An Overview of the FLINT/ML Compiler*

Zhong Shao

Dept. of Computer Science
Yale University
New Haven, CT 06520-8285
shao-zhong@cs.yale.edu

Abstract

The FLINT project at Yale aims to build a state-of-the-art systems environment for modern type-safe languages. One important component of the FLINT system is a high-performance type-directed compiler for SML'97 (extended with higher-order modules). The FLINT/ML compiler provides several new capabilities that are not available in other type-based compilers:

- type-directed compilation is carried over across the higher-order module boundaries;
- recursive and mutable data objects can use unboxed representations without incurring expensive runtime cost on heavily polymorphic code;
- parameterized modules (functors) can be selectively specialized, just as normal polymorphic functions;
- new type representations are used to reduce the cost of type manipulation thus the compilation time.

This paper gives an overview of these novel aspects, and a preliminary report on the current status of the implementation.

1 Introduction

The FLINT project at Yale aims to build a state-of-the-art systems environment for modern type-safe languages. One important component of the FLINT system is a high-performance type-directed compiler for Standard ML 1997 (SML'97) [21] extended with higher-order modules [20]. The FLINT/ML compiler provides several new capabilities that are not available in other existing type-based compilers (i.e., Gallium [16], SML/NJ [29], and TIL [30]):

- First, type-directed compilation is carried over to the ML module system including even extensions such as higher-order functors [20]. Neither Gallium [16] nor TIL [30] provides full support of ML modules. The type-based SML/NJ [29] does support the entire SML module language, but values of abstract types must use *recursively boxed* data representations.

*This research was sponsored in part by the Defense Advanced Research Projects Agency ITO under the title "Building Evolutionary Software through Modular Executable Specifications and Incremental Derivations," DARPA Order No. D961, issued under Contract No. F30602-96-2-0232, and in part by an NSF CAREER Award CCR-9501624, and NSF Grant CCR-9633390. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

- Second, recursive and mutable data objects can use unboxed representations without incurring expensive runtime cost on heavily polymorphic code [26]. The *coercion-based* approach used in Gallium [16] and SML/NJ [29] does not support unboxed representations on recursive and mutable objects. The *type-passing* approach used in TIL [30] does handle all data objects, but it involves heavy-weight runtime type analysis and code manipulations.

- Third, ML functors can be selectively *specialized* just as normal polymorphic functions. FLINT/ML can compile both functors and polymorphic functions into a predicative variant of the Girard-Reynolds polymorphic calculus, F_ω [9, 25], so functor specialization is just type application in F_ω. This is not supported in any of the other three compilers.

- Fourth, FLINT/ML uses several techniques such as *hash consing*, *memoization*, and Nadathur's suspension-based λ-calculus [23, 24] to optimize the representation of its typed intermediate format. The new representation can reduce the cost of type manipulations thus improving the compilation time.

The rest of this paper gives a brief overview of these innovative aspects. The FLINT/ML compiler is being developed based on the type-based version of the SML/NJ compiler [29]. Parts of the FLINT/ML code have also been incorporated into the most recent working release of SML/NJ (v109.27).

2 The FLINT Intermediate Language

The FLINT/ML compiler is organized around a strongly typed intermediate language named FLINT. An ML source program is first fed into the *front-end* which does parsing, elaboration, type-checking, and pattern-match compilation; the source program is translated into the FLINT intermediate format. The *middle-end* does simple dataflow optimizations and local or cross-module *type specializations*, and then produces an optimized version of the FLINT code. The *back-end* compiles FLINT into machine code through usual phases such as representation analysis [26], conventional and loop optimizations [1], CPS-based contractions and reductions [3], closure conversion [28], and machine-code generation [8]. All these compilation stages are made independent of each other so that they can also be used as compiler infrastructure for other programming languages.

Like the λ_i^{ML} calculus used in the TIL compiler [22, 30], the core of FLINT is simply a predicative variant of the Girard-Reynolds polymorphic calculus F_ω [9, 25]. Core-FLINT contains four syntactic classes: kinds (κ), constructors (μ), types (σ), and terms (e), defined as follows:

$$
\begin{array}{llll}
(\mathit{kinds}) & \kappa & ::= & \Omega \mid \kappa_1 \rightarrow \kappa_2 \mid \kappa_1 \otimes \kappa_2 \\
(\mathit{con's}) & \mu & ::= & t \mid \mathtt{Int} \mid \mathtt{Real} \mid \rightarrow(\mu_1, \mu_2) \mid \times(\mu_1, \mu_2) \mid \lambda t :: \kappa.\mu \mid \mu_1[\mu_2] \mid \otimes(\mu_1, \mu_2) \mid \Pi_1^{\kappa_1,\kappa_2}\mu \mid \Pi_2^{\kappa_1,\kappa_2}\mu \\
(\mathit{types}) & \sigma & ::= & T(\mu) \mid \forall t :: \kappa.\sigma \mid \sigma_1 \rightarrow \sigma_2 \mid \sigma_1 \times \sigma_2 \\
(\mathit{terms}) & e & ::= & x \mid i \mid f \mid \langle e_1, e_2 \rangle^{\sigma_1,\sigma_2} \mid \pi_1^{\sigma_1,\sigma_2} e \mid \pi_2^{\sigma_1,\sigma_2} e \mid \lambda x : \sigma.e \mid @^{\sigma} e_1 e_2 \mid \Lambda t :: \kappa.e \mid e[\mu]
\end{array}
$$

Here, kinds classify constructors, and types classify terms. Constructors of kind Ω name monotypes. The monotypes are generated from variables, `Int`, `Real` through the constructors $\rightarrow$ and $\times$. The application and abstraction constructors correspond to the function kind $\kappa_1 \rightarrow \kappa_2$. The pairing and selection constructors (i.e., $\otimes$, Π) correspond to the sequence kind $\kappa_1 \otimes \kappa_2$. Types in Core-FLINT include the monotypes, and are closed under products, function spaces, and polymorphic quantification. We use $T(\mu)$ to denote the type corresponding to the constructor μ (which must be of kind Ω). The terms are an explicitly typed λ-calculus with explicit constructor abstraction and application forms. The static semantics and the operating semantics for the core calculus are all standard as for λ_i^{ML} [14]. Harper and Morrisett [14, 22] have shown that type checking for predicative F_ω is decidable, and furthermore, its typing rules are consistent with its operational semantics.

```
type 'a icell = (int * 'a * aux_info) ref    (* internal hash-cell *)

datatype tkindI
  = TK_TYC                                   (* the monotype kind *)
  | TK_SEQ of tkind list                     (* the sequence kind *)
  | TK_FUN of tkind * tkind                  (* the function kind *)
  | ......................

and tycI
  = TC_VAR of DebIndex.index * int           (* tyvar in de Bruijn notation *)
  | TC_PRIM of PrimTyc.primtyc               (* primitive tycons *)
  | TC_FN of tkind list * tyc                (* constructor abstraction *)
  | TC_APP of tyc * tyc list                 (* constructor application *)
  | TC_SEQ of tyc list                       (* sequence of tycons *)
  | TC_PROJ of tyc * int                     (* projection on sequence *)
  | TC_FIX of (tkind * tyc) list * int       (* recursive tycon *)
  | TC_ABS of tyc                            (* abstract tycon *)
  | TC_IND of tyc * tycI                     (* tyc memoization node *)
  | TC_ENV of tyc * int * int * tycEnv       (* tyc suspension *)
  | ......................

and ltyI
  = LT_TYC of tyc                            (* monotype *)
  | LT_STR of lty list                       (* structure record type *)
  | LT_FCT of lty * lty                      (* functor arrow type *)
  | LT_POLY of tkind list * lty              (* polymorphic type *)
  | LT_IND of lty * ltyI                     (* lty memoization node *)
  | LT_ENV of lty * int * int * tycEnv       (* lty suspension *)
  | ......................

withtype tkind = tkindI icell                (* hash-consed tkindI cell *)
     and tyc = tycI icell                    (* hash-consed tycI cell *)
     and lty = ltyI icell                    (* hash-consed ltyI cell *)
     and tycEnv = ......                     (* tyc environment *)
```

Figure 1: Implementing kinds, constructors, and types in FLINT/ML

The actual FLINT intermediate language contains many more type and term constructs such as primitive constructors, concrete datatypes, conditional expressions, and recursive functions. Since FLINT is an explicitly typed language, adding new type constructs into FLINT does not involve any type reconstruction problem. In the long term, we intend to extend FLINT with commonly used representation types such as n-bit (trapping or non-trapping) integers and floats, type dynamic, C-like flat records (`struct`), Haskell-like thunk, etc. We believe FLINT can serve as a common intermediate format for many advanced type safe languages.

Representing FLINT types

A major challenge in implementing the FLINT intermediate language is to represent constructors and types compactly and efficiently. Type-based analysis often involve operations such as type application, normalization, and equality test. Naive implementation of these operations would lead to duplicate copying, redundant traversal, and extremely slow compilation.

We use the following techniques to optimize the representations of kinds, constructors, and types (see Figure 1 for a fragment of the FLINT definitions, written as ML datatype definitions):

- We represent type variables as de Bruijn indices [7]. Under de Bruijn notations, all constructors and types have unique representations.
- We then *hash-cons* all the kinds, constructors, and types into three separate hash-tables. Each kind (`tkind`), constructor (`tyc`), or type (`lty`) is represented as an internal hash cell (or *icell*). Each *icell* is a reference cell that contains three pieces of information: an integer hash code, a term, and a set of auxiliary information (`aux_info`). The `aux_info` for constructors and types maintains two attributes: a flag that shows whether it is already in the normal form, and if it is in the normal form, a set of its free type variables. Constructing a new type (or constructor) under this representation would involve: (1) calculating the hash code from its descendants; (2) look up the hash-table, if it is already in, we are done; otherwise, calculate the `aux_info`, and install the new *icell* into the hash-table.
- To make type reduction *lazy*, we use Nadathur's *suspension* notations [23, 24] to represent the intermediate result of *unevaluated* type applications. Intuitively, a type suspension such as `LT_ENV`(t, i, j, e) is a quadruple consisting of a term t with two indices and an environment. The first index i indicates an embedding level with respect to which variable references have been determined within the term, and the second index j indicates a new embedding level [24]. The environment e determines the bindings for the type variables. For more details about the *suspension*-based calculus, check out Nadathur [23, 24].

Figure 1 gives parts of the definitions of kind (`tkind`), constructor (`tyc`), and type (`lty`) in FLINT (using SML syntax). Here, constructor abstraction `TC_FN` and polymorphic type `LT_POLY` all abstract or quantify over a list of type variables; each type variable `TC_VAR`(i, j) is represented as a de Bruijn index i plus an integer j that indicates the exact position in the corresponding list. Suspension terms are denoted as `TC_ENV` and `LT_ENV`; when a suspension t is reduced, it will be replaced by a memoization node (i.e., `TC_IND` or `LT_IND`). Each memoization node contains a pair: the reduction result t_n and the original term t_o. We keep the original term in the memoization node so that future creations of term t_o can be directly hash-cons-ed to the same memoization node (which requires checking equality against t_o), thus saving unnecessary reductions.

The combination of these techniques have proven to be very effective. With *icell*-based hash-consing and memoization, common operations such as equality test, testing if a type is in the normal form, and finding out the set of free variables, can all be done in constant time. With the use of suspension terms, type application is always done on a *by-need* basis, and once it is done, the result will be memoized for future use. Our preliminary measurements have shown that on heavily functorized applications such as SML/NJ Compilation Manager [6], our optimized implementation is an order-of-magnitude faster than naive implementations.

3 Translation into FLINT

The *front-end* of the FLINT/ML compiler translates the entire SML'97 [21] plus higher-order modules [20]) into the FLINT intermediate language. The translation on the SML core language is quite similar to Harper and Stone's recent work on the type theoretic semantics of SML'97 [15]; the translation on the module language is, however, rather different. Unlike Harper and Stone's *Internal Language* (IL) [15], the FLINT intermediate language does not contain a separate module calculus. All module expressions and declarations (including higher-order functors) are directly translated into regular lambda terms in Core-FLINT. In the rest of this section, we summarize several important aspects about our translation; the detailed algorithm can be found in an upcoming technical report [27].

```
signature SIG = sig    type t      val x : t    end
funsig FSIG(X : SIG) = SIG

structure A : SIG = struct    type t = int      val x = 3    end
structure B :> SIG = struct    type t = real      val x = 3.5    end

functor F(X : SIG) : SIG = struct    type t = X.t -> int      fun x (z : X.t) = 1     end
functor G(F : FSIG) (A : SIG) = F(A)
structure Z = G F A
```

Figure 2: ML functors and higher-order functors

Module languages

The main challenge in translating ML modules into Core-FLINT is on how to deal with functors and higher-order functors [20]. Recent research on ML modules have focused on giving type-theoretic semantics using *dependent types* [19, 11], *translucent sums* [10], or *manifest types* [17, 18], none of these map ML modules directly into Core-FLINT-like calculus.

We have developed an algorithm that translates ML modules (including even higher-order ones) into the Core-FLINT calculus. We make two simplifications during our translation:

- Type generativity is ignored. This is fine because the elaborator in the front-end has already done the type-checking according to the SML'97 semantics [21]. Type generativity has little impact on the type-directed compilation in general.
- Opaque signature matchings (i.e., *abstractions*) are implemented as explicit coercions. An abstract type t is represented as `TC_ABS`(s) in FLINT (see Figure 1), where s is the internal implementation type. Although abstract types are supposed to hide implementation from programmers, compiler writers often need to know their internal definitions to support advanced representation analysis, pickling, and debugging. In the example code in Figure 2, structure B uses opaque signature matching (denoted by `:>` under SML'97 syntax); the type $B.t$ will be translated into `TC_ABS`(*real*) in FLINT; the value $B.x$ will be a *packed* version of the real constant 3.5.

To translate a functor declaration into Core-FLINT, we use a *phase-splitting* algorithm [27, 12]. Each ML functor, such as F in Figure 2, often plays a double role: on the "typing" aspect, it is a constructor function that maps from $X.t$ to a result constructor $X.t \rightarrow int$; on the "value" aspect, it is a function that maps $X.x$ into a result closure $\lambda z.1$. The phase-splitting can be done in a way so that the "value" aspect can refer to results produced from the "typing" aspect, but not vice versa.

Intuitively, given a functor declaration such as "*functor H(X : SIG) = StrBody*," the corresponding Core-FLINT term will be of the form: $\Lambda t_1 :: k_1 .... \Lambda t_n :: k_n.(\lambda x : \sigma_X . e_{sb})$ where $t_1, ..., t_n$ are all the *flexible* type constructors in the argument signature *SIG* (this also includes all the functor components, with each's "typing" aspect contributing as a higher-order type constructor); k_i is the kind of t_i for $i = 1, ..., n$; and σ_X is the type of the value component in structure X; finally, e_{sb} is the corresponding Core-FLINT term for the *StrBody*.

Similarly, given a functor application such as "*H(S)*", the corresponding Core-FLINT term will be of the form: $@(e_H[\mu_1][\mu_2]...[\mu_n])(e_S)$ where $@e_1 e_2$ is the Core-FLINT syntax for function applications (applying e_1 to e_2); symbol e_H and e_S denote the corresponding Core-FLINT terms for functor H and structure S (matched against signature *SIG*); and $\mu_1, ..., \mu_n$ are the actual constructor definitions for those flexible components $(t_1, ...t_n)$ in signature *SIG*.

We omit the details of our algorithm due to space limitations. Instead, we give the corresponding Core-FLINT expressions for several example modules in Figure 2. Functor declaration F is translated into an expression e_F with type σ_F; its "typing" aspect is a constructor μ_F:

$$\begin{array}{rrl} \text{“value” aspect:} & e_F = & \Lambda t :: \Omega.\lambda x : t.(\lambda z : t.1) \\ e_F\text{'s type:} & \sigma_F = & \forall t :: \Omega.T(t \rightarrow (t \rightarrow \texttt{Int})) \\ \text{“typing” aspect:} & \mu_F = & \lambda t :: \Omega.t \rightarrow \texttt{Int} \end{array}$$

The higher-order functor declaration G is translated into an expression e_G with type σ_G; its "typing" aspect is a constructor μ_G:

$$\begin{array}{rrl} \text{“value” aspect:} & e_G = & \Lambda t_f :: \Omega \rightarrow \Omega.\Lambda t_a :: \Omega.\ \lambda f : \sigma_f.\lambda a : T(t_a).(@^{T(t_a)}(f[t_a])a) \\ e_G\text{'s type:} & \sigma_G = & \forall t_f :: \Omega \rightarrow \Omega.\forall t_a :: \Omega.\ (\sigma_f \rightarrow (T(t_a) \rightarrow T(t_f[t_a]))) \\ \text{where} & \sigma_f = & \forall t :: \Omega.T(t \rightarrow t_f[t]) \\ \text{“typing” aspect:} & \mu_G = & \lambda t_f :: \Omega \rightarrow \Omega.\lambda t_a :: \Omega.t_f[t_a] \end{array}$$

The functor application "GFA" is translated into an expression e_Z with type σ_Z:

$$\begin{array}{rrl} \text{“value” aspect:} & e_Z = & (@(@(e_G[\mu_F][\texttt{Int}])e_F)3) \\ e_Z\text{'s type:} & \sigma_Z = & (\texttt{Int} \rightarrow \texttt{Int}) \end{array}$$

Datatype specifications

FLINT/ML translates both datatype declarations and specifications into recursive sum types (i.e, `TC_FIX` in Figure 1). The treatment of datatype specifications is different from Harper and Stone's [15]. For example, in the following functor declaration, t is a datatype specification inside the argument signature,

```
functor F(A : sig datatype t = A1 | A2 | ... | An
                             | B1 of T1
                               ......
                             | Bm of Tm
              end)
  = struct ... end
```

Harper and Stone [15] translates the datatype spec into a signature consisting of an *abstract* representation type plus a list of operations to create, destroy, and analyze values of that type. This treatment, unfortunately, relaxes the constraint on the argument spec t in that F can now apply to a structure of any type t (not necessarily a recursive sum type) plus a set of properly typed operations. In the corresponding intermediate code for the body of functor F, all injection and projection functions for constructor `A1`, ..., `Bm` must now be implemented as *abstract* functions (they cannot be inlined because their implementation won't be known until F is applied).

In FLINT/ML, we make datatype declarations and specifications *concrete* all the time. The datatype spec t in the previous example is considered as a type abbreviation for the corresponding recursive sum type. The FLINT intermediate language provides the same set of generic injection (inj_i^t) , projection ($proj_i^t$), and roll and unroll operators for recursive sum types as in [15]. All occurrences of `A1`, ..., `Bm` in the body of functor F are implemented by the corresponding inj_i^t and $proj_i^t$ primitives.

Most often, the primitive inj_i^t and $proj_i^t$ operators can be determined and inlined at compile time. For example, in the following ML code,

```
functor H(S: sig type 'a t
                 datatype 'a foo = A | B of 'a t
                 datatype 'a bar = C | D of 'a * 'a bar
             end) = struct  ...  end

structure T = struct datatype 'a foo = A | B of 'a * 'a foo
                     datatype 'a bar = C | D of 'a * 'a bar
                     type 'a t = 'a * 'a foo
              end
```

the implementation of data constructors such as `S.D`, `T.B`, and `T.D` are all known at compile time (they can be implemented as flat *untagged* record). The implementation of `S.B` is, however, not clear because t won't be known until functor H is applied. In FLINT/ML, the injection and projection functions for `S.B` are implemented as type-directed primitives, which will check the runtime value of t to decide whether to use flat record or indirect pointer representations.

This scheme also solves the nasty *datatype representation* problem raised by Appel [4]. In the previous example, applying functor H to structure T is not allowed in the old SML/NJ compiler [29, 5] because data representation for `S.foo` is inconsistent with that for `T.foo`; this is no longer a problem in FLINT/ML. Furthermore, unlike other solutions for this problem [2], under our scheme, the implementation of formal data constructors remains to be *concrete* most of the time (e.g., `S.D`).

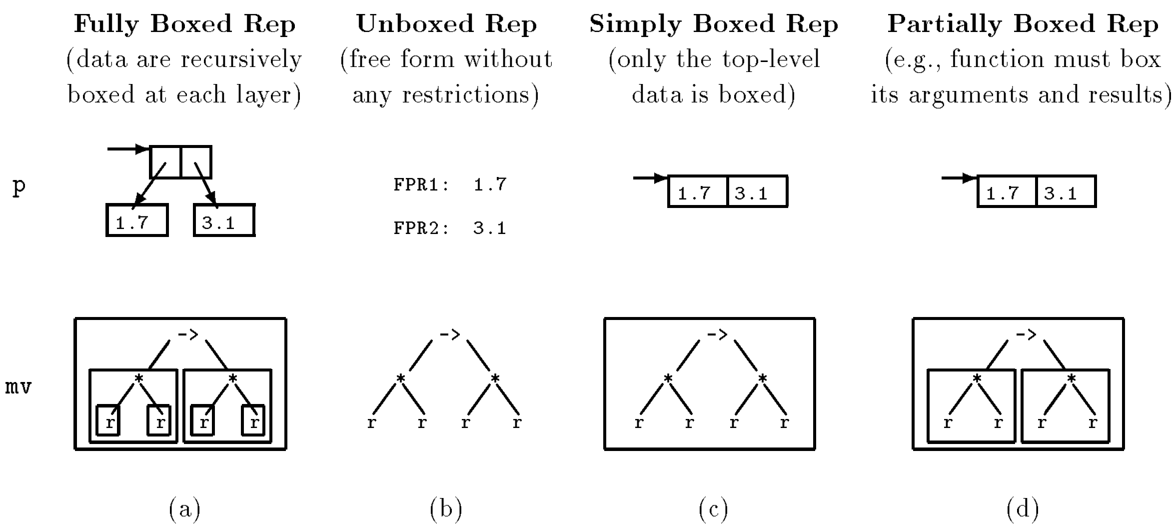
4 Compiling FLINT

The back-end of the FLINT/ML compiler translates the FLINT intermediate code into the machine code. One novel aspect in our back-end is to use the new *flexible representation analysis* technique [26] to compile the polymorphic functions and functors. Under flexible representation analysis, recursive and mutable data objects can use unboxed representations without incurring expensive runtime cost on heavily polymorphic code. In contrast, the *coercion-based* approach used in Gallium [16] and SML/NJ [29] does not support unboxed representations on recursive and mutable objects; the *type-passing* approach used in TIL [30] does handle all data objects, but it involves heavy-weight runtime type analysis and code manipulations.

The basic idea of our flexible scheme can be illustrated using the following simple example:

```
fun quad (f,x) = let val z = f(f(f(f(x)))) in (z::z::nil) end
val p = (1.7, 3.1)
fun mv (x,y) = (x * 3.1, y * 2.7)
```

Here, function `quad` is a polymorphic function with type $\forall\alpha.(((\alpha \rightarrow \alpha) * \alpha) \rightarrow \alpha\ \texttt{list})$; value `p` is a pair of floats, and function `mv` has monomorphic type $(\texttt{real} * \texttt{real}) \rightarrow (\texttt{real} * \texttt{real})$.



We use boxed type trees to illustrate boxing for complicated type structures (e.g., `mv`'s). Each box refers to one boxing layer. ML type `real` is abbreviated as the symbol `r`.

Figure 3: Comparison of Various Data Representations.

Under Leroy's scheme [16], both `p` and `mv` can use efficient *unboxed* representations (see Figure 3b): value `p` can stay in two floating-point (FP) registers and function `mv` can freely pass the arguments and return the results in two FP registers. When monomorphic objects are passed to polymorphic contexts (as in `quad(mv, p)`), they are coerced into *fully boxed* representations (see Figure 3a). Leroy's technique does not handle recursive and mutable objects because they can not be coerced efficiently or correctly.

Under our new scheme, instead of doing full-boxing, we use the *simply boxed* representation (see Figure 3c) or other *partially boxed* representations (see Figure 3d) to represent the polymorphic objects. Intuitively, a simply boxed object just boxes the top layer of the data structure so that the entire object can be referenced as a single-word pointer. Simple-boxing is generally much cheaper than full-boxing, and most of the time, it is just an identity function because the natural representations of many "unboxed" objects (e.g., lists, closures, records, arrays) are already simply boxed. Simple-boxing solves the problem of recursive and mutable types because any simply boxed object can be easily unboxed (at the top layer) before being *cons*-ed onto lists or put inside arrays.

Partial-boxing is similar to simple-boxing, except it also maintains the invariant that all function arguments and results are also partially boxed. Both simple-boxing and partial-boxing are quite tricky to implement because the coercion may also rely on the runtime type information. For example, coercing an object of type $\beta * \gamma$ into type α would involve first unboxing the β and γ fields, and then pairing them up based on the actual types β and γ have at runtime.

Unlike in the type-passing approach [30, 13], all polymorphic values in our flexible scheme are indeed always *boxed.* This dramatically simplifies the implementation of polymorphism, because all polymophic objects can be manipulated just as a single-word data. In the type-passing approach, polymorphic objects are not always boxed, so all polymorphic primitives become dependent on runtime types.

5 Summary and Conclusions

All techniques discussed in this paper have been implemented in the FLINT/ML compiler. Parts of the FLINT/ML code have also been incorporated into the most recent working release of SML/NJ (v109.27). When compared with the old type-based SML/NJ compiler [29], FLINT/ML gives better performance (about 20% speedup) on benchmarks involving recursive and mutable types. Benchmarks involving heavy polymorphic code remain as efficient as before.

The most important contribution of FLINT/ML lies, however, on its ability to compile the entire SML'97 plus higher-order modules into a strongly typed intermediate language (FLINT). Flexible representation analysis provides an efficient way to compile polymorphism in FLINT without restricting the data representations on monomorphic code. In the future, we plan to expand and evolve FLINT into a common typed intermediate format for the advanced type-safe languages.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] W. E. Aitken and J. H. Reppy. Abstract value constructors. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 1–11, June 1992. Longer version available as Cornell Univ. Tech. Report.

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

[4] A. W. Appel. A critque of Standard ML. *Journal of Functional Programming*, 3(4):391–429, October 1993.

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

[6] M. Blume. A compilation manager for SML/NJ. as part of SML/NJ User's Guide, 1995.

[7] N. de Bruijn. A survey of the project AUTOMATH. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Edited by J. P. Seldin and J. R. Hindley, Academic Press, 1980.

[8] L. George, F. Guillaume, and J. Reppy. A portable and optimizing backend for the SML/NJ compiler. In *Proceedings of the 1994 International Conference on Compiler Construction*, pages 83–97. Springer-Verlag, April 1994.

[9] J. Y. Girard. *Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur.* PhD thesis, University of Paris VII, 1972.

[10] R. Harper and M. 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.

[11] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Trans. Prog. Lang. Syst.*, 15(2):211–252, April 1993.

[12] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 341–344, New York, Jan 1990. ACM Press.

[13] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. Technical Report CMU-CS-94-185, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1994.

[14] R. Harper and G. 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.

[15] R. Harper and C. Stone. A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU-CS-96-136R, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1996.

[16] X. Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, Jan 1992. ACM Press. Longer version available as INRIA Tech Report.

[17] X. 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.

[18] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 142–153, New York, Jan 1995. ACM Press.

[19] D. MacQueen. Using dependent types to express modular structure. In *Proc. 13th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 277–286. ACM Press, 1986.

[20] D. B. MacQueen and M. Tofte. A semantics for higher order functors. In *The 5th European Symposium on Programming*, pages 409–423, Berlin, April 1994. Spinger-Verlag.

[21] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Massachusetts, 1997.

[22] G. Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1995. Tech Report CMU-CS-95-226.

[23] G. Nadathur. A notation for lambda terms II: Refinements and applications. Technical Report CS-1994-01, Duke University, Durham, NC, January 1994.

[24] G. Nadathur and D. S. Wilson. A representation of lambda terms suitable for operations on their intensions. In *1990 ACM Conference on Lisp and Functional Programming*, pages 341–348, New York, June 1990. ACM Press.

[25] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pages 408–425. Springer-Verlag, Berlin, 1974.

[26] Z. Shao. Flexible representation analysis. Technical Report YALEU/DCS/RR-1125, Dept. of Computer Science, Yale University, New Haven, CT, April 1997.

[27] Z. Shao. Typed cross-module compilation. Technical Report YALEU/DCS/RR-1126, Dept. of Computer Science, Yale University, New Haven, CT, May 1997.

[28] Z. Shao and A. W. Appel. Space-efficient closure representations. In *1994 ACM Conference on Lisp and Functional Programming*, pages 150–161, New York, June 1994. ACM Press.

[29] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proc. ACM SIGPLAN '95 Conf. on Prog. Lang. Design and Implementation*, pages 116–129. ACM Press, 1995.

[30] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 181–192. ACM Press, 1996.