables $\alpha_1 \ldots \alpha_n$, we have

$$\Gamma + \alpha_1 \leftarrow \mathcal{V}_1 + \ldots + \alpha_n \leftarrow \mathcal{V}_n \models v : \tau \approx v' : \tau'$$

- $\Gamma \models e : E \approx e' : E'$ iff the domains of e, E, e', E' are the same, and for all $x \in Dom(e)$, we have $\Gamma \models e(x) : E(x) \approx e'(x) : E'(x)$.

We can now state the semantic correctness of the translation:

Proposition 2 *Assume that:*

$$E \vdash a : \tau \Rightarrow a', \quad \Gamma \models e : E \approx e' : E, \quad e \vdash a \xrightarrow{u} v.$$

Then, there exists a value v' such that:

$$e' \vdash a' \xrightarrow{m} v' \quad \textit{and} \quad \Gamma \models v : \tau \approx v' : \tau.$$

Proof: the proof makes use of the results below.

Lemma 2 *Let ρ be the substitution $\{\alpha_1 \leftarrow \tau_1 \ldots \alpha_n \leftarrow \tau_n\}$. Define $\mathcal{V}_i = \{(v, v') \mid \Gamma \models v : \tau_i \approx v' : [\tau_i]\}$. Then, the $\mathcal{V}_i$ are legal interpretations for the α_i. And the following two results are equivalent:*

$$\Gamma + \alpha_1 \leftarrow \mathcal{V}_1 + \ldots + \alpha_n \leftarrow \mathcal{V}_n \models v : \tau \approx v' : \tau \quad (1)$$
$$\Gamma \models v : \rho(\tau) \approx v' : [\rho](\tau) \quad (2)$$

Proof: by induction over τ. □

Lemma 3

1. *If $e' \vdash a' \xrightarrow{m} v'$ and $\Gamma \models v : \rho(\tau) \approx v' : [\rho](\tau)$, then there exists v'' such that $e' \vdash S_\rho(a' : \tau) \xrightarrow{m} v''$ and $\Gamma \models v : \rho(\tau) \approx v'' : \rho(\tau)$.*

2. *If $e' \vdash a' \xrightarrow{m} v'$ and $\Gamma \models v : \rho(\tau) \approx v' : \rho(\tau)$, then there exists v'' such that $e' \vdash G_\rho(a' : \tau) \xrightarrow{m} v''$ and $\Gamma \models v : \rho(\tau) \approx v'' : [\rho](\tau)$.*

Proof: by induction over τ'. □

The proof of proposition 2 itself is a simple inductive argument on the translation derivation. The only interesting case is $a = x$. Then, the translation derivation is:

$$\frac{E(x) = \forall\alpha_1 \ldots \alpha_n.\tau \quad Dom(\rho) \subseteq \{\alpha_1 \ldots \alpha_n\}}{E \vdash x : \rho(\tau) \Rightarrow S_\rho(x : \tau)}$$

The only evaluation possibility is $v = e(x)$. By hypothesis,

$$\Gamma \models e(x) : \forall\alpha_1 \ldots \alpha_n.\tau \approx e'(x) : \forall\alpha_1 \ldots \alpha_n.\tau.$$

By definition of $\models$ on type schemes, and by lemma 2, we have:

$$\Gamma \models e(x) : \rho(\tau) \approx e'(x) : [\rho](\tau).$$

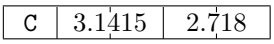
Then, the expected result follows from lemma 3 (case 1), taking $a' = x$ and $v' = e'(x)$. □

4 Concrete data types

Until now, we have only dealt with simple data structures such as tuples and records. This section discusses more complex data structures: ML concrete data types.

Except in degenerate cases, values belonging to concrete data types are best kept boxed at all times, in the unwrapped state as well as in the wrapped state. This conclusion can be drawn separately from two features of the concrete data types: they are sum types; and they can be recursive. Since data types are sum types, we do not know the exact size for values of these types, only an upper bound. Keeping these values unallocated would waste resources (e.g. registers). Since data types can be recursive, values of these types cannot be allocated completely flatly: sub-components of the same type must be handled through pointers.

Therefore, data types are represented by a heap block containing the constructor tag and the argument to the constructor, as usual. However, specialized representations result in a layout of the constructor argument that is flatter than usual, and therefore more space- and time-efficient. For instance, assuming the constructor declaration `C of float` $\times$ `float`, the value `C`$(3.1415, 2.718)$ is represented by the 5-word block:

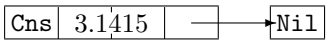


This optimized layout is natural when we use mixed representations. We statically know that the constructor argument is of type `float` $\times$ `float`. Hence the constructor argument is evaluated as an unallocated pair of two unallocated numbers. When applying constructor `C` to this argument, the components of the argument are not boxed yet, and we are free to choose the most compact memory layout for them.

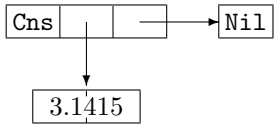
ML data types can be parameterized by other types, as the familiar `list` type:

$$\texttt{datatype } \alpha \texttt{ list} = \texttt{Nil} \mid \texttt{Cns of } \alpha \times \alpha \texttt{ list}$$

This raises a subtle issue. If we naively follow the approach above, a list of floating-point numbers `Cns`$(3.1415, $`Nil`$)$, with static type `float list`, is represented with the numbers unboxed, as follows.



However, generic functions over lists, such as the `length` function, are compiled without knowing the exact type of the list elements, and therefore they assume wrapped representations for the list elements. Hence, before being passed to a generic function over lists, the list above must be coerced to:



That is, to coerce a τ `list` to an α `list`, we would have to apply coercion `wrap`(τ) to each list element. More generally, transformation S would be defined on lists as:

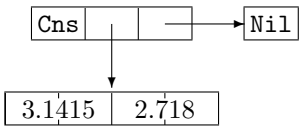
$$S_\rho(a' : \tau \texttt{ list}) = \texttt{map } (\lambda x.\, S_\rho(x : \tau))\ a'.$$

This operation requires time and space proportional to the length of the list, making this approach clearly impractical. To avoid copying, we must require that list elements are in the wrapped state at all times, even if their type is statically known. In other words, all list cells must share the same layout, with only one word allocated for the list element (last format above). This layout is determined once and for all when the `list` type is defined, assuming wrapped representations for the components of type α (the type parameter). Then, nothing needs to be done when specializing or generalizing a list:

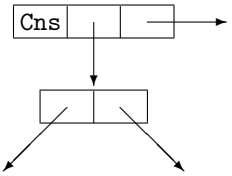
$$S_\rho(a' : \tau \texttt{ list}) = a'.$$

Instead, some wrapping and unwrapping steps are required when constructing or accessing lists. To insert them correctly, it suffices to consider constructors and accessors as polymorphic functions that are used with more specific types: the translation given above inserts the right coercions.

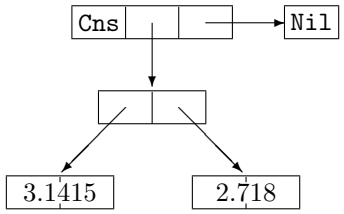
To correctly handle lists and other generic data structures, it is not enough to keep list components in the wrapped state. We must also impose some additional compatibility conditions on the `wrap` and `unwrap` transformations. Namely, we must ensure that the components of a wrapped object are themselves in the wrapped state. This is what we call "recursive wrapping". Consider the list `l` $=$ `Cns`$((3.1415, 2.718), $`Nil`$)$, with type $($`float` $\times$ `float`$)$ `list`. If we don't perform recursive wrapping, the natural representation for list `l` is:



This object is not a suitable argument to a function `f` with type $\forall\alpha.\,(\alpha \times \alpha)$ `list` $\rightarrow \ldots$ Such a function assumes its list argument to be of the format:



And no coercion will take place on `l` before it is passed to `f`, since `l` is a list. Hence, the correct representation for `l` is:



This means that the correct wrapped representation for a pair of floating-point numbers is a boxed pair of two boxed floating-point numbers. Similarly, the wrapped representation for a function with type `float` $\rightarrow$ `float` is not a boxed closure of the original function on unwrapped numbers, but a boxed closure of the corresponding function on wrapped numbers. More generally, the wrapped representation for an object of type τ must be compatible with the wrapped representation for objects of type τ', for all types τ' more general than τ. To this end, we redefine the `wrap` and `unwrap` coercions on product types and on function types in terms of the S and G translations:

$$\begin{aligned}
&\texttt{wrap}(\tau_1 \times \tau_2)(a') = \\
&\quad \texttt{wrap}(\times)(G_{\{\alpha\leftarrow\tau_1,\beta\leftarrow\tau_2\}}(a' : \alpha \times \beta)) \\
&\texttt{wrap}(\tau_1 \rightarrow \tau_2)(a') = \\
&\quad \texttt{wrap}(\rightarrow)(G_{\{\alpha\leftarrow\tau_1,\beta\leftarrow\tau_2\}}(a' : \alpha \rightarrow \beta)) \\
&\texttt{unwrap}(\tau_1 \times \tau_2)(a') = \\
&\quad \texttt{unwrap}(\times)(S_{\{\alpha\leftarrow\tau_1,\beta\leftarrow\tau_2\}}(a' : \alpha \times \beta)) \\
&\texttt{unwrap}(\tau_1 \rightarrow \tau_2)(a') = \\
&\quad \texttt{unwrap}(\rightarrow)(S_{\{\alpha\leftarrow\tau_1,\beta\leftarrow\tau_2\}}(a' : \alpha \rightarrow \beta))
\end{aligned}$$

Here, we have introduced primitive coercions `wrap` and `unwrap` that are attached to the type constructors $\times$ and $\rightarrow$ themselves, and no more to product types and to function types. These coercions can be implemented arbitrarily, for instance by boxing and unboxing.

5 Application to ML

The author has implemented the ideas above in Gallium, a prototype high-performance compiler for the ML language.

5.1 Representations for ML data types

We first describe the exact data representations used. Tuples, records, and floating-point numbers are represented as described above: unboxed in the unwrapped state, boxed in the wrapped state. Since moving around a large object is expensive, it would be wise to limit the size of an unboxed tuple. We could decide that any tuple requiring more than 4 words, for instance, is always boxed. This can be determined from the type of the tuple.

Concrete data type values are kept boxed at all times, as explained in section 4.

Unwrapped integers are unboxed, 32-bit wide. Even though they fit in one word, they have to be boxed in the wrapped state, for garbage collection reasons (see below). The `wrap(int)` operation is therefore implemented as boxing, and `unwrap(int)` as unboxing. An alternative would be tagged, unboxed, 31-bit wide integers both for the unwrapped state and the wrapped state; this would reduce heap allocation, but arithmetic on tagged integers is slower, and interface with C functions is complicated.

The garbage collector allows small, 8-bit wide integers to remain unboxed in the wrapped state. These small integers are used to encode booleans, as well as a built-in `char` type. The `wrap` and `unwrap` operations for these types are no-ops. As an easy generalization of the case for booleans, small integers could also represent enumerated data types (concrete types with constant constructors only).

The only value of type `unit`, `()`, is represented in the unwrapped state as the 0-tuple — the absence of any value, actually. Wrapped values of type `unit` are represented as a given one-word constant. The `wrap(unit)`(a) operation consists in evaluating a and loading the constant; the `unwrap(unit)`(a) operation simply evaluates a and throws the result away.

Closures representing functional values are represented by two unallocated words. One word points to the code part of the function. The other word points to a heap block containing the environment part: the values for the free variables of the function. Allocation of the environment part is not performed if it fits in one word. This simple approach already eliminates some closure allocations, especially in the case of simple curried functions. It seems impossible to avoid boxing the environment part in all cases: the type-based techniques proposed for tuples do not apply here, since we do not know the types of the values contained in the environment part. (The type of the function does not say anything about these types.)

Arrays are generic data structures that can be arbitrarily large. This makes coercion by copying impractical on arrays. Actually, ML arrays can be physically

Test	Gallium	Gallium0	SML-NJ	Caml	cc -02	What is tested
1 Takeushi	3.00	5.09	4.47	34.0	1.96	function calls (3 args), integer arithmetic
2 Integral	0.80	2.83	8.46	15.2	0.40	floating-point arithmetic, loops
3 List summation	3.60	3.45	5.12	7.90		list processing, integer arithmetic
4 Sieve	1.00	0.94	2.31	5.74		list processing, functionals, polymorphism
5 Boyer	1.80	2.76	3.60	14.6		term processing, function calls
6 Knuth-Bendix completion	0.90	0.98	1.11	12.4	0.86	term processing, functionals, polymorphism
7 Church integers	6.58	2.40	2.90	16.1		functionals, polymorphism
8 Solitaire	5.84	10.8	12.6	17.1	0.70	function calls, arrays, loops

Figure 3: Experimental results

modified, making copying semantically incorrect. Gallium uses a simple representation for arrays: it always keeps array elements in the wrapped state, as in the case of lists. Unwrapped array elements are desirable, however, since they lead to flat arrays, that are more compact and have better locality properties. One approach is to represent arrays as a flat block of unwrapped objects, paired with functions to read or write an element. The access functions coerce the array elements to or from the wrapped representation as needed. References to an array with a known type would directly access the array; references to an array with an unknown type would go through the access functions.

5.2 An overview of the implementation

The Gallium system compiles the Caml Light dialect of ML into assembly code for the MIPS R3000 processor [11]. It combines the data representation technique presented here with a conventional, non CPS-based back end, using some of the standard techniques from [1].

A compilation involves two passes that communicate through an intermediate language nicknamed "C−−". This is a simple expression-based language, that manipulates unboxed tuples of integers, floating-point numbers, or pointers (either code pointers or heap addresses). C−− provides most of the operations and control structures of C, minus the operations on `struct` and `union`. In addition, C−− directly supports exceptions and garbage collection.

This intermediate language is weakly typed: to each expression is attached a machine-level type expression. A machine-level type is simply a sequence of atomic types: either `int`, `float`, or `address`. Machine-level types contains just enough information for the back-end to determine the sizes and the calling conventions, and for the garbage collector to trace all pointers into the heap.

The front-end performs type inference, expands pattern-matching into decision trees, inserts the `wrap` and `unwrap` operations, and explicits closures. The front-end is entirely machine-independent. It embodies all the ML-specific treatments in the compiler. By contrast, the back-end is machine-dependent, but it knows almost nothing about ML. It performs instruction selection, reordering of computations, liveness analysis, register allocation by priority-based coloring of the interference graph [8], and emission of MIPS assembly code.

In the run-time system, the main originality is the use of static typing information to supplement the lack of tagging on objects that are not pointers. Traditionally, garbage collectors rely on run-time tags to distinguish pointers into the heap from other objects. Tags are also used to implement certain primitives such as generic equality. This solution is not adequate in our case, since we use native, unallocated 32-bit integers and floating-point numbers, that cannot be tagged. Instead, we make some of the static typing information available at run-time. Namely, each boxed object is adorned with a header giving the machine-level type of the object; each stack frame is associated with a descriptor giving the locations of live objects of type address in the corresponding function; a table contains the locations of all global variables of type address. This information allows the garbage collector to trace all valid pointers

into the heap.

It has been pointed out that normally this approach does not work well in the presence of polymorphism [2], since an object with static type α can be either an address into the heap, or an unboxed integer. In our case, such an object is guaranteed to be in the wrapped state; and we have arranged for all wrapped representations to be valid pointers, by boxing wrapped integers. Hence we can assume that all objects of type α are valid pointers. This fact allows the use of a simple, fast copying collector – at the cost of allocating wrapped integers. The alternative is to keep integers unboxed at all times, and revert to a collector with ambiguous pointers [5], which is slower and more complex.

5.3 Benchmarks

Figure 5.1 gives some experimental results obtained with the Gallium compiler. The tests were run on a DecStation 5000/200. All tests were limited to 8 megabytes. The times given are user CPU times, including garbage collection time. "Gallium" is the compiler described above; "Gallium0" is a version of the Gallium compiler that shares the same back-end and code generator, but uses conventional, boxed data representations; "SML-NJ" is Standard ML of New Jersey version 0.66, from Bell Labs and Princeton university; "Caml" is Caml version 3.1, from INRIA; and "cc" is the Ultrix 4.1 C compiler, at optimization level 2.

These figures indicates that the data representation technique described here lead to important speedups on some programs; have little impact on other programs; and really slow down one (fairly contrived) test program.

The best results are achieved on programs that perform mostly numerical computations (tests 1 and 2): unboxed, untagged integer and floating-point numbers really pay off in this case. For these programs, the execution times for Gallium are comparable to the times for the C compiler. The author believes that this data representation issue was the main bottleneck that prevented C-like code written in ML from being compiled as efficiently as in C.

Programs that perform mostly symbolic computation (tests 5, 6, and 8) also benefit from specialized data representations, although the speedups are less dramatic. This is somehow surprising, since these programs mostly manipulate values of concrete data types, that are always boxed. However, they benefit from having unallocated tuples to communicate with functions with several arguments, and unallocated closures to communicate with higher-order functions.

The most interesting tests are those that make heavy use of polymorphic data structures and polymorphic higher-order functions (tests 3, 4, 6, and 7). Polymorphic higher-order functions tend to execute less efficiently with mixed data representations: the stub code inserted around their functional arguments introduces extra function calls. In test 6, this potential slowdown is overcome by the other benefits of mixed representations (unallocated tuples and closures). Tests 3 and 4 shows a slight slowdown; apparently, it could be avoided by performing some compile-time reductions on a local scale. Test 7, however, demonstrates a major slowdown on a highly polymorphic program. The test consists in mapping `quad quad` (λ`x. x + 1`) on a list of integers, where `quad` is `double double`, and `double` is Church's numeral number two: λ`f. `λ`x. f(f x)`. In this example, the closure for `double` gets considered in rapid succession with different types (α, $\alpha \to \alpha$, and $(\alpha \to \alpha) \to (\alpha \to \alpha)$). The compiled code spends a lot of time switching between the various representations corresponding to these types. Aggressive compile-time reductions are required to eliminate these unnecessary coercions. The author has not encountered this phenomenon in more realistic examples than test 7, however.

6 Concluding remarks

The technique presented in this paper, while resulting in important speedups, is essentially local, and based solely on static typing information. This means it remains easy to prove correct, and easy to implement. No extra static analysis is required; such analyses are often quite expensive, to the point of being impractical. Higher-order functions cause no difficulties, while most other systems static analyses fail in this case. And separate compilation remains possible, since all we need are the types of external identifiers – an information provided by any module system.

Mixed data representations not only speed programs up, but also make it easier to interface with libraries written in another language, such as C: it suffices to take unwrapped representations compatible with the C data formats and calling conventions.

Standard ML features type abstraction (at the level of modules) in addition to polymorphism (at the level of terms). From the standpoint of data representation, type abstraction raises the same issues as polymorphism. For instance, there is this nasty restriction in Modula-2 [17], that an abstract type can only be implemented by a pointer type or an ordinal type, to ensure values of an abstract type fit in one word. Mixed data representations also work well with type abstraction: we take values of an abstract type to be unwrapped inside the structure that defines the abstract type, and wrapped outside, in the clients of the structure; the right coercions are introduced by applying the G transformation from section 3.2 to the values exported by the structure.

The technique presented in this paper works better in conjunction with compile-time reductions such as function inlining (though the Gallium compiler currently performs no inlining). Reductions can be performed before or after introducing the coercions. In the latter case, inlining a polymorphic function creates redexes of the form $\texttt{wrap}(\tau)(\texttt{unwrap}(\tau)(a'))$ or $\texttt{unwrap}(\tau)(\texttt{wrap}(\tau)(a'))$, that can trivially be replaced by a', saving one boxing step and one unboxing step. In the former case, a polymorphic function, once inlined, has a more specific type, and therefore can be compiled more efficiently.

When all polymorphic functions are systematically inlined, the program becomes completely monomorphic, and it can be compiled with optimal data representations. This essentially amounts to the Ada approach referred to in the introduction. The strength of our technique is that it is possible to stop compile-time reductions at any time (when the code becomes too large), and still get a correct program.

This paper has only considered simple coercions between the wrapped representations and the unwrapped representations. More elaborate coercion schemes can certainly be found. (Thatte [15] gives interesting examples of complex coercions.) In particular, all coercions considered here are strict; lazy coercions (coercions that would be performed only on demand) could lead to a better utilization of unwrapped data structures inside generic data structures such as lists and arrays. A more axiomatic presentation of the translation proposed here, giving minimal semantic conditions over the coercions, would certainly help in finding good sets of coercions.

Acknowledgments

Many thanks to Ian Jacobs, Damien Doligez and Luc Maranget for their comments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

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

[3] A. W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[4] A. W. Appel and D. B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture 1987*, volume 242 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[5] J. F. Bartlett. Compacting garbage collector with ambiguous roots. Technical report, DEC Western Research Laboratory, 1988.

[6] L. Cardelli. The functional abstract machine. *Polymorphism*, 1(1), 1983.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing surveys*, 17(4):471–522, 1985.

[8] F. Chow and J. Hennessy. Register allocation by priority-based coloring. *SIGPLAN Notices*, 19(6), 1984.

[9] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[10] L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[11] G. Kane. *MIPS RISC architecture*. Prentice-Hall, 1990.

[12] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. The MIT Press, 1990.

[13] 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.

[14] S. L. Peyton-Jones. Unboxed values as first-class citizens. In *Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, 1991.

[15] S. R. Thatte. Coercive type isomorphism. In *Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, 1991.

[16] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), 1990.

[17] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1983.