Implementing Typed Intermediate Languages*

Zhong Shao Christopher League Stefan Monnier
Dept. of Computer Science
Yale University
New Haven, CT 06520
{shao,league,monnier}@cs.yale.edu.

Abstract

Recent advances in compiler technology have demonstrated the benefits of using strongly typed intermediate languages to compile richly typed source languages (e.g., ML). A type-preserving compiler can use types to guide advanced optimizations and to help generate provably secure mobile code. Types, unfortunately, are very hard to represent and manipulate efficiently; a naive implementation can easily add exponential overhead to the compilation and execution of a program. This paper describes our experience with implementing the FLINT typed intermediate language in the SML/NJ production compiler. We observe that a type-preserving compiler will not scale to handle large types unless all of its type-preserving stages preserve the asymptotic time and space usage in representing and manipulating types. We present a series of novel techniques for achieving this property and give empirical evidence of their effectiveness.

1 Introduction

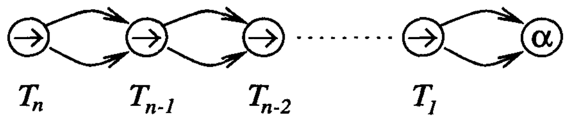
Compilers for richly typed languages (e.g., ML [21]) have long used variants of the untyped λ-calculus [2, 10] as their intermediate languages. An untyped compiler first type-checks the source program, and then translates the program to the intermediate language, discarding all the type information. Types are used to ensure that the program will not "go wrong" at run time, but they do not affect the rest of compilation and execution in any way.

Recent advances in compiler technology have demonstrated many distinct advantages of using strongly typed intermediate languages to compile richly typed source languages. A type-preserving compiler type-checks the source program, but then translates both the program and the (inferred) type information into the intermediate language. The rest of the compiler can use types to guide advanced optimizations [28, 16, 22, 34, 6] and to help generate provably secure mobile code [13, 26, 19, 23, 17]. The compiler can also propagate the type information into the target code to support sophisticated run-time type dispatches and garbage collection [14, 22, 34, 43].

Unfortunately, type information is very hard to represent and manipulate efficiently, especially when the underlying type system involves ML-like polymorphic types and module types [21]. A naive implementation can easily add exponential overhead to the compilation and execution of a program. For example, in the following ML program:

```
fun f x = x
fun toy() =
   let fun g y = ((((f f) f) f) ... f) y
    in g 3
   end
```

the identity function f has polymorphic type $\forall\alpha.\alpha \to \alpha$. Suppose we apply f to itself n times as shown. According to the ML type inference algorithm [5], the rightmost f has type $T_1 = \alpha \to \alpha$, while the leftmost f gets instantiated to $T_n = T_{n-1} \to T_{n-1}$. Clearly, representing T_n as a tree-like structure would require $O(2^n)$ space, so a sufficiently small n (e.g., 30) would wreck the efficiency of the compiler. To avoid such exponential blowup, we must represent and manipulate T_n as a linear-sized dag:



In fact, we must ensure that all type-related operations in the compiler (including those at run time if we pass types there) would handle such large types in the same way. For instance, in the let body above, when g is specialized to int→int, we need to apply a substitution (from α to int) to all instances of T_i; clearly, we must traverse the dag in linear time and preserve its shape.

Although the preceding example is a bit contrived,[1] large

*This research was sponsored in part by the Defense Advanced Research Projects Agency ITO under the title "Software Evolution using HOT Language Technology," DARPA Order No. D888, 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.

[1]It is well known that ML type inference can take exponential time and space on certain kinds of ML programs [20]; the toy function defined here, however, does not belong to this category. An untyped compiler could compile the toy function without any problem.

Ratio of tree size over dag size	**CM**			**eXene**		
	number of types	average tree size	percentage of total size	number of types	average tree size	percentage of total size
1-3	12,210	42	16.39%	32,376	58	1.91%
4-15	1,304	469	19.22%	7,547	998	7.57%
16-63	102	9,224	29.54%	2,889	8,866	25.73%
64-255	47	21,704	32.03%	818	75,153	61.74%
256-1023	2	45,518	2.86%	8	379,315	3.05%

For each type built during the compilation, we calculate the size of its tree representation and dag representation (in number of nodes). The ratio of these two shows the amount of savings of the dag representations. We then use the range of this ratio (2^i to $2^{i+2}-1$, where $i = 0,\ldots,4$) to classify all the types; for each category, we list the number of its members, the average size of the tree representations, and the percentage of its total size over that of all categories.

Figure 1: A profile of compile-time type information.

types are ubiquitous in real-world ML applications. For example, a 200-line ML program `cm.sml` in the SML/NJ compilation manager (CM) [4] contains more than 36 functor applications and more than 80 structure references; each of these modules may contain a dag of sub-structures or functors. Figure 1 gives a profile of types built while compiling two large ML applications in our type-preserving compiler (see later sections for details about the compiler). Here, CM is the compilation manager [4] and eXene is an ML-based X window system tool-kit [31]. If we use tree representations, a single type can contain more than 45,518 nodes for CM and 379,315 nodes for eXene. These large types can be dramatically reduced in size if we use dag representations. For example, under CM, 32.03% of the space occupied by types can be improved by a factor of at least 64 when we use dag representations. For eXene, the savings are even more dramatic. Of course, no real compiler will use the dumb tree representations all the time, but the profile does show that any loss of sharing in type representations could potentially incur huge costs to the compilation time and space usage.

This paper describes our experience with implementing the FLINT typed intermediate language [36] in the SML/NJ production compiler [39, 3]. FLINT is based on a predicative variant of the polymorphic λ-calculus F_ω [12, 32, 15], extended with a rich set of primitive types and functions. FLINT supports both polymorphic types and higher-order type constructors, so the type language itself is a full-scale λ-calculus. To support various type-directed optimizations [14, 34], we perform a large number of type-related operations during compilation. The main challenge is to represent complex FLINT types (which can be arbitrary lambda terms) as compact dags so that common type-related operations (e.g., lambda reductions, equality) can always work efficiently and yet still preserve sharing.

More generally, we believe that a type-preserving compiler will not scale to handle large types unless all of its type-preserving stages can preserve the asymptotic time and space usage in representing and manipulating types. To achieve this property, we present a novel and efficient representation scheme for the FLINT type calculus. Our main idea is to combine hash-consing, memoization, and advanced lambda encoding [24, 1, 9] to ensure that (1) types are always represented as dags; (2) type reductions are done on a *by-need* basis; and (3) the cost of handling types is proportional to the size of the dag representations. In a companion paper [33], we have presented a new optimal type-lifting algorithm that lifts all run-time type constructions to the top level; in fact, we can guarantee that the number of types built at run time is a compile-time constant; furthermore, all of them are represented as efficiently as their compile-time counterparts.

The main contributions of this paper are:

- As far as we know, our work is the first comprehensive study on how to build *scalable* implementations of type-preserving production compilers. Several existing compilers [41, 27, 42] have also used typed intermediate languages, but none of them have attempted to scale their implementations to handle large types; in fact, all these compilers have reported extremely slow compilation times as a result of keeping types during compilation.[2]
- We combine hash-consing, memoization, and advanced lambda encoding [24, 1, 9] to support efficient type representation and manipulation. Although each of these techniques has been researched and implemented before, nobody has ever tried to combine them to represent compiler type information. Combining these techniques is crucial yet non-trivial, as we will demonstrate in Section 5 and Section 7.
- We describe several different ways of representing type variables bound in the term languages and then compare their performance. Representing type variables as de Bruijn indices is faster but it also makes type manipulation harder. We show that using explicit names to represent type variables might be a more desirable alternative.
- All techniques discussed in this paper have been implemented and incorporated into the SML/NJ production compiler since version 109.24 (January 1997). The resulting compiler has been used and tested world-wide on a large number of ML applications for more than 14 months. We have not received any complaints about the compilation time after we switched to the type-preserving implementation. We are not aware of any

[2]Although GHC makes little use of its type information in the back end, it still runs out of memory when compiling the toy benchmark.

other type-preserving ML compilers that can handle large applications such as CM and eXene.

- To verify the effectiveness of these techniques, we have measured and compared several versions of the SML/NJ compiler on a variety of benchmark programs. The combination of these techniques can reduce the total compilation time by up to 72% on large applications (a reduction of 93% in the type-preserving phases).
- We also present a detailed comparison between our scheme and the **lettype** scheme used in the TIL/ML compiler (also informally described in Tarditi's thesis [40]).

2 Related Work

Typed intermediate languages have received much attention lately, especially in the HOT (higher-order and typed) language community. However, recent work [14, 22, 36, 30, 8, 29, 23] has mostly focused on the theoretical foundations or other language design issues. This paper complements previous work by showing that typed intermediate languages can indeed have *practical* and *scalable* implementations, but only if extreme care is taken. In fact, most of the techniques described in this paper have been incorporated into the SML/NJ production compiler since version 109.24 (January 1997). Many results reported here are inspired by feedback from the SML/NJ user community.

Several existing compilers such as TIL [41], GHC [27], and ML-Kit [42] have also used an F_ω-like calculus as their typed intermediate languages. However, none of them has seriously addressed the problem of how to handle large types, nor do they support efficient run-time type passing.

The suspension-based lambda encoding used in our implementation is directly borrowed from Nadathur's recent work on efficient lambda representations [25, 24]. In addition to doing an in-depth theoretical study of the underlying encoding calculus, Nadathur [24] has also used his encoding to implement the λ-Prolog system. The main contribution of our work is to combine Nadathur's encoding with hash-consing and memoization, and then apply it to the context of typed intermediate languages. Combining these techniques is non-trivial because of the presence of higher-order types and the need to memoize intermediate reduction results.

Explicit substitutions [9, 1] is another related lambda encoding scheme. Cardelli's Quest compiler [1] contains an implementation of this encoding; however, he did not combine it with other techniques we used. Nor was he working in the context of type-preserving compilers.

Shao and Appel [39] used hash-consing to enforce dag representations for types; however, their intermediate language is only monomorphically typed, so it is much easier to support than FLINT-like languages. Tarditi [40] used the **lettype** constructs (in both the constructor calculus and the term language) to A-normalize [10] all types in order to express sharing explicitly. But he relies on a separate common sub-expression elimination phase to identify the sharing information. This amounts to hash-consing with the disadvantage that it comes too late (huge redundant types have already been built) and it does not guarantee that further redundancies will not be introduced later in the compilation process. So it is not clear that the **lettype** scheme will address issues such as compilation time in the face of real-world applications. Finally, **lettype** has poorly understood theoretical foundations; we do not know of any notion of normal form [3] in the context of **lettype**, for example. A more detailed comparison between our scheme and the **lettype** scheme is given in Section 8.

$$\begin{array}{llcl}
(\mathit{kind}) & \kappa & ::= & \Omega \mid \kappa_1 \rightarrow \kappa_2 \\
(\mathit{tycon}) & \mu & ::= & t \mid \mathtt{Int} \mid \mu_1 \rightarrow \mu_2 \mid \lambda t :: \kappa.\mu \mid \mu_1[\mu_2] \\
(\mathit{type}) & \sigma & ::= & T(\mu) \mid \sigma_1 \rightarrow \sigma_2 \mid \forall t :: \kappa.\sigma \\
(\mathit{term}) & e & ::= & i \mid x \mid \lambda x : \sigma.e \mid @x_1 x_2 \\
 & & \mid & \Lambda t :: \kappa.e \mid x[\mu] \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2
\end{array}$$

Figure 2: Syntax of the Core-FLINT calculus.

3 An Overview of FLINT

The core language of FLINT is based on a predicative variant of the Girard-Reynolds polymorphic λ-calculus F_ω [12, 32], with the term language written in A-normal form [10]. It contains the following four syntactic classes: kinds (κ), type constructors (μ), types (σ), terms (e), as shown in Figure 2. Here, kinds classify type constructors, and types classify terms. Constructors of kind Ω name monotypes. The monotypes are generated from variables, from **Int**, and through the $\rightarrow$ constructor. As in F_ω, the application and abstraction constructors (i.e., $\mu_1[\mu_2]$ and $\lambda t :: \kappa.\mu$) correspond to the function kind $\kappa_1 \rightarrow \kappa_2$. Types in Core-FLINT include the monotypes, and are closed under function spaces and polymorphic quantification. We use $T(\mu)$ to denote the type corresponding to the constructor μ when μ is of kind Ω. As in F_ω, the term language is an explicitly typed polymorphic λ-calculus (but written in A-normal form); both type abstraction and type application are explicit.

The actual FLINT language contains other familiar constructs such as record, recursive datatype, and a rich set of primitive types and operators. Large types mainly come from ML-style modules (which are represented as FLINT records [37]) and recursive datatypes, but the challenge of implementing FLINT still lies on how we handle three forms of type abstractions, i.e., constructor function ($\lambda t :: \kappa.\mu$), polymorphic type ($\forall t :: \kappa.\sigma$), and polymorphic function ($\Lambda t :: \kappa.e$). We present our solutions in Sections 5 and 6.

The structure of our type-preserving compiler is very similar to that of conventional untyped compilers. Programs written in the source languages (e.g., ML) are first fed into a language-specific *front end* which does parsing, elaboration, type-checking, and pattern-match compilation; the source program is then translated into the FLINT typed intermediate format. The *middle end* does conventional dataflow optimizations, type specializations, and λ-calculus-based contractions and reductions, producing an optimized version of the FLINT code. The *back end* compiles FLINT into machine code through the usual phases such as representation analysis [34], safe-for-space closure conversion [38], register allocation, instruction scheduling, and machine-code generation [11].

[3] Of course, we could always expand out the **lettype** definitions to get to a normal form, but this eliminates the benefit of using **lettype** and is equivalent to using tree representations.

```
signature LTYEXTERN = sig
(* abstract types *)
   type tkind  (* κ *)
   type tyc    (* μ *)
   type lty    (* σ *)
(* constructors *)
   val tcc_int    : tyc                   (* Int *)
   val tcc_var    : tvar -> tyc           (* t *)
   val tcc_arrow  : tyc * tyc -> tyc      (* μ → μ *)
   val tcc_fn     : tkind * tyc -> tyc    (* λκ.μ *)
   val tcc_app    : tyc * tyc -> tyc      (* μ[μ] *)
(* selectors *)
   val tcd_var    : tyc -> tvar
   val tcd_arrow  : tyc -> tyc * tyc
   val tcd_fn     : tyc -> tkind * tyc
   val tcd_app    : tyc -> tyc * tyc
(* predicates *)
   val tcp_int    : tyc -> bool
   val tcp_var    : tyc -> bool
   val tcp_arrow  : tyc -> bool
   val tcp_fn     : tyc -> bool
   val tcp_app    : tyc -> bool
(* utility functions *)
   val tc_eqv     : tyc * tyc -> bool
   val tc_print   : tyc -> string
   ...
end (* LTYEXTERN *)
```

Figure 3: Interface to the FLINT constructor language (μ). Some constructor forms are omitted. Similar interfaces exist for FLINT types (σ) and kinds (κ).

4 Implementation Criteria

In this section, we list the goals that guided the implementation of the FLINT type language, and we describe its interface. We present the implementation details in Section 5. The following criteria are, in our experience, important for an efficient implementation of a typed intermediate language.

Compact space usage. As demonstrated in Section 1, large types are ubiquitous in real-world ML applications. For this reason, it is imperative that we represent these types efficiently. Fortunately, large types are also highly redundant, so a well-constructed dag representation can be quite compact. The representation should, however, come with either a guarantee that *all* such redundancy is exploited, or empirical evidence showing that, in practice, types remain compact even as they are manipulated and transformed.

Linear-time traversal of types. Compact representations are not enough to ensure efficiency in a type-preserving compiler. Many operations on types (e.g., substitution and reduction) require traversing the graph. If we are not careful, these operations might traverse isomorphic subgraphs multiple times, even though they share the same representation. In order to maintain reasonable compilation time, such operations must traverse the representation linearly.

Fast equality. Checking the equivalence of two types can be non-trivial, because there are many ways to represent the same type. Equality checking is often used by type-directed optimizations. For example, representation analysis [34] uses equality to determine where wrapping is necessary. Moreover, two compelling operations made possible by type-preserving compilation perform equality tests repeatedly: type-checking intermediate phases and certifying object code [26, 23]. Thus, the implementation should support efficient equality tests.

Simple Interface. The software engineering benefits of hiding implementation details from clients are widely recognized. Besides concealing the tricks used to meet the other criteria, we would like clients to treat each type as its intension; different representations of a single type should all look the same. There are two ways to achieve such an interface. Either all types passed across the interface are in normal form (corresponding to eager reduction), or the top node of a type is the same as for the normalized version (*weak head normal form*), and the rest is normalized on demand.

Our implementation meets all of these goals. We guarantee sharing by hash-consing and storing each type in a global table. Isomorphic types *always* share their representation, regardless where they appear or how they were constructed. All reductions and substitutions are guaranteed to traverse the representation linearly (because these important operations are specifically supported by the implementation). For clients implementing other transformations, we provide a memoizing *fold* function that is guaranteed to traverse the representation linearly.

For equality testing, thanks to our guarantee that isomorphic types share the same representation, types in normal form can be compared very quickly using pointer equality. If the types are not in normal form and pointer equality fails, then we reduce the types to *weak head normal form*, check if the heads have the same shape, and continue recursively on the sub-terms. In practice, this leads to very cheap equality tests. Complete type-checking of the intermediate code after every phase does not incur noticeable overhead.

Figure 3 gives part of the FLINT type language interface. All operations on FLINT types (μ) are done through a set of basic primitives: constructor functions create types from their components; predicates test for particular constructs; selectors project the components, assuming an appropriate construct is given. Additionally, the interface contains functions for equivalence testing, pretty-printing, etc.

The interface behaves as if all types are kept in normal (fully reduced) form, even though the underlying implementation uses lazy reduction. For example, suppose we create a type `t` by applying the identity function to `Int`. Then, `tcp_app(t)` will return false, whereas `tcp_int(t)` will return true.

5 Representing Types

Now we turn to a detailed explanation of our implementation techniques. We show how to represent complex FLINT types as compact dags and make the costs of all type-related operations (e.g., substitution, equality) proportional to the dag size. We will focus our discussion on the FLINT constructors (μ) only, though these techniques apply to the FLINT types (σ) and kinds (κ) as well. In fact, the issues involved in implementing polymorphic types ($\forall t :: \kappa.\sigma$) are precisely same as those for higher-order constructors ($\lambda t :: \kappa.\mu$).

5.1 Suspension-based lambda encoding

The first challenge in representing FLINT constructors is to choose an appropriate encoding for efficient manipula-

(*r1*)	$(\lambda\kappa.\mu_1)[\mu_2] \Longrightarrow \mathtt{Env}(\mu_1, (1, 0, (\mu_2, 0) :: nil))$
(*r2*)	$\mathtt{Env}(\mu, (0, 0, nil)) \Longrightarrow \mu$
(*r3*)	$\mathtt{Env}(\mu, (i, j, env)) \Longrightarrow \mu$ if μ is closed, i.e., it has no free type variables)
(*r4*)	$\mathtt{Env}(\#n, (i, j, env)) \Longrightarrow \#(n - i + j)$ if $n > i$
(*r5*)	$\mathtt{Env}(\#n, (i, j, env)) \Longrightarrow \#(j - j')$ if $n \leq i$ and the n-th element of env is j'
(*r6*)	$\mathtt{Env}(\#n, (i, j, env)) \Longrightarrow \mathtt{Env}(\mu, (0, j - j', nil))$ if $n \leq i$ and the n-th element of env is (j', μ)
(*r7*)	$\mathtt{Env}(\mathtt{Int}, \rho) \Longrightarrow \mathtt{Int}$
(*r8*)	$\mathtt{Env}(\mu_1 \rightarrow \mu_2, \rho) \Longrightarrow \mathtt{Env}(\mu_1, \rho) \rightarrow \mathtt{Env}(\mu_2, \rho)$
(*r9*)	$\mathtt{Env}(\mu_1[\mu_2], \rho) \Longrightarrow (\mathtt{Env}(\mu_1, \rho))[\mathtt{Env}(\mu_2, \rho)]$
(*r10*)	$\mathtt{Env}(\lambda\kappa.\mu, (i, j, env)) \Longrightarrow \lambda\kappa.\mathtt{Env}(\mu, (i + 1, j + 1, j :: env))$
(*r11*)	$\mathtt{Env}(\mathtt{Env}(\mu, (i, j, env)), (0, j', nil)) \Longrightarrow \mathtt{Env}(\mu, (i, j + j', env))$

Figure 4: Type reductions under suspension-based lambda encoding.

tion. Under the syntax in Figure 2, testing the equality of α-convertible constructors such as $\mu_1 = \lambda t_1 :: \Omega.t_1 \rightarrow t_1$ and $\mu_2 = \lambda t_2 :: \Omega.t_2 \rightarrow t_2$ is non-trivial. We use de Bruijn indices [7] to represent type variables, so that α-equivalent constructors always have the same representation. For example, both μ_1 and μ_2 are represented as $\lambda\Omega.(\#1 \rightarrow \#1)$. The λ no longer binds any named type variables (though the kind is still retained). Instead, we use a positive integer $\#n$ to denote the variable bound by the nth surrounding λ-binder.

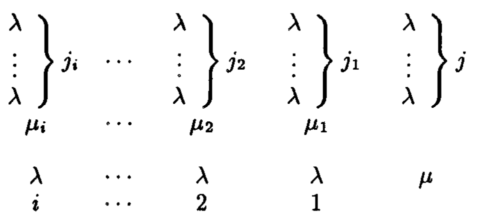
Another important requirement is that type reduction should be done *lazily*. To achieve this, we enrich the constructor calculus to support a new ***suspension*** term [25, 24] of the form $\mathtt{Env}(\mu, \rho)$. Intuitively, a suspension represents an *unevaluated* type "$\rho(\mu)$"; it corresponds to the intermediate result of some unevaluated type applications. The substitutions involved (ρ) are also known as ***explicit substitutions*** [1, 9]:

$$
\begin{array}{llll}
(constructor) & \mu & ::= & \#n \mid \mathtt{Int} \mid \mu_1 \rightarrow \mu_2 \\
 & & \mid & \lambda\kappa.\mu \mid \mu_1[\mu_2] \mid \mathtt{Env}(\mu, \rho) \\
(substitution) & \rho & ::= & (i, j, env) \\
(environment) & env & ::= & nil \mid j :: env \mid (j, \mu) :: env
\end{array}
$$

Following Nadathur [24, 25], we represent each such substitution as a triple (i, j, env) where the first index i indicates the current embedding level of bound type variables, the second index j indicates its new embedding level, and the environment env contains the actual bindings of all i bound variables. Each entry in the environment is represented as a pair (j', μ') or as an integer j' (which has same meaning as $(j', \#0)$); in either case, j' denotes the definitional depth of type μ'. Figure 5 shows the relationship among these components for a type $\mathtt{Env}(\mu, (i, j, env))$, assuming all environment entries of form j' are represented as $(j', \#0)$, and the environment env is equal to $(j_1, \mu_1) :: \ldots :: (j_i, \mu_i) :: nil$.

For example, the standard β-contraction $(\lambda\kappa.\mu_1)[\mu_2]$ results in a constructor of the form $\mathtt{Env}(\mu_1, \rho_0)$ where $\rho_0 = (1, 0, (0, \mu_2) :: nil)$. This represents the following fact: the constructor μ_1, which was originally in the scope of 1 abstraction, is now to be thought of as being in the scope of none; μ_2, originally in the scope of 0 abstractions, is to be substituted for the first free variable in μ_1.

Figure 4 gives the set of type reductions used in our λ encoding. Here, Rule (*r1*) turns a type application into the suspension form. Rules (*r2*) and (*r3*) are two straightforward optimizations, capturing the fact that applying an empty substitution to a type or applying a substitution to a closed



λ ⋮ λ j_i … λ ⋮ λ j_2 λ ⋮ λ j_1 λ ⋮ λ j
μ_i … μ_2 μ_1
λ … λ λ μ
i … 2 1

Figure 5: Illustration of a suspension type

type should have zero effect. Because we memoize the set of free variables in our type representations (see Section 5.2), it is very easy to check whether a type is closed. Rules (*r4*) to (*r6*) show how we adjust and substitute each de Bruijn index-based type variable: for variables that are bound outside the current binding level (i), the new de Bruijn index would be $(n - i) + j$; for variables bound in the current environment, we find out its corresponding mapping and adjust the result from its definitional level j' to the new embedding level j. Rules (*r7*) to (*r10*) push the substitution recursively into the subterm of each type; for type functions ($\lambda\kappa.\mu$), we need to add a new entry into the current environment (see Rule (*r10*)). Rule (*r11*) is a simple optimization to merge two nested substitutions. Notice that because all intermediate results are expressible in our calculus, the reduction rules do not involve any external substitution machinery. More details about the suspension-based calculus can be found in Nadathur's excellent paper [24].

5.2 Hash-consing and memoization

After we choose the appropriate encoding scheme, we ***hash-cons*** all FLINT kinds, type constructors (including substitutions), and types into three separate hash tables. Under hash-consing, all FLINT types built during the compilation are guaranteed to use the most compact dag representation. Because we are using de Bruijn notation, type variables are represented as integers and all α-convertible types have identical representations, which allow them to be collapsed via hash-consing.

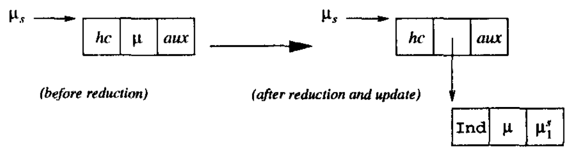
For each hash entry, we use ***weak pointers*** so that if an element in the hash table is no longer used anywhere else, it will be garbage collected. Internally, each constructor

μ is now accessed indirectly via an updateable *hash cell*,[4] denoted as μ^s:

$$\begin{array}{llcl} (\textit{hash-cell}) & \mu^s & ::= & \texttt{Ref}(\textit{hashcode},\ \mu,\ \textit{auxinfo}) \\ (\textit{constructor}) & \mu & ::= & \#n \mid \texttt{Int} \mid \mu_1^s \rightarrow \mu_2^s \\ & & \mid & \lambda\kappa^s.\mu^s \mid \mu_1^s[\mu_2^s] \\ & & \mid & \texttt{Env}(\mu^s, \rho^s) \mid \texttt{Ind}(\mu_1^s, \mu_2) \end{array}$$

Here, a hash cell is a mutable record containing the following three fields: an integer hash code (*hashcode*), a term (μ), and a set of auxiliary information (*auxinfo*). The *auxinfo* maintains two attributes: a flag that shows whether μ is already in normal form[5] and if so, the set of free type variables in μ (in de Bruijn indices,[6] of course). Building a new constructor under this representation takes two steps: (1) calculate the hash code, and (2) if the constructor is not already in the hash table, calculate the *auxinfo* and insert the new cell.

The most interesting aspect of our representation scheme is that we can also memoize the result of every sequence of type reductions (e.g., those in Figure 4). Given a constructor $\mu^s = \texttt{Ref}(\textit{hashcode},\ \mu,\ \textit{auxinfo})$, suppose μ can be reduced to μ_1; then, we can do an in-place update, changing the second field of μ^s to a memoization node $\texttt{Ind}(\mu, \mu_1^s)$:



We keep the original μ in the new memoization node so that all future creations of μ (which will always have the same hash code) will be directly hash-consed to this new memoization node. The hashing procedure might require checking syntactic equality against μ because of potential hashing conflicts.

Note that the update is always safe, because it is only done to constructors that are not in normal form, so we do not have to recalculate the free variables, etc.

Memoization of reduction results has very interesting consequences: if we do not garbage-collect any of these memoization nodes (we may since they are weak pointers), then any redex of form μ can reuse the memoized result, μ_1^s. This leads to a very practical implementation that approximates optimal lambda reductions [18], with the caveat of using hash-consing, of course.

The combination of these techniques has proven to be very effective. With hash-consing and memoization, common operations such as equality tests, testing if a type is in 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 measurements have shown that these techniques reduce the compile time of large applications by an average of 45% (see Section 7.2).

[4]Hash-consed substitutions (ρ^s) and kinds (κ^s) are represented in the same way. Actually, because substitutions are simply finite mapping from de Bruijn indices to constructors, they share the same hash table with type constructors (we could simply encode them as a record constructor).

[5]By *normal form*, we mean those constructors that do not contain any redexes, i.e., no sub-term matches the left-hand side of the reduction rules in Figure 4.

[6]If we use named variables to represent the type abstraction in the term language (see Section 6), we would need to maintain two separate lists of free variables, one using de-Bruijn indices, another using named variables.

6 Manipulating Types

Although we use hash-consing, memoization, and the suspension-based lambda encoding to support efficient type handling, none of these implementation details are exposed to the clients of the type interface (see Figure 3). In fact, manipulating types under our type interface is still much like manipulating simple datatype-based representations. The only thing we have lost is the pattern matching capability.

Our interface also treats each type as its intension, that is, clients never need to think whether or not a type should be represented in normal form (or weak-head normal form). All operations in the type interface can apply to types of any form. Type reductions are completely hidden inside the underlying implementation and they are always done lazily.

Because of the various memoizations we do, our type interface also provides unusually fast implementations of several common operations. For example, we can check if a type is in normal form in constant time; we can also find the set of free variables in a type in constant time as well.

The only remaining issue is on how to represent type variables bound by polymorphic functions in the term language (i.e., $\Lambda t :: \kappa.e$). For a long time (including our most recent release), we have used the same de Bruijn indices to represent these type variables. This strategy requires no changes to the existing interface, but it has the unfortunate effect that the representation of a type annotation with free variables is now dependent on its lexical depth (the number of type abstractions under which it appears). The implication is that the client must adjust the representation when moving types from one depth to another.

Although we provide several utility functions to support this operation, having de Bruijn indices exposed does complicate certain optimization phases. Inlining, for example, requires adjusting types if the definition and call site are at different lexical depths. Specialization requires particularly drastic (yet subtle) adjustments to the types, since type abstractions themselves are being inlined and even eliminated.

We have experimented with an alternate design which hides the de Bruijn indices by supporting two different representations. Inside the type language, the type function (λ) and the polymorphic quantifier ($\forall$) still bind de Bruijn-indexed type variables. In the term language, however, type abstraction (Λ) binds *named* variables. This way, type annotations can be moved freely across depths because all free type variables are guaranteed to be named.

Naturally, this simplicity has a price. First, in order to reconstruct the type of a Λ term, we must traverse the types, converting the named variables into de Bruijn indices before placing the quantifier in front. Second, to memoize the set of free variables in each type representation, we now need to maintain two separate lists of type variables, one using de-Bruijn indices, another using named variables. Third, α-equivalent named types will not share the same representation. Our intuition, however, is that the additional cost for Λ-bound type variables will be acceptable, because these represent a very small portion of the total type size. In Sec-

Benchmark	Source Lines	Program Description	Code Size (bytes)	Tree Size (nodes)	Dag Size (nodes)
simple	918	A spherical fluid-dynamics program	114,944	34,118	1,913
vliw	3,682	A VLIW instruction scheduler	273,836	646,215	5,682
sml-nj	89,432	SML/NJ compiler v109.32	6,779,308	20,749,395	125,044
CM	7,703	SML/NJ Compilation Manager by Blume	487,048	3,186,279	27,968
cml	5,966	Concurrent ML by Reppy	366,684	1,203,391	17,106
eXene	35,662	An X-window system by Reppy & Gansner	2,291,628	99,567,031	78,671
ml-lex	1,232	A lexical-analyzer generator	103,604	112,091	3,122
toy	7	Identity function applied 18 times	22,148	30,409,149	14
toyp	8	Similar, with curried application	30,016	183,463,919	743

Figure 6: Description of benchmarks used. The tree size expresses the number of nodes in the type forest, if types were represented as trees (with no sharing of any kind). The dag size is the number of nodes actually created to represent the types in the compiler. The comparison between tree size and dag size is only intended to demonstrate the amount of redundancy in the types.

tion 7.3, we give preliminary measurements indicating that the additional costs are indeed acceptable. We conclude that this simpler design (using named variables) is quite feasible and will most likely be used in future versions of the compiler.

7 Experimental Results

This section gives empirical evidence demonstrating the effectiveness of the techniques presented in Sections 5 and 6. All techniques have been implemented in the FLINT/ML compiler [35] and in the SML/NJ production compiler since version 109.24 (January 9, 1997). All tests were performed on a Pentium Pro 200 Linux workstation with 64M physical RAM.

Figure 6 shows the set of benchmarks we used along with a summary of their salient features, including the size of the types. The dag size is the number of nodes when maximal sharing is realized, meaning that even α-equivalent types share the same representation. The ratio of tree size to dag size is intended to demonstrate the amount of redundancy in the types; it is not meant as a comparison of our representation to the completely naive one.

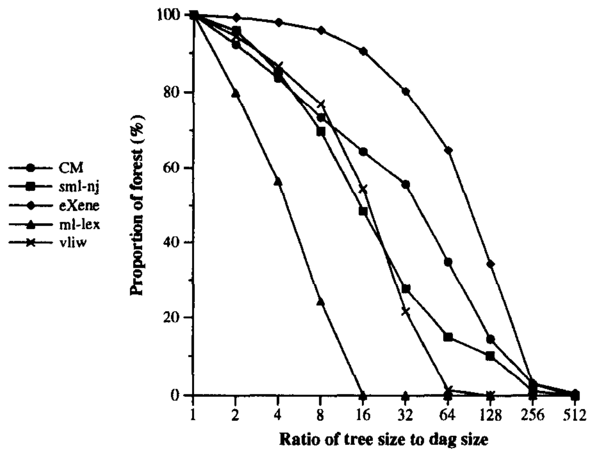


Figure 7: Amount of redundancy in types. The x-axis represents the minimum reduction factor realizable on some proportion of the type forest. For instance, with ml-lex, 80% of the type forest can be cut at least in half, and 25% can be reduced by a factor of 8.

7.1 Hash-consing results

This redundancy is examined for several benchmarks in Figure 7. Here, the y-axis represents some proportion of the type forest, while the x-axis shows the minimum reduction factor realized on that proportion of the forest thanks to hash-consing with de Bruijn indices.

The results for VLIW are particularly interesting. VLIW is written in an algorithmic style, making little use of higher-order functions, functors, or polymorphism. Nevertheless, we still get considerable reduction of the types used in the intermediate representation. This shows that hash-consing is not only beneficial for heavily functorized applications such as eXene and CM.

In order to get an idea of the cost of hash-consing, we measured the performance of our hash table. The table is an array containing 2,048 lists; collisions are handled by prepending new entries onto the list. Subscripting the array is very fast, so we need only be concerned with the cost of traversing the lists. Figure 8 shows the dynamic distribution of the lengths of list traversals. Most queries are satisfied after looking at only one or two list entries. One of the reasons is locality; we place new entries at the head of the list, so subsequent accesses are immediate. Furthermore, the table never gets very big; the maximum length of a list is 12. It seems clear that we should not be concerned about the performance of the hash table.

7.2 Memoization results

Figure 9 summarizes the results of doing various combinations of memoizations. The y-axis represents the compilation time of each benchmark, relative to the CPU time without any memoizations (the absolute time is printed above each set of bars). The memoizations performed are a normal-form indicator (NF), the set of free variables (FV), intermediate reduction results (RD), and combinations of these.

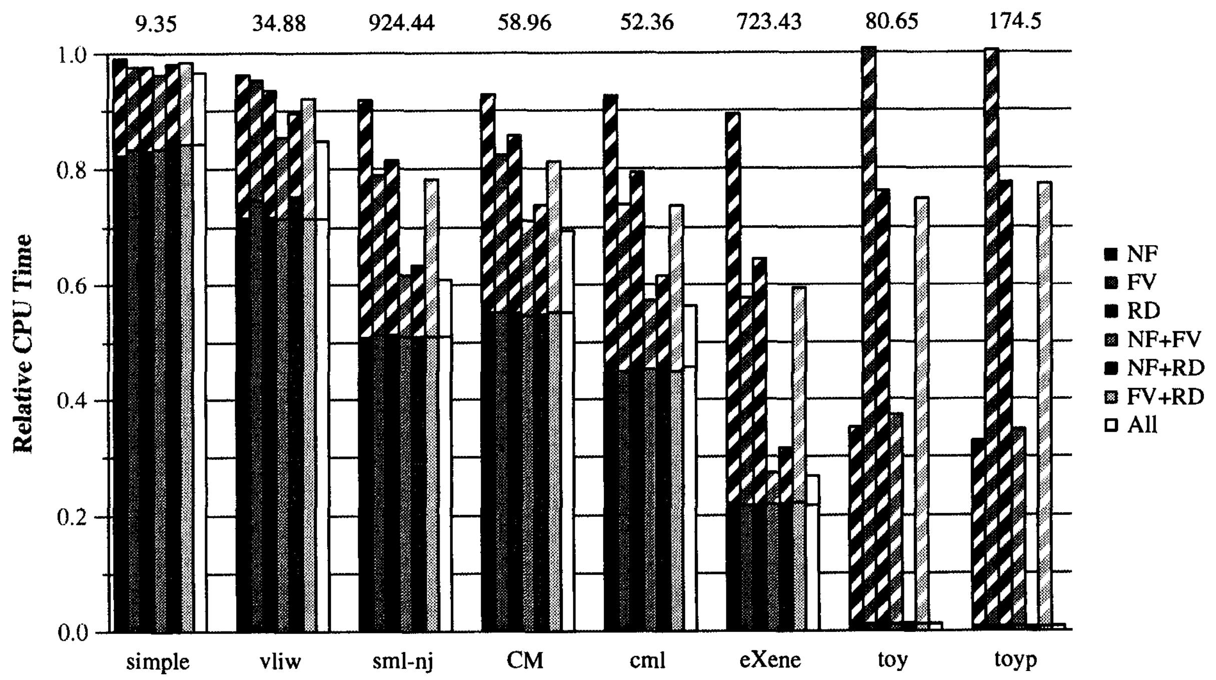


Figure 9: Memoization results. This shows the compilation times for each benchmark using various combinations of memoizations, relative to the time without memoizations. The striped part of each bar represents the type-preserving phases of the compilation. The results for ml-lex are very similar to those for vliw; they were omitted due to space constraints.

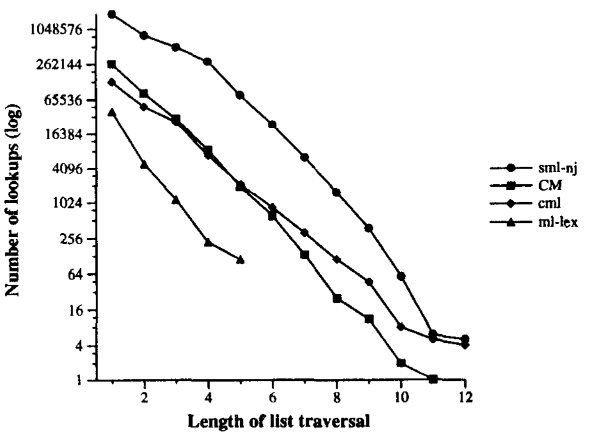


Figure 8: Hash table performance. This shows the dynamic number of hash table lookups (y-axis) that must search a bucket of particular length (x-axis). Most queries are satisfied after looking at only one or two list entries.

The striped part of each bar represents the type-preserving phases of the compilation, where our memoizations should have the most effect. Variation in the rest of the compilation time (represented by the solid bars) can be attributed to measurement error and secondary effects. Notice that, without memoizations, the type-preserving phases represent a significant portion of the compilation time, even on Simple and VLIW (18 and 27%, respectively).

The eXene benchmark is a large, heavily-functorized application on which our techniques are particularly effective. They reduce the total compilation time of eXene by 72% (a reduction of 93% in the type-preserving phases). Without memoizations, the type-preserving phases represent a dominant 78% of the compilation time; with them, these phases are a manageable 25%. Taking the average over the large benchmarks (sml-nj, CM, cml, and eXene), our techniques reduce the total compilation time by 45%.

In most benchmarks, memoizing NF+FV+RD does not seem to win much over just NF+FV. Also, in most cases, FV+RD achieves results somewhat similar to just FV. One might be tempted to assume that the other memoizations effectively subsume memoizing reduction results. However, there are extreme cases (toy and toyp, for example) where RD does improve compilation time when combined with other memoizations. These programs contain huge polymorphic types that are later specialized because they are only applied to integers. Without memoization of reduction results, specialization blows up.

7.3 Named variable results

Finally, we give preliminary measurements of the cost of using named type variables in the term language. As discussed in Section 6, the phases of the current compiler that are most inconvenienced by de Bruijn indices are inlining and specialization.

We added support in our type interface for named variables, and changed the FLINT representation to use them behind type abstractions. Next, we modified all compiler phases through specialization to use the named variables. *The modifications were fairly painless; deleting the most*

Benchmark	Compilation Time (seconds)		Ratio
	deBruijn	namedvar	
simple	8.53	8.44	0.99
vliw	28.78	28.37	0.99
sml-nj	566.96	565.91	1.00
CM	57.83	63.63	1.10
cml	105.17	108.54	1.03
eXene	188.46	188.55	1.00
ml-lex	8.19	8.33	1.02
toy	0.10	0.11	1.10
toyp	0.21	0.21	1.00

Figure 10: Named variable results. This shows compilation times for each benchmark using de Bruijn indices throughout, and then using named variables in the term language (and converting to de Bruijn indices after specialization). The last column gives the ratio of the second version over the first.

subtle parts of the specialization code was downright enjoyable. We have not yet modified the later phases. Instead, we temporarily inserted a phase after specialization to convert all remaining named type variables into de Bruijn indices. The cost incurred by this extra phase is included in the measurements given in Figure 10.

The compilation times of most benchmarks are not noticeably affected by the change. CM, the primary exception, suffered a 10% increase in compilation time due to the use of named variables. These results are preliminary because we have yet to modify the later phases of the compiler to use the new mixed representation (which would obviate the need for the extra conversion phase). We suspect that the remaining modifications will have no serious impact on performance, and with additional profiling and tuning, we may even be able to reduce the current overhead. We conclude that the simplified interface (made possible by using de Bruijn-indexed type variables internally and named variables externally) is quite feasible and will most likely be used in future versions of the compiler.

8 Comparison

Returning to our implementation criteria, we can certainly say that our scheme is very effective at representing types in a concise manner and provides us with a fast type equality test. Type manipulations are also made efficient by systematic use of memoization. Our experience with the interface is very positive since all the machinery is well hidden within a few core modules which export simple and intuitive type operations. There are nonetheless a few weaknesses:

- The interface hides the actual implementation behind functions which prevent the use of the pattern matching facilities of ML. This could be circumvented if really necessary, but it turned out to be a non-issue. Furthermore, the functional interface gives us a lot of flexibility.
- The de Bruijn indices make some manipulations more subtle than we would like. By using a mix of named variables and de Bruijn indices, we are able to simplify such manipulations outside the core modules and still achieve acceptable performance.
- In order to make sure type traversals are efficient, we have to use a *fold* function on types which encapsulates the memoization. Here also, our experience has shown that it is not a serious issue.

Our choice of techniques to provide efficient type manipulation should be contrasted with the **lettype** scheme used in the TIL compiler [40]. It should be noted here that very little has been published about the **lettype** scheme, so this comparison is based on our own understanding of what **lettype** could look like under the ideal scenario rather than any existing implementation such as the one in the TIL compiler.

The basic approach is to extend the notion of A-normal form to types by providing **lettype** (in both the term and the type languages). For example, the identity function on integer pairs would look like:

$$\begin{array}{l}\texttt{lettype}\ t_1 = \texttt{Int} * \texttt{Int} \\ \texttt{in lettype}\ t_2 = t_1 \to t_1 \\ \quad \texttt{in}\ (\lambda x : t_1.x) : t_2\end{array}$$

This has the advantage of making the sharing explicit since all types are referenced through names bound in the type environment. The explicit sharing basically eliminates the risk of accidentally traversing the type tree in an inefficient way. In other words, type traversals get memoized for free. But **lettype** suffers from many problems:

- There is no known way to define a compact normal form for such type representations. This implies that type equality tests become much more expensive. All existing theoretical framework treats **lettype** $t = \mu_1$ **in** μ_2 as if it is a β-reduction of form $(\lambda t :: \kappa.\mu_2)[\mu_1]$. This would clearly expand into a normal form, but on the other hand, this reduction is precisely one that is banned by the **lettype** scheme, as otherwise, type expressions would degenerate into inefficient tree representations.
- Similarly it is unclear how one could provide a clean interface that allows its clients to be oblivious to normalization issues while still ensuring efficient execution, since memoizing the normalization steps would require adding types to the environment which in turn would force the rewrite of the whole term.
- Expressing sharing is not enough: we still first need to find that sharing. We might be able to get some sharing information straight from the type-inference phase, but this will require careful coding. Also we might not get as much sharing as we would want. TIL's solution is to go through a common sub-expression elimination phase. This would indeed allow us to merge all the common types, but requires precisely the same machinery as hash-consing and is done after the fact, whereas we are careful to eliminate common sub-expressions as soon as they appear. Furthermore, many more common sub-expressions will appear during the compilation process which will require additional passes through the CSE phase while our scheme takes advantage of the hash-consing all along the compilation process to guarantee that sharing is constantly maintained.

- Another subtle difference is that `lettype` traverses its types most naturally in a bottom-up fashion which precludes (or rather reduces the effectiveness of) optimizations that cut-off the traversal of types. More specifically, `lettype` would not let us make as good a use of information such as free-variables or a normal-form bit.

To summarize, `lettype` seems to provide a clean way to represent types efficiently, but it ends up having to pay the cost of hash-consing anyway without reaping all the benefits of our more straightforward scheme. Also it is yet to be seen how `lettype` scales to real world situations such as eXene, which our scheme handles easily.

Our approach manages to hide most if not all the complexity of type manipulation, providing programmers with a simple and intuitive interface. It ensures that maintenance of type information is non-intrusive, which is greatly appreciated for optimization phases that do not rely on type information. `lettype` on the other hand would most likely force every phase to maintain at the very least a type environment.

Finally, our implementation is straightforward since it relies on well understood techniques and it does not suffer from hidden costs since all the hash-consing and memoizing is done once and for all.

9 Conclusions

Implementing typed intermediate languages is not a trivial task. In this paper, we have presented a series of novel techniques that make type-preserving compilers practical and scalable. We argue that a type-preserving compiler will not scale to handle large types unless all of its type-preserving stages preserve the asymptotic time and space usage in representing and manipulating types. We believe what we learned from our implementation will be valuable to future implementations of other emerging typed intermediate languages.

Availability

The implementation discussed in this paper is now released with the Standard ML of New Jersey (SML/NJ) compiler and the FLINT/ML compiler [35]. SML/NJ is a joint work by Lucent, Princeton, Yale and AT&T. FLINT is a modern compiler infrastructure developed at Yale University. Both FLINT and SML/NJ are available from the following web site:

`http://flint.cs.yale.edu`

Acknowledgement

We would like to thank Valery Trifonov, Bratin Saha, and the anonymous referees for their comments and suggestions on an early version of this paper. Valery Trifonov helped implement the FLINT type-checker and also participated in numerous discussions on topics covered by this paper.

References

[1] M. Abadi, L. Cardelli, P. Curien, and J. Levy. Explicit substitutions. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 31–46, New York, Jan 1990. ACM Press.

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

[3] 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.

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

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Ninth Annual ACM Symp. on Principles of Prog. Languages*, pages 207–212, New York, Jan 1982. ACM Press.

[6] O. Danvy. Type-directed partial evaluation. In *Proc. 23rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 242–257. ACM Press, 1996.

[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] A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 11–24. ACM Press, June 1997.

[9] J. Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 1–15, New York, Jan 1990. ACM Press.

[10] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. ACM SIGPLAN '93 Conf. on Prog. Lang. Design and Implementation*, pages 237–247, New York, June 1993. ACM Press.

[11] 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.

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

[13] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[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] G. Huet. *Logical Foundations of Functional Programming.* Addison-Wesley, 1990.

[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 and F. Rouaix. Security properties of typed applets. In *Twenty-fifth Annual ACM Symp. on Principles of Prog. Languages*, page (to appear), New York, Jan 1998. ACM Press.

[18] J.-J. Levy. Optimal reductions in the lambda calculus. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Edited by J. P. Seldin and J. R. Hindley, Academic Press, 1980.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[20] H. G. Mairson. Deciding ML typability is complete for determinsitic exponential time. In *Proc. 17th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 382–401. ACM Press, 1990.

[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. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. 25rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, page (to appear). ACM Press, 1998.

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

[25] 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.

[26] G. Necula. Proof-carrying code. In *Twenty-Fourth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1997. ACM Press.

[27] S. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[28] S. Peyton Jones and J. 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.

[29] S. Peyton Jones, J. Launchbury, M. Shields, and A. Tolmach. Bridging the gulf: a common intermediate language for ML and Haskell. In *Proc. 25rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, page (to appear). ACM Press, 1998.

[30] S. Peyton Jones and E. Meijer. Henk: a typed intermediate language. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[31] J. H. Reppy and E. R. Gansner. The eXene library manual. Cornell Univ. Dept. of Computer Science, March 1991.

[32] 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.

[33] B. Saha and Z. Shao. Optimal type lifting. In *Proc. 1998 International Workshop on Types in Compilation*, March 1998.

[34] Z. Shao. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 85–98. ACM Press, June 1997.

[35] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[36] Z. Shao. Typed common intermediate format. In *Proc. 1997 USENIX Conference on Domain Specific Languages*, pages 89–102, October 1997.

[37] Z. Shao. Typed cross-module compilation. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*. ACM Press, September 1998.

[38] 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.

[39] 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.

[40] D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1996. Tech Report CMU-CS-97-108.

[41] 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.

[42] M. Tofte. Region-based memory management (invited talk). In *Proc. 1998 International Workshop on Types in Compilation*, March 1998.

[43] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 1–11, New York, June 1994. ACM Press.