A Type-Based Compiler for Standard ML

Zhong Shao* Andrew W. Appel†
Yale University Princeton University
PRINCETON-CS-TR-487-95
March 28, 1995

Abstract

Compile-time type information should be valuable in efficient compilation of statically typed functional languages such as Standard ML. But how should type-directed compilation work in real compilers, and how much performance gain will type-based optimizations yield? In order to support more efficient data representations and gain more experience about type-directed compilation, we have implemented a new type-based middle end and back end for the Standard ML of New Jersey compiler. We describe the basic design of the new compiler, identify a number of practical issues, and then compare the performance of our new compiler with the old non-type-based compiler. Our measurement shows that a combination of several simple type-based optimizations reduces heap allocation by 36%; and improves the already-efficient code generated by the old non-type-based compiler by about 19% on a DECstation 5000.

1 Introduction

Compilers for languages with run-time type checking, such as Lisp and Smalltalk, must often use compilation strategies that are oblivious to the actual types of program variables, simply because no type information is available at compile time. For statically typed languages such as Standard ML (SML) [19], there is sufficient type information at compile time to guarantee that primitive operators will never be applied to values of the wrong type. But, because of SML's parametric polymorphism, there are still contexts in which the types of (polymorphic) variables are unknown. The program can still manipulate these values without inspecting their internal representation, as long as the size of every variable is known. The usual solution is to discard all the static type information and adopt the approach used for dynamically typed languages (e.g., Lisp), that is, to represent all program variables using *standard boxed representations*. This means that every variable, function closure, and function parameter, is represented in exactly one word. If the natural representation of a value (such as a floating-point number) does not fit into one word, the value is boxed (i.e., allocated on the heap) and the pointer to this boxed object is used instead. This is inefficient.

Leroy [15] has recently presented a *representation analysis* technique that does not always require variables be boxed in one word. In his scheme, data objects whose types are not polymorphic can be represented in multiple words or in machine registers; only those variables that have polymorphic types must use boxed representations. When polymorphic functions are applied to monomorphic values, the compiler automatically inserts appropriate coercions (if necessary) to convert polymorphic functions from one representation to another.

For example, in the following ML code:

```
fun quad (f, x) = (f(f(f(f(x)))))

fun h x =  x * x * 0.50 + x * 0.87 + 1.3

val res = h(3.14) + h(3.84) + quad(h, 1.05)
```

here *quad* is a polymorphic function with type

$$\forall\alpha.(((\alpha \to \alpha) * \alpha) \to \alpha);$$

all of the four calls to f inside *quad* must use the standard calling convention—passing x in a general-purpose register. On the other hand, h is a monomorphic function with type *real* $\to$ *real*, so every monomorphic application of h (such as $h(3.14)$ and $h(3.84)$) can use a more efficient calling convention—passing x in a floating-point register.

*Address: Department of Computer Science, Yale University, 51 Prospect Street, New Haven, CT 06520-8285. Email address: shao-zhong@cs.yale.edu.

†Address: Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544-2087. Email address: appel@cs.princeton.edu.

When h is passed to the polymorphic function *quad* (e.g., in $quad(h, 1.05)$), h must be *wrapped* to use the standard calling convention so that f will be called correctly inside *quad*. Suppose `fwrap` and `funwrap` are the primitive operations to box and unbox floating point numbers, then the compiler will wrap h into h':

$$h' = (\lambda y.\mathtt{fwrap}(h(\mathtt{funwrap}(y))));$$

the actual function application $quad(h, 1.05)$ is implemented as

$$\mathtt{funwrap}(quad(h',\ \mathtt{fwrap}(1.05))).$$

Representation analysis enables many interesting type-based compiler optimizations. But since no existing compiler has fully implemented representation analysis for the complete SML language, many practical implementation issues are still unclear. For example, while Leroy [15] has shown in detail how to insert coercions for core-ML, he does not address the issues in the ML module system [19, 17], that is, how to insert coercions for functor application and signature matching. Propagating type information into the middle end and back end of the compiler can also incur large compilation overhead if it is not done carefully, because all the intermediate optimizations must preserve type consistency. The major contributions of our work are:

- Our new compiler is the first type-based compiler for the entire Standard ML language.
- We extend Leroy's representation analysis to the SML module language to support module-level abstractions and functor applications.
- We improve compilation speed and code size by using partial types at module boundaries, by statically hash-consing lambda types, and by memo-izing coercions.
- We evaluate the utility of *minimum typing derivations* [7]—a method for eliminating unnecessary "wrapper" functions introduced by representation analysis.
- We show how the type annotations can be simplified in successive phases of the compiler, and how representation analysis can interact with the *Continuation-Passing Style* [24, 3] used by the SML/NJ compiler's optimizer [5, 3].
- We compare representation analysis with the crude (but effective) known-function parameter specialization implemented by Kranz [14].
- Our measurements show that a combination of several type-based optimizations reduces heap allocation by 36%, and improves the already-efficient code generated by the old non-type-based compiler by about 19%. We have previously reported a 14% speedup using new closure representations for accessing variables [23]; the two optimizations together yield a 28% speedup.

2 Data Representations

One important benefit of type-directed compilation is to allow data objects with specialized types to use more efficient data representations. In this section, we explain in detail what the *standard boxed representations* are, and what other more efficient alternatives one can use in type-based compilers.

Non-type-based compilers for polymorphic languages, such as the old SML/NJ compiler [5], must use the standard boxed representations for all data objects. Primitive types such as integers and reals are always tagged or boxed; every argument and result of a function, and every field of a closure or a record, must be either a tagged integer or a pointer to other objects that use the standard boxed representations. For example, in Figure 1a, the value x is a four-element record containing both real numbers and strings; each field of x must be boxed separately before being put into the top-level record. Similarly, y is a record containing only real numbers, but each field still has to be separately boxed under standard boxed representations.

We would like to use more efficient data representations, so that values such as x and y can be represented more efficiently, as shown in Figure 1b. But the intermixing of pointers and non-pointers (inside x) requires a complicated object-descriptor for the garbage collector, so we will reorder the fields to put all unboxed fields ahead of boxed fields (see Figure 1c); the descriptor for this kind of object is just two short integers: one indicating the length of the unboxed part, another indicating the length of the boxed part.

For recursive data types such as list z in Figure 2, the standard boxed representation will box each element of z, as shown in Figure 2a. More efficient data representations are also possible: if we know z has type $(real * real)$ *list*, z can be represented more compactly as shown in Figure 2b or Figure 2c. The major problem with these representations is that when z is passed to a polymorphic function such as *unzip*:

```
fun unzip l =
  let fun h((a,b)::r,u,w) = h(r,a::u,b::w)
        | h([],u,w) = (rev u, rev w)
   in h(l,[],[])
  end
```

the list z needs to be coerced from the more efficient representations (shown in Figure 2b and 2c) into the standard boxed representation (shown in Figure 1a). This coercion can be very expensive because its cost is proportional to the length of the list.[1] There are two solutions to solve this problem:

- One approach—proposed by Leroy [15]—is to use standard boxed representations for all recursive datatype objects. In other words, even though we know

[1] And this cost is not necessarily amortized, if the function takes time sublinear in the length of the list.

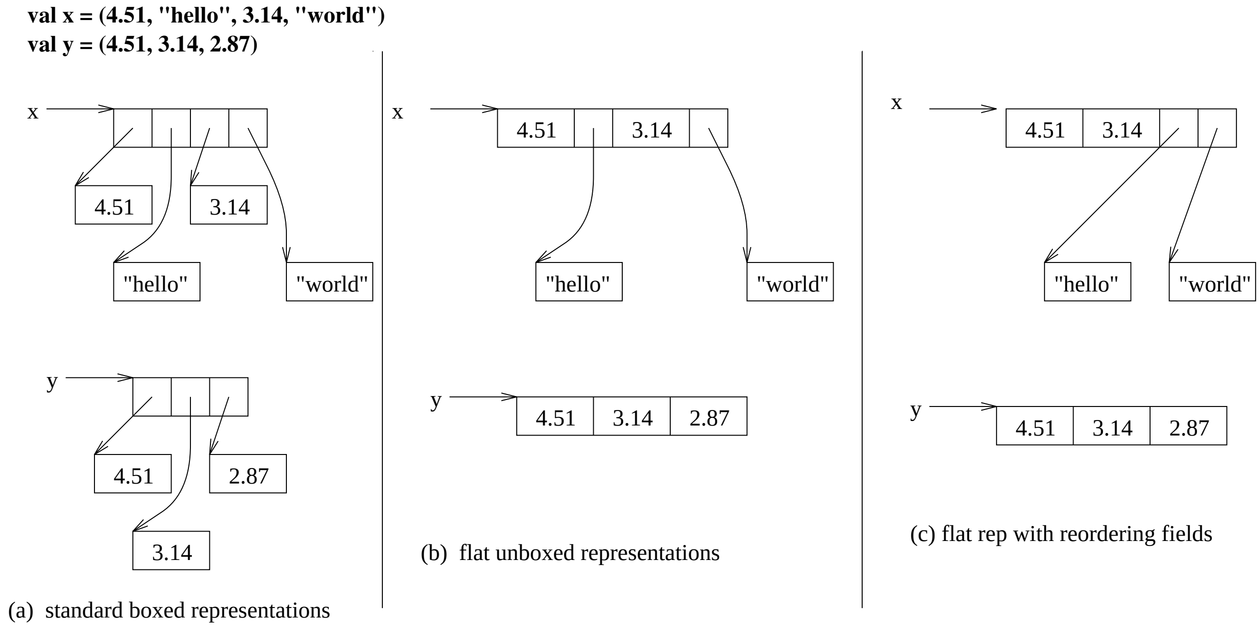

Figure 1: Standard boxed representations vs. Flat unboxed representations

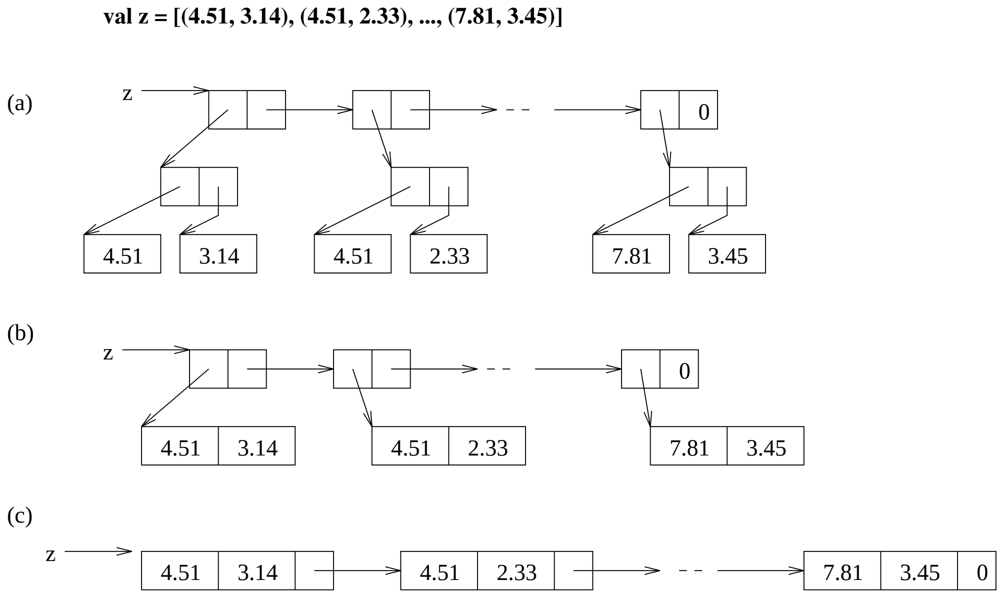

Figure 2: Recursive data type: (a) standard boxed representations (b-c) flat unboxed representations

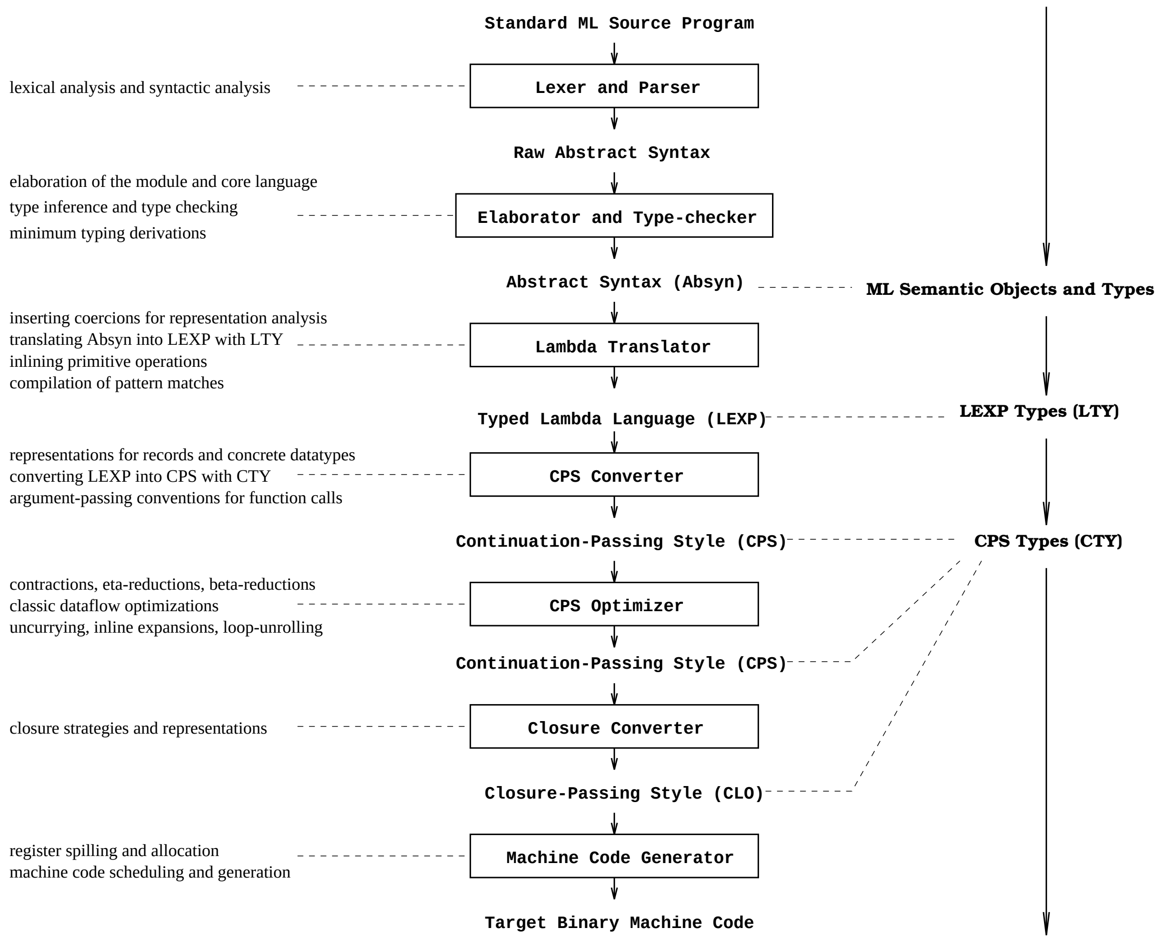

Figure 3: Overview of the new type-based SML/NJ compiler

z has type (*real* $*$ *real*) *list*, we still represent z as shown in Figure 2a, with each *car* cell pointing to an object that uses standard boxed representation. For example, if pairs such as `(4.51,3.14)` are normally represented as flat real vectors, when they are being added to (or fetched from) a list, they must be coerced from flat representations to (or from) standard boxed representations. The type-based compiler described in this paper also uses this approach. The LEXP language described later in Section 4.1 has a special lambda type called `RBOXEDty` to express this requirement.

- Another approach—proposed by Harper and Morrisett [12]—would allow recursive data types to use more efficient representations as shown in Figure 2b and 2c. In their scheme, types are passed as arguments to polymorphic routines in order to determine the representation of an object. For example, when z is passed to *unzip*, a type descriptor is also passed to indicate how to extract each *car* field. Because the descriptor has to be interpreted at runtime at each function call, it is not clear how efficient this approach would be in practice (see Section 7).

3 Front-End Issues

The Standard ML of New Jersey compiler is composed of several phases, as shown in Figure 3. The *Elaborator/Type-checker* produces typed abstract syntax (Absyn), which is almost unchanged from previous versions of the compiler [5], except that we annotate each occurrence of a polymorphic variable or data constructor with its type instantiation at each use, inferred by the type checker. The front end must also remember the details of each module-level (structure or functor) abstraction and instantiation in order to do module-level type-based analysis.

For example, in the following core-ML program,

```
fun square (x : real) = x * x

fun sumsquare (l : real list) =
     let fun h ([], s : real) = s
           | h (a::r, s) = h(r, a+s)
      in h(map square l, 0.0)
     end
```

the standard library function *map* (on lists) has polymorphic type

$$\forall\alpha\forall\beta.(\alpha \rightarrow \beta) \rightarrow (\alpha\ list \rightarrow \beta\ list);$$

(a) structure	(b) signature	(c) functor and functor application
`structure S =` `  struct type t = real` `         val p = 3.0` `         fun f x = (x, p)` `         val q = 4.0` `  end`	`signature SIG =` `  sig type t` `      val p : t` `      val f : t -> (t * t)` `  end`	`functor F(A : SIG) =` `  struct val r = A.f(A.p)` `  end` `structure W = F(S)`

Figure 4: Front end issues in the module language

	structure declaration	signature matching	abstraction declaration
ML code	`structure S = ...`	`structure U : SIG = S`	`abstraction V : SIG = S`
f's type	$S.f$: $\forall\alpha.\alpha \rightarrow (\alpha * \mathit{real})$	$U.f$: $\mathit{real} \rightarrow (\mathit{real} * \mathit{real})$	$V.f$: $t \rightarrow (t * t)$
p's type	$S.p$: *real*	$U.p$: *real*	$V.p$: t

Figure 5: Signature matching is *transparent* but abstraction matching is *opaque*

the *Elaborator/Type-checker* will annotate *map* with this polymorphic type, plus its instantiation

$$(\mathit{real} \rightarrow \mathit{real}) \rightarrow (\mathit{real\ list} \rightarrow \mathit{real\ list}).$$

Similarly, the polymorphic data constructor, "::", is also annotated with its original polymorphic type

$$\forall\alpha.(\alpha * \alpha\ \mathit{list}) \rightarrow \alpha\ \mathit{list},$$

plus its instantiation

$$(\mathit{real} * \mathit{real\ list}) \rightarrow \mathit{real\ list}.$$

To correctly support type-directed compilation for the entire SML language, all type abstractions and type instantiations in the module system must also be carefully recorded. In ML, basic modules (*structures* in Figure 4a) are encapsulated environments; module interfaces (*signatures* in Figure 4b) associate specifications with component names, and are used to both describe and constrain structures. Parameterized modules (*functors* in Figure 4c) are functions from structures to structures. A functor's argument is specified by a signature and the result is given by a structure expression, which may optionally be constrained by a result signature.

Abstraction and instantiation may occur in signature matching (Figure 5), abstraction declaration (Figure 5), functor application (see Figure 4c), and functor signature matching (used by higher-order modules [25, 17]). We use the example in Figure 4 and 5 to show what must be recorded in the Absyn during elaboration:

- Signature matching checks that a structure fulfills the constraints specified by a signature, and creates a new *instantiation structure* that is a restricted "view" of the original structure. The elaboration phase generates a *thinning* function that specifies all the visible components, their types (or thinning functions if they are substructures) in the original structure, and their new types in the instantiation structure. In Figure 5, U is bound to the result of matching structure S against signature *SIG*. Signature matching in ML is *transparent* [18, 16, 11], so f and p in the instantiation structure U have type $\mathit{real} \rightarrow (\mathit{real} * \mathit{real})$ and *real*. The new types and their old types in structure S (also shown in Figure 5) will be recorded in the thinning function.

- Abstraction is similar to signature matching; but matching for abstraction is *opaque* [18, 16, 11]. In addition to the thinning function, the elaboration phase also remembers the result signature. In Figure 5, V is an abstraction of structure S on signature *SIG*. V remembers the thinning function generated when doing signature matching of S against *SIG*, plus the actual signature *SIG*. During the elaboration of this abstraction, the type of f in S
$$\forall\alpha.\alpha \rightarrow (\alpha * \mathit{real})$$
is first instantiated into
$$\mathit{real} \rightarrow (\mathit{real} * \mathit{real})$$
in signature matching, and then abstracted into
$$t \rightarrow (t * t).$$

- Each functor application must remember its argument thinning function and its actual instantiation functor. In Figure 4c, functor F takes *SIG* as its argument signature, and returns a body structure that contains a value declaration r; the type of r is $A.t * A.t$. When F is applied to structure S, S is first matched against the argument signature *SIG* to get the actual argument instance, say S'; the elaborator then reconstructs the result structure W by applying F to S'. The component r in W has type $(\mathit{real} * \mathit{real})$. The front end records: (1) the thinning function generated when S is matched against *SIG*, (2) the actual instantiation of F, which has S' as its argument and W as its result.

- Functor signatures are essentially "types" of functors. Given a functor signature *FSIG* defined as

```
funsig FSIG (X: ASIG) = RSIG,
```

and a functor F, elaborating functor signature matching

```
functor G : FSIG = F
```

is equivalent to elaborate the functor declaration

```
functor G(X: ASIG) : RSIG = F(X).
```

Therefore, for each functor signature matching, the elaborator records everything occurring in functor application $F(X)$ plus the thinning function generated for matching $F(X)$ against the result signature *RSIG*.

Minimum typing derivation

Similar to the Damas-Milner type assignment algorithm W [10], the old *Elaborator/Type-checker* in our compiler infers the most general type schemes for all SML programs. As a result, local variables are always assigned the most general polymorphic types even though they are not used polymorphically. For example,

```
fun f(u,v) =
  let fun g(x, y, z) = ((x=y) andalso (y=z))
   in g(u*2.0, v*3.0, u+v)
  end
```

function f has type

$$real * real \to bool,$$

the let-bound function g is assigned a polymorphic type:

$$\forall\alpha.(\alpha * \alpha * \alpha) \to bool$$

where α is an equality type variable. But g is only used monomorphically with type

$$(real * real * real) \to bool.$$

To avoid coercion between polymorphic and mono morphic objects, we have implemented a "minimum typing derivation" phase in our new *Elaborator/Type-checker* to give all local variables "least" polymorphic types. The derivation is done after the elaboration so that it is only applied to type-correct programs. Our algorithm, which is similar to Bjørner's algorithm M [7], does a bottom-up traversal of the Absyn. During the traversal, we mark all non-exported variables: let-bound variables and those that are hidden by signature matching. Then, for each marked polymorphic variable v, we gather all of its actual type instantiations, say $\tau_1, ..., \tau_n$, and reassign v a new type—the *least general* type scheme that generalizes $\tau_1, ..., \tau_n$. The new type assigned to v is propagated into v's declaration d, constraining other variables referenced by d.

In the previous example, the let-bound function g is constrained by a new type assignment

$$(real * real * real) \to bool,$$

so the "=" operator can be implemented as the primitive equality function on real numbers, which is much more efficient than the polymorphic equality operator. Moreover, because g is no longer polymorphic, no coercion is necessary when applied to monomorphic values.

4 Translation into LEXP

The middle end of our compiler translates the Absyn into a simple typed lambda language called LEXP. During the translation, all the static semantic objects in Absyn, including types, signatures, structures, and functors, are translated into simple lambda types (LTY); coercions are inserted at each abstraction and instantiation site (marked by the front end) to correctly support representation analysis. In this section, we explain the details of our translation algorithm, and present solutions to several practical implementation problems.

4.1 The typed lambda language LEXP

The typed call-by-value lambda language LEXP is very similar to the untyped lambda languages[3, sec. 4.7] used in previous versions of the compiler: it contains lambda, application, constants, tuple and selection operators (i.e., RECORD and SELECT), and so on. But now it is a typed language[22], with types LTY described by this simple set of constructors:

```
datatype lty = INTty
             | REALty
             | RECORDty of lty list
             | ARROWty of lty * lty
             | BOXEDty
             | RBOXEDty
```

A lambda type LTY can be a primitive numeric type (INTty or REALty); a record type RECORDty$[t_1, t_2, \ldots]$ whose fields have type $t_1, t_2, ...$; a function type ARROWty(t, s) from t to s; or a boxed pointer type.

There are two kinds of boxed types. BOXEDty is a one-word pointer to an object (such as a record) whose fields may or may not be boxed. RBOXEDty is a one-word pointer to a recursively boxed object whose components are always recursively boxed; such objects use the standard boxed representations that are often used in a non-type-based ML compiler. Section 4.3 discusses how and where recursive boxing is necessary.

Our typed lambda language is a simply typed lambda calculus. Every function formal parameter is annotated by an LTY. Prim-ops and the exception-RAISE operator are also type-annotated. Types of other expressions (function application, record construction and selection) can be calculated bottom-up from the types of the subexpressions. To handle polymorphism, we introduce two new LEXP operators: WRAP(t, e) that boxes an evaluated expression e of type t into exactly one word; and UNWRAP(t, e) that unboxes an expression e into type t.

In ML, a let-bound variable can be polymorphic; but each use is treated as an instance of the type scheme. We treat this as a coercion, and we define a compilation function *coerce* that produces the right combination of WRAP and UNWRAP operators. Our *coerce* is similar to Leroy's *wrap* and *unwrap* [15]; but ours does not require that one type be an instantiation of the other. This generalization is useful in translating the ML module language into the LEXP language.

4.2 The Definition of Coerce

Coerce is a compile-time operation; given two LTYs t_1 and t_2, $\mathit{coerce}(t_1, t_2)$ returns a coercion function that coerces one *lexp* with type t_1 into another *lexp* with type t_2.

- If t_1 and t_2 are equivalent, no coercion is necessary, $\mathit{coerce}(t_1, t_2)$ returns the identity function.

- If one of t_1 and t_2 is `BOXEDty`, this requires coercing an arbitrary unboxed object into a pointer (or *vice versa*); the coercion here is a primitive `WRAP` or `UNWRAP` operation, written as
$$\mathit{coerce}(\mathtt{BOXEDty}, t_2) = \lambda e.\mathtt{UNWRAP}(t_2, e) \text{ and}$$
$$\mathit{coerce}(t_1, \mathtt{BOXEDty}) = \lambda e.\mathtt{WRAP}(t_1, e).$$

- If one of t_1 and t_2 is `RBOXEDty`, this requires coercing an arbitrary unboxed object into a pointer (or *vice versa*); moreover, the object itself must be coerced into the standard boxed representation (or *vice versa*); this coercion is similar to the *recursive wrapping* operations defined by Leroy [15]. It is defined as
$$\mathit{coerce}(\mathtt{RBOXEDty}, t_2) = \mathit{coerce}(\mathit{dup}(t_2), t_2) \text{ and}$$
$$\mathit{coerce}(t_1, \mathtt{RBOXEDty}) = \mathit{coerce}(t_1, \mathit{dup}(t_1)),$$
where the *dup* operation is defined as follows:
$$\mathit{dup}(\mathtt{RECORDty}[x_1, ..., x_n]) = \mathtt{RECORDty}[\mathtt{RBOXEDty}, ..., \mathtt{RBOXEDty}]$$
$$\mathit{dup}(\mathtt{ARROWty}(x_1, x_2)) = \mathtt{ARROWty}(\mathtt{RBOXEDty}, \mathtt{RBOXEDty})$$
$$\mathit{dup}(x) = \mathtt{BOXEDty}, \text{ for all other LTY } x$$

- If t_1 and t_2 are record type, i.e.,
$$t_1 = \mathtt{RECORDty}[a_1, ..., a_n] \text{ and}$$
$$t_2 = \mathtt{RECORDty}[r_1, ..., r_n],$$
we first build a list of coercions $[c_1, ..., c_n]$ for every record field where $c_i = \mathit{coerce}(a_i, r_i)$ for $i = 1, ..., n$. Assume v is a new lambda variable that corresponds to the original record, then the field of the new record is
$$f_i = c_i(\mathtt{SELECT}(i, v));$$
the coercion of expression e from t_1 to t_2 is
$$(\lambda v : \mathtt{BOXEDty}.\ \mathtt{RECORD}[f_1, ..., f_n])e.$$

- If t_1 and t_2 are function type, i.e.,
$$t_1 = \mathtt{ARROWty}(a_1, r_1) \text{ and}$$
$$t_2 = \mathtt{ARROWty}(a_2, r_2),$$
we first build the coercions c_a and c_r for the argument and the result, that is, $c_a = \mathit{coerce}(a_2, a_1)$ and $c_r = \mathit{coerce}(r_1, r_2)$; then assume u and v are two new lambda variables, the coercion of expression E from t_1 to t_2 is
$$\lambda u : a_2.(\lambda v : \mathtt{BOXEDty}.\ c_b(v))(E(c_a(u))).$$

4.3 Translating static semantic objects into LTY

The Absyn is translated into the lambda language LEXP through a simple top-down traversal of the Absyn tree. During the traversal, all static semantic objects and types used in Absyn are translated into LTYs. A signature or structure object s is translated into `RECORDty` where each field is the LTY translated from the corresponding component in s; a functor object is translated into `ARROWty` with the argument signature being the argument LTY, and the body structure being the result LTY. The translation of an ML type t into LTY is done using the algorithm described in Figure 6 (see function `ty2lty` for the pseudo code).

```
fun ty2lty(t) =
       (mark all type variables in t
        that ever appear in a constructor type;
        return lty(t))

fun lty(∀α.σ) = lty(σ)
  | lty({l1 : τ1,  ...,  ln : τn}) =
             RECORDty[lty(τ1), ..., lty(τn)]
  | lty(τ1 → τ2) = ARROWty(lty(τ1),lty(τ2))
  | lty(int) = INTty
  | lty(bool) = INTty
  | lty(unit) = INTty
  | lty(real) = REALty
  | lty(α) = if α is a marked type variable
                  then RBOXEDty else BOXEDty
  | lty(t) = if the constructor type t is rigid
             then BOXEDty else RBOXEDty
```

Figure 6: Translating ML type into LTY (pseudo code)

More specifically, given a type t, the translation algorithm `ty2lty` divides the type variables in t into two categories:

- Those that ever appear in constructor types, [2] such as α in type $(\alpha{*}\alpha\ \mathit{list}) \rightarrow \alpha\ \mathit{list}$, and β in type $(\beta\ \mathit{ref} * \beta) \rightarrow \mathit{unit}$; they are translated into `RBOXEDty` (the need for this is explained in Section 2);

- All other type variables, such as γ in $\gamma * \gamma \rightarrow \gamma$; they are translated into `BOXEDty`;

[2] Record type constructors and function type constructors ("→") are not counted here.

A polymorphic type $\forall\alpha_1 ... \forall\alpha_n.\tau$ is translated by ignoring all quantifications. The arrow type constructor "$\rightarrow$" for functions is translated into `ARROWty`; the record type constructor is translated into `RECORDty`, with its fields ordered properly.

All *rigid* constructor types,[3] such as *string*, α *list*, and (*real* $*$ *real*) *ref*, are translated into `BOXEDty`. All *flexible* constructor types are translated into `RBOXEDty`.

Type abstraction

One main challenge in doing module-level representation analysis is to deal with *flexible* constructor type (also called type abstraction). For example, in the following ML program,

```
signature SIG =
  sig type 'a t
      val p : real t
      val f : 'a t -> 'a t
  end

functor F(S : SIG) =
  struct datatype 'a foo = FOO of 'a S.t
         val r = S.f(S.p)
         fun g (FOO x) = [x]
  end
```

`t` is a flexible type constructor (with arity 1) that will not be instantiated until functor F is applied. Simple representation analysis [15] would run into two problems when compiling F's body:

- At the function application *S.f(S.p)*, since *S.f* is polymorphic and *S.p* is monomorphic, a coercion must be inserted here; but the detail of this coercion is not known because it depends on the actual instantiation of t. For example, if `'a t` is instantiated into `'a * 'a` when F is applied, `S.f` has to be "unwrapped" from type `('a * 'a) -> ('a * 'a)` into type `(real * real) -> (real * real)`.
- Function g puts the variable x of type `'a t` into a list. This requires *recursive wrapping* (see Section 2). But because `t` is unknown, this recursive coercion is also unknown.

We solve these two problems by forcing all objects with flexible type to use the standard boxed representations (i.e., as `RBOXEDty`) and by properly coercing all structure components (including values, functions, and data constructor injections and projections) from the abstract types into their actual instantiations (during functor application).

In the previous example, the body of functor F references several identifiers defined in the argument signature *SIG*. Because *S.t* is flexible (i.e., abstract) inside F, the identifier *S.p* has LTY

$$lt_p = \texttt{RBOXEDty}$$

and the identifier *S.f* has LTY

$$lt_f = \texttt{ARROWty(RBOXEDty,BOXEDty)}.$$

Because *S.f* and *S.p* are already recursively boxed, no coercion is necessary when *S.f* is applied to *S.p*.

Similarly, the projection of data constructor *FOO* used in the body of F has type `'a foo -> 'a S.t`; its corresponding LTY is

$$lt_c = \texttt{ARROWty(BOXEDty,RBOXEDty)},$$

that is, the value carried by *FOO* (i.e., the argument x of function g) is already recursively boxed, therefore, no recursive coercion is needed when putting x into a list.

When F is applied to the following structure *A*,

```
structure A = struct type 'a t = 'a * 'a
                     val p = (3.0,3.0)
                     fun f x = x
              end

structure T = F(A)
```

first, the argument structure A is matched against the signature *SIG* via *abstraction matching*, producing a structure A' that precisely matches *SIG*; the components p and f are coerced to LTY lt_p and lt_f in A'. Then, F is applied to this "abstract" structure A', produce another "abstract" structure—the functor body T'. Finally, T' is coerced back to the more "concrete" structure T; for example, T'*.r* which has LTY `RBOXEDty` is coerced into a record (*T.r*) that has LTY

$$\texttt{RECORDty[REALty,REALty]},$$

the projection of data constructor T'*.FOO* which has LTY lt_c is coerced into a projection (for *T.FOO*) that has LTY

$$\texttt{ARROWty(BOXEDty,RECORDty[BOXEDty,BOXEDty])}.$$

Here, coercions of projections and injections for data constructors can be implemented by recording the origin type lt_c with *T.FOO* or by using abstract value constructors proposed by Aitken and Reppy [1].

4.4 Translating Absyn into LEXP

Now that we have explained how to translate static semantic objects into LTY and how to coerce from one LTY to another, the translation of Absyn into LEXP is straightforward. Coercions (built from `WRAP` and `UNWRAP`) are inserted at each use of a polymorphic variable, and at module-level signature matching.

polymorphic variables: Given a polymorphic variable v in Absyn, the front end has annotated every use of v with its polymorphic type σ plus its actual instantiation τ. Assume that σ and τ are translated into LTYs s and t, variable v is then translated into the LEXP expression *coerce*$(s,t)(v)$.

[3] Following the SML Definition and Commentary [19, 18], all type constructor names defined as type specifications in signatures are *flexible*; all other type constructor names are *rigid*.

polymorphic data constructors: Polymorphic data constructors are treated like polymorphic variables; coercions are applied to data constructor injections and projections.

primitive operators: Polymorphic prim-ops whose implementations are known at compile time can be specialized based on their actual type instantiations. For example, polymorphic equality, if used monomorphically, can be translated into primitive equality; integer assignments and updates can use unboxed update.[4]

signature matching: Suppose structure S is matched against signature *SIG*, and U is the result instantiation structure; then the thinning function generated by the front end is translated into a coercion c, which fetches every component from S, and coerces it to the type specified in U. If S is denoted by v, then the translation of this signature matching is simply $c(v)$.

abstraction: Abstraction is translated in the same way as signature matching, except that the result $c(v)$ must also be coerced into the LTY for the signature *SIG*. Assume that the LTYs for U and *SIG* are respectively u and s, then the abstraction of structure S under *SIG* is $(\mathit{coerce}(u, s))(c(v))$.

functor application: Suppose the argument signature of functor F is *SIG* and F is applied to structure S. The front end has recorded the thinning function for matching S against *SIG* and the actual functor instance F' for F. As before, assume the result of matching S against *SIG* is $c(v)$, and F is denoted by the LEXP expression f, and the LTYs for F and F' are respectively s and t, then the LEXP expression for F' is $f' = (\mathit{coerce}(s, t))(f)$, and functor application $F(S)$ is translated into $\texttt{APP}(f', c(v))$.

4.5 Other practical issues

Because of large LTYs and excessive coercion code, a naive implementation of the translation algorithm can lead to large LEXP expressions and extremely slow compilation. This problem is severe for programs that contain many functor applications and large structure and signature expressions.

Since $\mathit{coerce}(s, t)$ is an identity function in the common case that $s = t$, we can improve the performance of the compiler by optimizing the implementation of coerce with an extra test:

$$\begin{array}{l} \mathit{coerce}(s,t) = \\ \quad \textbf{if } s = t \\ \quad \textbf{then } \lambda f.f \\ \quad \textbf{else } \textit{structural induction} \end{array}$$

But how can $s = t$ be tested efficiently? We use *global static hash-consing* to optimize the representation for lambda types; hash-consed structures can be tested for equality in constant time, and hash-consing reduces space usage for shared data structures.

This optimization was crucial for the efficient compilation of functor applications: without hash-consing, a one-line functor application (whose parameter is a reference to a complicated, separately defined signature) could take tens of minutes and tens of extra megabytes to compile; with hash-consing, functor application is practically immediate. Also, general space usage of the compiler is less with hash-consing, as the static representations of different functors can share structure.

Management of a global hash-cons table is not completely trivial; we would like to delete stale data from the table, but how do we know what is stale? We use *weak pointers* (pointers that the garbage collector ignores when tracing live data and that are invalidated when the object pointed to is collected), a special feature supported by SML/NJ; this is effective, though it seems a bit clumsy.

Naive translation of static semantic objects may also drag in large LTYs that are mostly useless. For example, to compile

```
(Compiler.Control.CG.calleesaves := 3;
 Compiler.AllocProf.reset())
```

we need to know only that variable `calleesaves` has type *int*, and variable `reset` has type *unit* → *unit*. However, our translation algorithm will have to include the type of structure `Compiler` which may contain hundreds of components. So we extend our lambda type notation with the following new constructs:

```
datatype lty  = ......
              | GRECty of (int * lty) list
              | SRECORDty of lty list
```

Here, `SRECORDty` is same as `RECORDty` except that it is used particularly for module constructs (i.e., structures). The LTY `GRECty` is used to type external structures such as the `Compiler` structure above; a `GRECty` specifies a subset of record fields (and their corresponding LTYs) that are interesting to the current compilation unit. The LTYs for all external structure identifiers are inferred during the *Lambda Translation* phase rather than being translated from their corresponding static semantic objects. For example, the LTY for structure `Compiler` in the above example will be lt_{comp}:

$$lt_{comp} = \texttt{GRECty}[(3, lt_{ctrl}), (7, lt_{alloc})] \text{ where}$$

[4] In order to support generational garbage collection [26], most compilers must do some bookkeeping at each update so that the pointers from older generation to youngest generation are correctly identified. *Unboxed update* is a special operator that assigns a non-pointer value into a reference cell; such updates cannot cause older generations to point to newer ones, so no bookkeeping is necessary.

lt_{ctrl} = GRECty[(0, GRECty[(43,INTty)])] and

lt_{alloc} = GRECty[(0, ARROWty(INTty,INTty))].

Here, we assume that `Control` and `AllocProf` are the 3rd and 7th fields of `Compiler`, `CG` is the 0th field of `Control`, `calleesaves` is the 43nd field of `CG`, and `reset` is the 0th field of `AllocProf`.

We also save code size and compilation time by sharing coercion code between equivalent pairs of LTYs, using a table to memo-ize the *coerce*(s, t) function. Coercions introduced in the *coerce* procedure are normally inlined in the CPS optimization phase, because they are applied just once. Shared coercions are often not inlined, because they can cause excessive code explosion. Because shared coercions can be more expensive than normal inlined coercions, we only use this hashing approach for coercions between module objects. This compromise works quite well in practice, because it is often the large module objects that are causing the "excessive coercion code" problem. Since module-level coercions are not executed often, the generated code is not noticeably slower.

5 Typed CPS Back End

Standard ML of New Jersey uses *continuation-passing style* (CPS) as its intermediate representation for program optimization. The LEXP language is converted into a CPS notation (*cexp*) that makes flow of control explicit. CPS [24, 14] and its use in the SML/NJ compiler [4, 3] have been described in the literature.

Previous versions of SML/NJ [5] have used an untyped CPS language. But now we propagate some very simple type annotations into CPS. Each variable in the CPS language is annotated, at its binding occurrence, with a "CPS type" (CTY). The CTYs are very simple:

```
datatype cty = INTt | PTRt of int option
             | FUNt | FLTt | CNTt
```

so they are very easy and cheap for the back end to maintain.

A CPS variable can be a tagged integer (`INTt`), a pointer to a record of length k (`PTRt(SOME k)`), a pointer to a record of unknown length (`PTRt(NONE)`), a floating-pointer number (`FLTt`), a function (`FUNt`), or a continuation (`CNTt`). The translation frm LTY to CTY is straight-forward:

```
fun lty2cty lt =
  case lt of INTty => INTt
           | REALty => FLTt
           | BOXEDty => PTRt(NONE)
           | RBOXEDty => PTRt(NONE)
           | ARROWty _ => FUNt
           | RECORDty l => PTRt(SOME(length l))
```

[5] up to version 0.93, released in 1993

Because the CPS conversion phase has made implementation decisions for records and functions, the CTY is no longer concerned with the details of `RECORDty` and `ARROWty`.

We augment the set of CPS prim-ops with specific wrapper/unwrapper operations for integers (i.e., `iwrap` and `iunwrap`), floating-point numbers (i.e., `fwrap` and `funwrap`), and pointers (i.e., `wrap` and `unwrap`). For example, `fwrap` "boxes" a floating-point number, producing a pointer, and `funwrap` is the inverse operation. For m-bit integers represented in an n-bit word with an $(n - m)$-bit tag, `iwrap` could apply the tag (by shifting and OR-ing), and `iunwrap` could remove it. For integers represented by boxing, then `iwrap` could heap-allocate a boxed integer.

5.1 CPS conversion

The overall structure and algorithm of our CPS conversion phase is almost same as the one described by Appel [3, Ch. 5]. The conversion function takes two arguments: an LEXP expression E and a "continuation" function c of type *value* $\rightarrow$ *cexp*; and returns a CPS expression as the result. But now, during the conversion process we gather the LTY information for each LEXP expression, and maintain an LTY environment for all CPS variables. The LTYs are used to make implementation decisions for records and function calls, and are also translated into CTYs to annotate CPS variables.

The CPS-conversion phase decides how to represent each record, and encodes its decisions in the CPS operation sequences it emits. Converting LEXP records is the most interesting case. Given an LEXP expression RECORD[u_1, u_2, ..., u_n], suppose the LTY for each u_i is t_i ($i = 1, ..., n$), we can represent the record using any of the layouts shown in Figures 1: with every field boxed and every integer tagged (Figure 1a), using flat records of reals (y in Figure 1b), with mixed boxed and unboxed fields (x in Figure 1b), or with segregated boxed/unboxed fields (Figure 1c). The translation of `SELECT` expressions must correspond to the layout convention used for records.

CPS conversion also decides the argument-passing convention for all function calls and returns. In ML, each function has exactly one argument, but this argument is often an n-tuple. In most cases, we would like to pass the n components in registers. The previous SML/NJ compilers could "spread" the arguments into registers only when caller and callee were in the same compilation unit, and the call sites were not obscured by passing functions as arguments. The new, type-based compiler can use register arguments based on type information. If the type of f's argument is RECORDty[$t_1, \ldots, t_n$], and n is not so large that the register bank will be exhausted,[6] we pass all components in registers (unlike LEXP, CPS does have multi-argument functions). Similarly, a function that returns an n-tuple value

[6] We currently use a threshold of $n \leq 10$ on 32-register RISC machines.

will be translated using a n-argument continuation function, for suitably small n.

Finally, the primitive coercion operations, `WRAP`(t,e) and `UNWRAP`(t,e), are converted into corresponding CPS primitive operations. Based on whether t is `INTty`, `REALty`, or other pointer types, `WRAP` and `UNWRAP` are translated into `iwrap` and `iunwrap`, `fwrap` and `funwrap`, or `wrap` and `unwrap`.

5.2 Optimization and closure conversion

When the CPS conversion phase is finished, the compiler has made most of the implementation decisions for all program features and objects: structures and functors are compiled into records and functions; polymorphic functions are coerced properly if they are being used less polymorphically; pattern matches are compiled into switch statements; concrete data types are compiled into tagged data records or constants; records are laid out appropriately based on their types; and the function calling conventions are mostly decided.

The optimizer of the SML/NJ compiler operates on the CPS intermediate representation. Optimization phases are almost unchanged, except that they must correctly propagate the CTYs, which is simple to do; CPS optimizations are naturally type-consistent. Besides those described by Appel [3], two new CPS optimizations are performed: pairs of "wrapper" and "unwrapper" operations are cancelled; and record copying operations of the form

```
let val (x,y) = a in (x,y) end
```

can be eliminated, since now we know the size of a at compile time.

To represent environments for higher-order functions with nested scope, the compiler uses our new space-efficient closure conversion algorithm [23]. Previously, this phase had to discover which functions are continuations by dataflow analysis [6]; now the information is manifest in the CTYs.

When the closure analysis phase must build heap records for closure environments, it can use all the record representations shown in Figure 1.

The closure conversion algorithm is very cautious about "optimizing" transformations that extend the lifetime of variables, since this can cause a kind of memory leak [23, 3]. The CTY information allows the lifetime of integers (and other constant-size variables) to be safely extended, a useful benefit.

6 Performance Evaluation

Type-directed compilation should support much more efficient data representations. In order to find out how much performance gain we can get for different type-based optimizations, we have measured the performance of six different compilers on twelve SML benchmarks (described in Shao [22]). Among these twelve benchmarks, **MBrot**, **Nucleic**, **Simple**, **Ray**, and **BHut** involve intensive floating-point operations; **Sieve** and **KB-Comp** frequently use first-class continuations or exceptions; **VLIW** and **KB-Comp** make heavy use of higher-order functions. The average size of these benchmarks is 1820 lines of SML source code.

The six compilers we use are all simple variations of the Standard ML of New Jersey compiler version 1.03z. All of these compilers use the new closure conversion algorithm [23] and with three general purpose callee-save registers [6], and all use tagged 31-bit integer representations. Other aspects of these compilers are close to those described by Appel [3].

sml.nrp A non-type-based compiler. No type information is propagated beyond the elaboration phase. Data uses standard boxed representations. Functions take one argument and return one result.

sml.fag The **sml.nrp** compiler with the *argument flattening* optimization [14, 3]. If the call sites of a function are known at compile time, its n-tuple argument can be flattened and passed in n registers. This compiler is similar to SML/NJ 0.93 [3].

sml.rep The new type-based compiler that supports very basic representation analysis (on records). Floating point numbers still use boxed representations. Hash-consing of LTY's (Section 4.5) is not used (had not yet been implemented) in this version.

sml.mtd The **sml.rep** compiler plus the implementation of *minimum typing derivations*.

sml.ffb The **sml.mtd** compiler plus support of unboxed floating point numbers. Function call and return pass floating-point arguments in floating-point registers. Records of floats are represented "flat," as in Figure 1b.[7] Records that contain both boxed and unboxed values are still represented as two layers, with each unboxed value being boxed separately.

sml.fp3 The **sml.ffb** compiler, but with three floating-point callee-save registers [22].

All measurements are done on a DEC5000/240 workstation with 128 megabytes of memory, using the methodology described in Shao's Ph.D. thesis[22]. In Figure 7 we show the execution time of all the benchmarks using the above six compilers, using **sml.nrp** as the baseline value. Figure 8 compares execution time, heap allocation, code size, and compile time (based on the average ratios of all twelve benchmarks). We can draw the following conclusions from these comparisons:

[7] Unfortunately, the SML/NJ version 1.03z still uses the old runtime system [2]. Reals are not always aligned properly, so memory fetch (or store) of a floating-point number must be implemented using two single-word memory-load (memory-store) instructions.

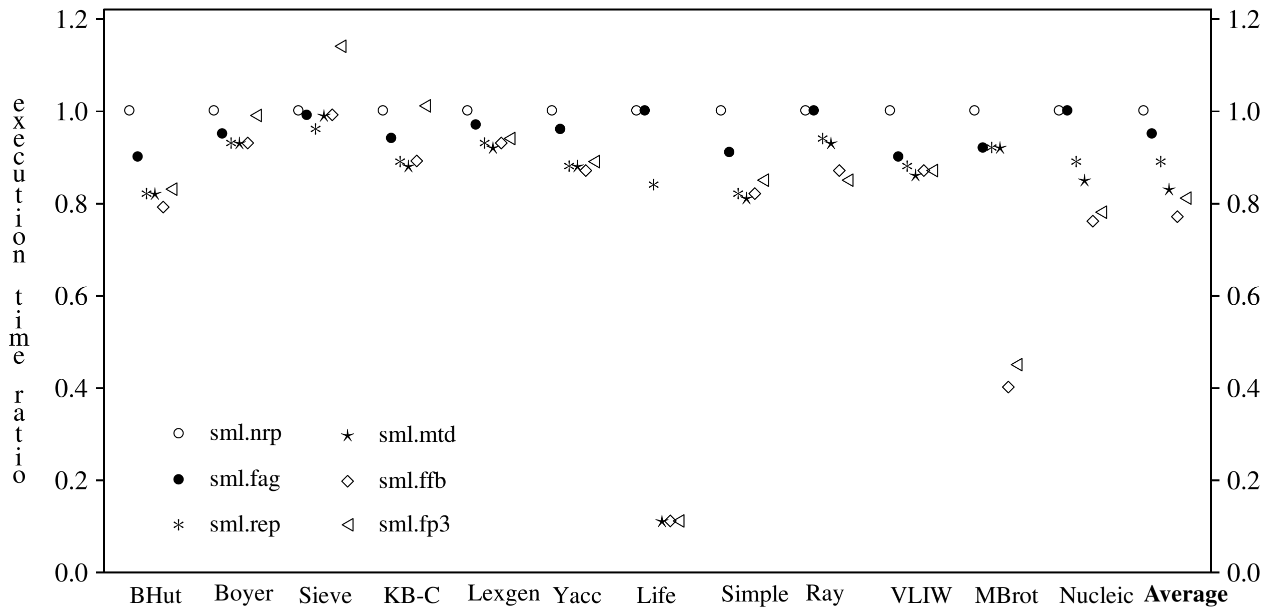

Figure 7: A comparison of execution time

Program	**sml.nrp** (base)	**sml.fag** (ratio)	**sml.rep** (ratio)	**sml.mtd** (ratio)	**sml.ffb** (ratio)	**sml.fp3** (ratio)
Execution time	1.00	0.95	0.89	0.83	0.77	0.81
Heap allocation	1.00	0.90	0.70	0.66	0.58	0.63
Code size	1.00	0.98	0.97	0.97	0.99	1.01
Compilation time	1.00	1.04	1.06	1.09	1.10	1.17

Figure 8: Summary comparisons of resource usage

- The type-based compilers perform uniformly better than older compilers that do not support representation analysis. The **sml.ffb** compiler gets nearly 19% speedup in execution time and decreases the total heap allocation by 36% (on average) over the older SML/NJ compiler (i.e., **sml.fag**) that uses uniform standard boxed representations. This comes with an average of 6% increase in the compilation time. The generated code size remains about the same.

- The simple, non-type-based argument flattening optimization in the **sml.fag** compiler gives a useful 5% speedup.

- The **sml.rep** compiler, which supports passing argument in registers (but not floating-point registers), only improves the performance of the non-typed-based **sml.fag** compiler by about 6%. It does decrease heap allocation by an impressive 20%.

- *Minimum typing derivations* were intended to eliminate coercions; but most of the coercions eliminated by MTD would have been eliminated anyway by CPS contractions. The only significant speedup of the **sml.mtd** compiler over **sml.rep** is from the **Life** benchmark where with MTD, the (slow) polymorphic equality in a tight loop (testing membership of an element in a set) is successfully transformed into a (fast) monomorphic equality operator—and the program runs 10 times faster.

7 Related Work and Conclusion

Representation analysis, proposed and implemented by Leroy [15] (for ML-like languages) and Peyton Jones and Launchbury [20] (for Haskell-like languages), allows data objects whose types are not polymorphic to use more efficient unboxed representations. Peyton Jones and Launchbury's approach [20] requires extending the language (i.e., Haskell) with a new set of "unboxed" monomorphic types; the programmer has to explicitly write "boxing" coercions when passing unboxed monomorphic values to polymorphic functions. Leroy's [15] approach is more attractive because it requires no language extension or user intervention. The work described in this paper is an extension and implementation of Leroy's techniques for the entire SML language. We concentrate on practical issues of implementing type-directed compilation such as interaction with ML module system and efficient propagation of type information through many rounds of compiler transformations and optimizations.

Many people have worked on eliminating unnecessary "wrapper" functions introduced by representation analysis. Both Peyton Jones [20] and Poulsen [21] let the programmer to tag some types with a *boxity* annotation, and then statically determine when to use boxed representations. Henglein and Jorgensen [13] present a term-rewriting method that translates a program with many coercions into one that contains a "formally optimal" set of coercions. Neither technique appears easy to extend to the SML module language. We use *minimum typing derivations* [7] to decrease the degree of polymorphism for all local and hidden functions. This is very easy to extend to the module system. Our approach may achieve almost the same result as "formally optimal" unboxing. And we have shown that simple dataflow optimizations (cancelling wrap/unwrap pairs in the CPS back end) is almost as effective as type-theory-based wrapper elimination.

Type specialization or customization [9, 8] is another way to transform polymorphic functions into monomorphic ones. Specialization can also be applied to parameterized modules (i.e., functors), just as *generic* modules are implemented in Ada and Modula-3. Because of the potential code explosion problem, the compiler must do static analysis to decide when and where to do specialization. Our type-based compiler uses coercions rather than specializations; however, because our CPS optimizer [3] always inline-expands small functions, small and local polymorphic functions still end up being specialized in our compiler. We believe that a combination of representation analysis and type specialization would achieve the best performance, and we intend to explore this in the future.

Harper and Morrisett [12] have recently proposed a type-based compilation framework called *compiling with intensional type analysis* for the core-ML language. They use a typed lambda calculus with explicit type abstractions and type applications as the intermediate language. Their scheme avoids recursive coercions by passing explicit type descriptors whenever a monomorphic value is passed to a polymorphic function. Since they have not implemented their scheme yet, it is unclear how well it would behave in practice. Because their proposal only addresses the core-ML language, we still do not know how easily their scheme can be extended to the SML module language.

We believe that type-based compilation techniques will be widely used in compiling statically typed languages such as ML in the future. The beauty of type-based representation analysis is that it places no burdens on the user: the source language does not change, programmers do not need to write coercions, and separate compilation works cleanly because interfaces are speficied using types.

By implementing a fully working type-based compiler for the entire SML language, we have gained experience with type-directed compilation, and solved many practical problems involved in the implementations. Our performance evaluation shows that type-based compilation techniques can achieve significant speedups on a range of benchmarks.

Acknowledgments

We would like to thank John Reppy, Sheng Liang, and Patrick Sansom for comments on an early version of this paper. This research is supported in part by National Science Foundation Grants CCR-9002786, CCR-9200790, and CCR-9501624.

References

[1] William E. Aitken and John H. Reppy. Abstract value constructors: Symbolic constants for Standard ML. Technical Report TR 92-1290, Department of Computer Science, Cornell University, June 1992. A shorter version appears in the proceedings of the "ACM SIGPLAN Workshop on ML and its Applications," 1992.

[2] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3(4):343–80, 1990.

[3] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symp. on Principles of Programming Languages*, pages 293–302, New York, 1989. ACM Press.

[5] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[6] Andrew W. Appel and Zhong Shao. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation*, 5:189–219, September 1992.

[7] Nikolaj S. Bjorner. Minimial typing derivations. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 120–126, June 1994.

[8] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Stanford,California, March 1992. Tech Report STAN-CS-92-1420.

[9] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proc. ACM SIGPLAN '89 Conf. on Prog. Lang. Design and Implementation*, pages 146–160, New York, July 1989. ACM Press.

[10] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–12, New York, 1982. ACM Press.

[11] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty First Annual ACM Symp. on Principles of Prog. Languages*, pages 123–137, New York, Jan 1994. ACM Press.

[12] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[13] Fritz Henglein and Jesper Jorgensen. Formally optimal boxing. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 213–226. ACM Press, 1994.

[14] David Kranz. *ORBIT: An optimizing compiler for Scheme*. PhD thesis, Yale University, New Haven, CT, 1987.

[15] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, January 1992. ACM Press.

[16] Xavier Leroy. Manifest types, modules, and separate compilation. In *Twenty First Annual ACM Symp. on Principles of Prog. Languages*, pages 109–122, New York, Jan 1994. ACM Press.

[17] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *Proc. European Symposium on Programming (ESOP'94)*, pages 409–423, April 1994.

[18] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, Cambridge, Massachusetts, 1991.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[20] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *The Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, New York, August 1991. ACM Press.

[21] Eigil Poulsen. Representation analysis for efficient implementation of polymorphism. Master's thesis, DIKU, University of Copenhagen, 1993.

[22] Zhong Shao. *Compiling Standard ML for Efficient Execution on Modern Machines*. PhD thesis, Princeton University, Princeton, NJ, November 1994. Tech Report CS-TR-475-94.

[23] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 150–161. ACM Press, 1994.

[24] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[25] Mads Tofte. Principal signatures for higher-order program modules. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 189–199, New York, Jan 1992. ACM Press.

[26] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, Cambridge, MA, 1986.