The following paper was originally published in the
Proceedings of the Conference on Domain-Specific Languages
Santa Barbara, California, October 1997

Typed Common Intermediate Format

Zhong Shao
Yale University

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Typed Common Intermediate Format

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

Abstract

Application languages are very effective in solving specific software problems. Unfortunately, they pose great challenges to reliable and efficient implementations. In fact, most existing application languages are slow, interpreted, and have poor interoperability with general-purpose languages.

This paper presents a framework on building high-quality systems environment for multiple advanced languages. Our key innovation is the use of a common typed intermediate language, named FLINT, to model the semantics and interactions of various language-specific features. FLINT is based on a predicative variant of the Girard-Reynolds polymorphic calculus F_ω, extended with a very rich set of primitive types and functions.

FLINT provides a common compiler infrastructure that can be quickly adapted to generate compilers for new general-purpose and domain-specific languages. With its single unified type system, FLINT serves as a great platform for reasoning about cross-language interoperations. FLINT types act as a glue to connect language features that complicate interoperability, such as mixed data representations, multiple function calling conventions, and different memory management protocols. In addition, because all runtime representations are determined by FLINT types, languages compiled under FLINT can share the same system-wide garbage collector and foreign function call interface.

1 Introduction

Application languages (a.k.a. domain-specific languages or DSLs) are very effective in solving specific software problems. Unfortunately, their focus on a particular domain and their (often) quick turn-around time make it unrealistic to develop full-scale compilers from scratch. In fact, due to the lack of compiler infrastructures, many existing application languages are interpreted but not compiled. As a result, software written in application languages are generally slow and have poor interoperability with general-purpose languages.

The interoperability problem, of course, also applies to advanced type safe languages such as Java [10], Modular-3 [27], ML [22], and Haskell [17]. Each of these programming languages, whether general-purpose or domain specific, often has its own syntax, semantics, and implementation specifics; it also must run under a special runtime system with its own garbage collector and foreign function call interface (to the low-level C code). Interoperation or communication among these languages is a nightmare, if not impossible. Several recently proposed object models (e.g, Microsoft's COM [32] and OMG's CORBA [11]) offer a partial solution, however, they all require that programs written in different languages run under different hardware-protection domains (i.e., address space). Wallach *et al* [41] have shown that cross-domain function calls under COM can be a factor of 1000 times slower than the regular function calls within a single domain. This is unacceptable for many applications.

The problem on (lack of) compiler infrastructures is even more serious. To write a compiler for a new language L, one has to either write everything from scratch, or compile L into some main-stream languages such as C and C++. However, for most advanced languages, C is much too low-level to serve as a good target language. Modern languages often support strong typing, automatic memory management, program exceptions, and higher-order functions (or closures), but C does not support any of these. Most C compilers are not designed to produce good code for these higher-level language features. To write a compiler from L to C, one still must write

multiple compilation phases and customize her own runtime system (including support to garbage collection, proper signal-handling, and foreign-function call interface).

This paper presents a new framework on building high-quality systems environment for *multiple* general-purpose and application-oriented languages. We are particularly interested in the class of HOT[1] languages, namely, languages that are *Higher-Order* and *Typed*. With a broader interpretation, we use Higher-Order to include languages where objects contain methods (even though functions are not first-class citizens), and Typed to include both static and dynamic typing. Thus, Java is HOT, so is ML, Haskell and Scheme. Because application languages are designed to exploit a higher-level of abstraction and program analysis, many of them are designed to be HOT as well.

Our key innovation is the use of a common typed intermediate language, named FLINT, to model the semantics and interactions of various language-specific features. FLINT is based on a predicative variant of the Girard-Reynolds polymorphic lambda calculus F_ω [9, 31], extended with a very rich set of primitive types and functions. Although HOT languages can be very different in semantics, they all have a mathematically rigorous type system. The fact that almost all HOT features can be compiled into an F_ω-like calculus is not surprising, because F_ω is frequently used as a meta-language for reasoning about formal logic and semantics.

FLINT provides a common compiler infrastructure that can be quickly adapted to generate compilers for new general-purpose or domain-specific languages. With its single unified type system, FLINT serves as a great platform for reasoning about cross-language interoperations. FLINT types act as a glue to connect language features that complicate interoperability, such as mixed data representations, multiple function calling conventions, and different memory management protocols. In addition, because all runtime representations are determined by FLINT types, languages compiled under FLINT can share the same system-wide garbage collector and foreign function call interface (to the low-level C code). Finally, because it has a more expressive type system, FLINT code can also serve as (or translated into) more powerful executable content than Java VM code, making all HOT programs internet ready.

[1] The acronym "HOT" is coined by Bob Harper, and is widely publicized by Phil Wadler in his recent editorial [40] for Journal of Functional Programming.

The FLINT system is being developed at Yale University, using the infrastructure in the type-based version of the SML/NJ compiler [37]. A preliminary implementation of the FLINT intermediate language has been incorporated into the working releases of the SML/NJ compiler since version 109.24 (January 9, 1997). The resulting compiler handles the entire SML'97 [22] plus MacQueen-Tofte higher-order modules [21]. It also gives better performance (about 20% speedup on benchmarks that involve recursive and mutable types) than the older versions of the SML/NJ compiler [37]. New front ends for other languages (e.g., Safe C, Haskell, Java) are under active development.

In the rest of this paper, we first give an introduction to the basic architecture of the FLINT system. We then present the current design of our typed common intermediate language, followed by a summary of the main implementation techniques we used to compile this intermediate language. We further show how different general-purpose or application-oriented languages might be translated into FLINT, and finally, we discuss the related work, and then conclude.

2 The FLINT Architecture

The FLINT system, as shown in Figure 1, is organized around a strongly typed intermediate language also named FLINT. Programs written in various source languages 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 intermediate format. The *middle end* does conventional dataflow and loop optimizations [1, 39], local and cross-module type specializations, and λ-calculus-based contractions and reductions [3]; it then produces an optimized version of the FLINT code. The *back end* compiles FLINT into machine code through the usual phases such as representation analysis [34] (to compile polymorphism), safe-for-space closure conversion [36] (to compile higher-order functions), register allocation, instruction scheduling, and machine-code generation [8]. All the compilation stages are deliberately made independent of each other so that they may be pieced together in different ways for different languages.

The runtime system provides support to system-wide garbage collection, foreign-function call interface, and connections to lower-level operating sys-

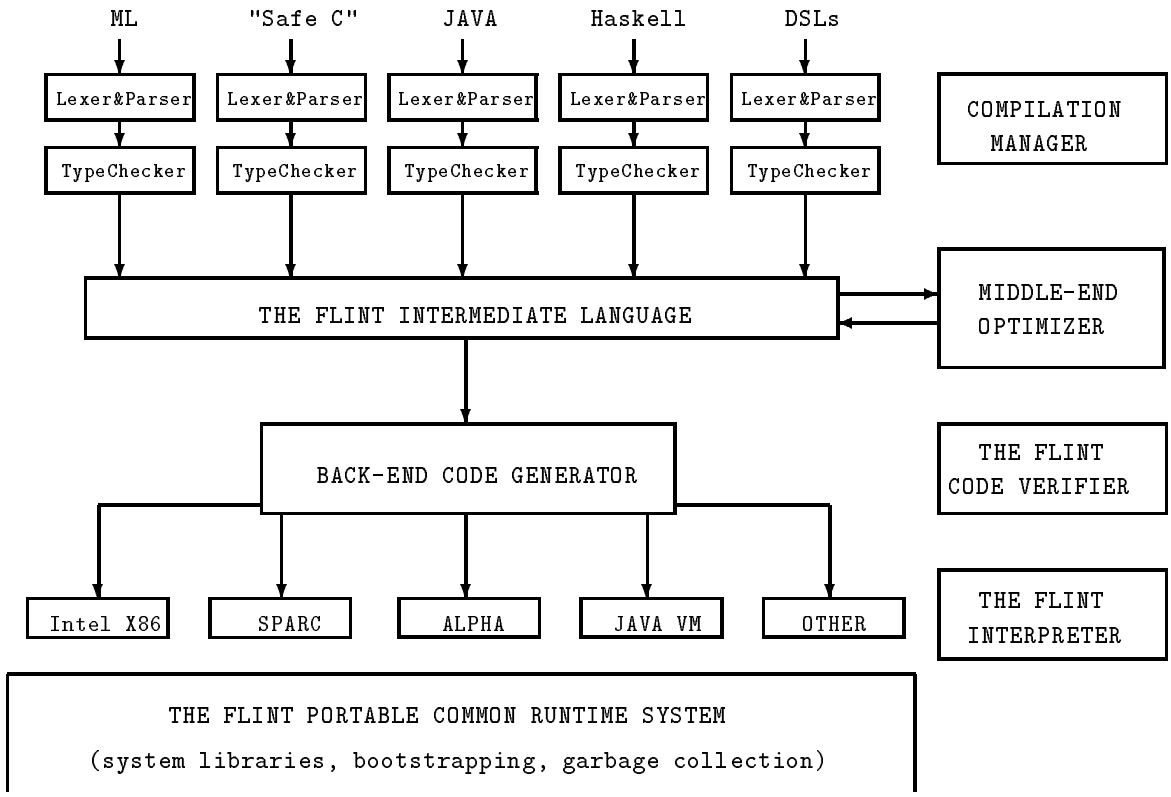

Figure 1: Top-Level Structure of the FLINT System

tem services. Our current implementation borrows SML/NJ's runtime system [30, 2, 18] which runs under all major machine platforms. We plan to extend it to support new services such as dynamic linking and bytecode execution.

It is important to emphasize the modular nature of the overall compiler structure. The FLINT intermediate language nicely separates the language-dependent front end from the language-independent back end. Compiler optimizations done at the middle end are always performed as FLINT-to-FLINT program transformations (so FLINT must be designed as suitable for optimizations). This orgnization also allows FLINT to be used as an advanced executable content language, just like the Java VM bytecode [20]. Here, the front end and the middle end can be thought as a source-to-FLINT compiler; the back end and the runtime system are some kind of *just-in-time* compiler and virtual machine. Because FLINT is designed as a compiler intermediate language, compiling into FLINT does not incur any efficiency loss as the stack-based Java bytecode.

Another important aspect is on the organization of the back end code generator. To keep FLINT's type system simple, we currently let the back end handle the compilation of polymorphism (i.e., representation analysis [19, 34]) and higher-order functions (i.e,. closure conversion [36]). However, there is no reason why we can't propagate and preserve type information throughout the back end. In fact, we intend to propagate the type information into the assembly or machine code to guide sophisticated instruction scheduling and to generate Necula-Lee style proof-carrying code [26].

To construct a compiler for a new application language, we only need to write a new front end that translates the source program into the FLINT intermediate code. If the language is embedded inside another general-purpose language, we simply modify the front end for the host language to support the domain-specific aspects. Most of the time, new primitive operators and type constructors must be added into the intermediate language to support the corresponding surface language constructs; the middle-end optimizer must also be tailored to support the corresponding domain-specific program

analysis and transformations. We believe that majority of the domain specific features can be abstracted into a set of algebraic data types, where each consists of a set of primitive types, primitive operators, and proper rewriting rules (i.e., axioms). So even the process of modifying the intermediate language and the middle-end optimizer can be automated.

3 Typed Intermediate Format

Using common intermediate languages to share compiler infrastructure is not a new idea. Many existing compilers, such as GNU GCC, Stanford's SUIF [12], and U. Washington's Vortex [7], all use some kind of shared intermediate format for multiple source languages. In addition, the C programming language has been used as the *de facto* standard target language for a long time. Since all these are mainly designed for conventional imperative languages, none of them directly support higher-order functions or advanced polymorphic type system.

FLINT is designed as a *strongly typed* common intermediate format for HOT languages. There are many advantages in making the intermediate language type-safe. First, a rigorous type system can be used to reason about the safety of a program, even at the intermediate language level. This is particularly important for applications that must be as secure and mobile as the Java VM code. Second, type information makes it possible to reason about principled interoperability among different languages. In fact, because all data representations and function calling conventions are decided based on a uniform type system, it is possible to make programs of different surface languages share the same runtime system (with the same garbage collector and foreign function call interface). Finally, type information have proven invaluable for efficient compilation of statically typed languages such as ML and Haskell [19, 28, 37, 38]; types are also useful for debugging compilers and proving properties of programs.

3.1 Rationale

The current FLINT language is designed based on the following principles:

- *Strong and explicit typing.* ML-like languages often have very tricky type inference problems. Having an explicitly typed intermediate language leaves the type inference issues completely to the front end.

- *Simple and well-defined semantics.* The intermediate language must be simple, clean, and semantically well-founded in order to be used as a common target language.

- *Expressiveness.* In order to support multiple HOT languages, the FLINT type system must be rich enough to express HOT features such as higher-order functions, ML-like polymorphism, and higher-order modules.

- *Pay-as-you-go efficiency.* The intermediate language must, of course, be compiled to generate efficient code. Furthermore, simple, first-order, monomorphic functions should be compiled as efficiently as in C or assembly languages, even though the presence of polymorphic functions might complicate data representations and function calling conventions.

- *Optimization ready.* The compiler middle end performs various kinds of optimizations on the intermediate code. For this reason, the intermediate representation must be compatible with all standard program analysis and transformations [3, 1]. The intermediate language should also contain explicit loop (and recursion) construct to support sophisticated loop optimizations.

- *System-programming friendly.* The intermediate language must provide excellent support to low-level system programming such as safe type-cast, dynamic types, and bit-manipulation primitives. It should also contain a subset of language features that can be used to write real-time programs (e.g., code fragments that do not involve garbage collections).

- *Extensible.* The intermediate language must be easily extended to support other advanced or domain-specific language features (e.g., concurrency, objects, and user-defined datatypes).

To summarize, what we want is a intermediate language that behaves like a strongly typed *assembly* language. It should be high-level enough to express polymorphism and higher-order functions but low-level enough to support all standard optimizations.

3.2 Background

The core language of FLINT is a predicative variant of the Girard-Reynolds polymorphic λ-calculus F_ω [9, 31], with the term language written in the A-normal form [33]. In the following, we first give a introduction about F_ω, and then formally define the Core-FLINT language.

The standard Girard-Reynolds polymorphic calculus F_ω is often defined as follows:

$$\begin{array}{llll} (\mathit{kinds}) & \kappa & ::= & \Omega \mid \kappa_1 \rightarrow \kappa_2 \\ (\mathit{types}) & \sigma & ::= & t \mid \sigma_1 \rightarrow \sigma_2 \mid \forall t :: \kappa.\sigma \\ & & \mid & \lambda t :: \kappa.\sigma \mid \sigma_1[\sigma_2] \\ (\mathit{terms}) & e & ::= & x \mid \lambda x : \sigma.e \mid @e_1 e_2 \\ & & \mid & \Lambda t :: \kappa.e \mid e[\sigma] \end{array}$$

The calculus contains three syntactic classes: kinds (κ), types (σ), and terms (e). Here, kinds classify types, and types classify terms. The extra "kind" hierarchy is used to regulate and define well-formed types. In F_ω, both simple types (e.g., functions, records, integers) and polymorphic types (e.g., $\forall t :: \kappa.\sigma$) have kind Ω; higher-order types (or really, type functions) such as $\lambda t :: \kappa.\sigma$ has kind $\kappa \rightarrow \kappa'$, if σ belong to kind κ'. A higher-order type σ_1 can be applied to another type σ_2, written as $\sigma_1[\sigma_2]$.

At the term level, in addition to the usual lambda abstraction and application, F_ω also support explicit type abstraction and type application (written as $\Lambda t :: \kappa.e$ and $e[\sigma]$). Every type abstraction term such as $\Lambda t :: \kappa.e$ has the polymorphic type $\forall t :: \kappa.\sigma$, assuming term e has type σ.

For example, an F_ω function $f = \Lambda t :: \Omega.\lambda x : t.x$ would have type $\sigma_0 = \forall t :: \Omega.t \rightarrow t$. In the standard F_ω, the polymorphic type such as σ_0 is still considered to have kind Ω, so expressions such as "$@(f[\sigma_0])f$" would type check, and yield type σ_0.

Because F_ω supports a very general kind of higher-order polymorphism, it is commonly used as the meta-language to reason about formal logic and semantics. In fact, many advanced languages such as ML and Haskell can be embedded into the F_ω-like calculus.

3.3 The Core Language

The core language of FLINT is based on the standard F_ω, but with the following three important changes:

- In standard F_ω, polymorphic types are treated same as monomorphic types, and they both have kind Ω. This complicates the semantics and makes the calculus *impredicative*. Following Harper and Morrisett [15], we split the type hierarchy into two levels: a *constructor* level characterizes the monomorphic types (and type functions), and a *type* level expresses the polymorphic types. "Kind" is now used to classify "constructors" only; polymorphic types such as the previous σ_0 no longer belongs to kind Ω. So expressions such as "$@(f[\sigma_0])f$" will no longer type check in our predicative variant.
- The call-by-value term language is split into two levels as well, with values denoting simple variables or constants. The usual term expressions must now satisfy new syntactic restrictions as standard A-normal forms [33]. More specifically, each function application (or type application) can only refer to values (as $@v_1 v_2$). The standard F_ω function application term $@e_1 e_2$ is rewritten (according to call-by-value semantics) into a nested `let` expressions followed by the actual value application.
- A new product kind $\kappa_1 \otimes \kappa_2$ is added into the kind language to express a sequence of type constructors. The product kind makes it possible to define type functions that takes a sequence of type constructors as argument and returns another sequence as the result. This is useful to express the parameterized modules such as ML higher-order functors [21].

The Core FLINT contains the following five syntactic classes: kinds (κ), constructors (μ), types (σ), terms (e), and values (v):

$$\begin{array}{llll} (\mathit{kinds}) & \kappa & ::= & \Omega \mid \kappa_1 \rightarrow \kappa_2 \mid \kappa_1 \otimes \kappa_2 \\ (\mathit{con's}) & \mu & ::= & t \mid \texttt{Int} \mid \rightarrow(\mu_1, \mu_2) \\ & & \mid & \lambda t :: \kappa.\mu \mid \mu_1[\mu_2] \\ & & \mid & \otimes(\mu_1, \mu_2) \mid \Pi_1 \mu \mid \Pi_2 \mu \\ (\mathit{types}) & \sigma & ::= & T(\mu) \mid \forall t :: \kappa.\sigma \mid \sigma_1 \rightarrow \sigma_2 \\ (\mathit{terms}) & e & ::= & v \mid \lambda x : \sigma.e \mid @v_1 v_2 \\ & & \mid & \Lambda t :: \kappa.e \mid v[\mu] \\ & & \mid & \texttt{let } x = e_1 \texttt{ in } e_2 \\ (\mathit{values}) & v & ::= & x \mid i \end{array}$$

Here, kinds classify constructors, and types classify terms and values. Constructors of kind Ω now only name monotypes. The monotypes are generated from variables, `Int`, through the constructors $\rightarrow$. As in F_ω, 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 product kind $\kappa_1 \otimes \kappa_2$. 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 (which must be of kind Ω). As in F_ω, the terms are an explicitly typed λ-calculus (but in A-normal form) with explicit constructor abstraction and application forms. We intentionally included the primitive constructor `Int` and the primitive constant i to show how the core calculus might be extended into a more complete languages.

The static semantics of Core-FLINT, given in Figure 2, consists of a collection of rules for constructor formation, constructor equivalence, type formation, type equivalence, and term formation. The term formation rules are in the form of $\Delta; , \vdash e : \sigma$ where Δ is a kind environment mapping type variables to kinds, and , is the type environment mapping term variables to types. Harper and Morrisett [15, 23] have shown that type checking for predicative F_ω-like calculus is decidable, and furthermore, its typing rules are consistent with the standard call-by-value operational semantics.

3.4 The Full Language

In order to make FLINT as simple as possible, we let the front end deal with many higher-level language constructs. For example, the front end for ML can translate higher-order modules into the Core-FLINT-like calculus [35, 14] in a type-preserving way, thus completely eliminating the need of module constructs from the intermediate language. Similarly, type classes in Haskell can also be embedded into F_ω through explicit dictionary passing.

The complete FLINT language still contains many more type and term constructs than the core languages. Because FLINT is an explicitly typed language, adding new type constructors into FLINT does not involve any type reconstruction problem. In the following, we summarize the main features in our current design:

- A `letrec` construct at the term level to allow the declaration of mutually recursive functions.
- A "sum" type constructor at the constructor level to represent ML-like concrete datatypes. Manipulating values of sum types are done through a set of injection functions plus a "switch"-based projection function.
- A recursive operator at the constructor level to allow definitions of recursive type constructors (e.g., `List`). At the term level, two primitive operators, `roll` and `unroll`, converts values of recursive types into those of the underlying sum types.
- A primitive exception type `Exn` at the constructor level and a pair of term-level constructs: "`raise` v" would raise the exception v, and "`try` e `handle` v" would run the expression e, if any exception is raised, the handler v is called.
- An `Abs` constructor at the constructor level and a pair of primitives `pack` and `unpack` at the term level, with the following kind and type signatures:

  $$\texttt{Abs} :: \Omega \rightarrow \Omega$$

  $$\texttt{pack} : \forall t :: \Omega.\, T(t) \rightarrow T(\texttt{Abs}(t))$$
  $$\texttt{unpack} : \forall t :: \Omega.\, T(\texttt{Abs}(t)) \rightarrow T(t)$$

  Every source-level abstract type t is represented in the form of $\texttt{Abs}[\mu]$ inside FLINT, where μ is the internal representation type (hidden from the programmer). The representation types are useful when pickling values of abstract types.

Almost all the rest FLINT constructs can be expressed using the same "signature" form as the above `Abs` primitives. Each signature defines a primitive type constructor at the constructor level and a set of primitive constants and operators at the term level. The primitive functions often satisfy a set of axioms that can be used to optimize the term-level expressions. Our current implementation hardwires the axioms into the middle-end optimizer, but we plan to automate this process in the future.

The FLINT language also includes primitives such as N-bit integers (trapping or non-trapping), N-bit words, N-bit characters (ascii or unicode), N-bit floating-point numbers, strings, boolean types, boxed reference cells, array, packed arrays, vectors, packed vectors, mono arrays and mono vectors, ML-like immutable records (nested or flat), first-class continuations, control continuations (used by CML [29]), suspensions (or thunks, to support lazy evaluations).

In the long term, we plan to add type dynamic and some form of F-bounded quantification to support object-oriented languages such as Java. Type dynamic would also make it possible to translate dynamically typed languages such as Scheme into

Constructor Formation and Constructor Equivalence:

$$(v/i/fn) \qquad \frac{}{\triangle \uplus \{t :: \kappa\} \triangleright t :: \kappa} \qquad \frac{}{\triangle \triangleright \mathtt{Int} :: \Omega} \qquad \frac{\triangle \triangleright \mu_1 :: \Omega \quad \triangle \triangleright \mu_2 :: \Omega}{\triangle \triangleright \rightarrow (\mu_1, \mu_2) :: \Omega}$$

$$(cfn/capp) \qquad \frac{\triangle \uplus \{t :: \kappa_1\} \triangleright \mu :: \kappa_2}{\triangle \triangleright (\Lambda t :: \kappa_1.\mu) :: \kappa_1 \rightarrow \kappa_2} \qquad \frac{\triangle \triangleright \mu_1 :: \kappa' \rightarrow \kappa \quad \triangle \triangleright \mu_2 :: \kappa'}{\triangle \triangleright \mu_1[\mu_2] :: \kappa}$$

$$(cprod) \qquad \frac{\triangle \triangleright \mu_1 :: \kappa_1 \quad \triangle \triangleright \mu_2 :: \kappa_2}{\triangle \triangleright \mu_1 \otimes \mu_2 :: \kappa_1 \rightarrow \kappa_2} \qquad \frac{\triangle \triangleright \mu :: \kappa_1 \otimes \kappa_2}{\triangle \triangleright \Pi_i \mu :: \kappa_i} \quad (i = 1, 2)$$

$$(cequiv) \qquad \frac{\triangle \uplus \{t :: \kappa'\} \triangleright \mu_1 :: \kappa \quad \triangle \triangleright \mu_2 :: \kappa'}{\triangle \triangleright (\Lambda t :: \kappa'.\mu_1)[\mu_2] \equiv [\mu_2/t]\mu_1 :: \kappa} \qquad \frac{\triangle \triangleright \mu_1 :: \kappa_1 \quad \triangle \triangleright \mu_2 :: \kappa_2}{\triangle \triangleright \Pi_i(\mu_1 \otimes \mu_2) \equiv \mu_i :: \kappa_i} \quad (i = 1, 2)$$

Type Formation and Type Equivalence:

$$(tform) \qquad \frac{\triangle \triangleright \mu :: \Omega}{\triangle \triangleright T(\mu)} \qquad \frac{\triangle \triangleright \sigma_1 \quad \triangle \triangleright \sigma_2}{\triangle \triangleright \sigma_1 \rightarrow \sigma_2} \qquad \frac{\triangle \uplus \{t :: \kappa\} \triangleright \sigma}{\triangle \triangleright \forall t :: \kappa.\sigma}$$

$$(tequiv) \qquad \frac{\triangle \triangleright \mu_1 :: \Omega \quad \triangle \triangleright \mu_2 :: \Omega}{\triangle \triangleright T(\rightarrow (\mu_1, \mu_2)) \equiv T(\mu_1) \rightarrow T(\mu_2)}$$

Term Formation:

$$(value) \qquad \frac{}{\triangle; , \vdash i : \mathtt{Int}} \qquad \frac{}{\triangle; , \vdash x : , (x)}$$

$$(fn/app) \qquad \frac{\triangle; , \uplus \{x : \sigma_1\} \vdash e : \sigma_2}{\triangle; , \vdash \lambda x : \sigma_1.e : \sigma_1 \rightarrow \sigma_2} \qquad \frac{\triangle; , \vdash v_1 : \sigma' \rightarrow \sigma \quad \triangle; , \vdash v_2 : \sigma'}{\triangle; , \vdash @v_1 v_2 : \sigma}$$

$$(let) \qquad \frac{\triangle; , \vdash e_1 : \sigma_1 \quad \triangle; , \uplus \{x : \sigma_1\} \vdash e_2 : \sigma_2}{\triangle; , \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \sigma_2}$$

$$(tfn/tapp) \qquad \frac{\triangle \uplus \{t : \kappa\}; , \vdash e : \sigma}{\triangle; , \vdash \Lambda t :: \kappa.e : \forall t :: \kappa.\sigma} \qquad \frac{\triangle \triangleright \mu :: \kappa \quad \triangle; , \vdash v : \forall t :: \kappa.\sigma}{\triangle; , \vdash v[\mu] : [\mu/t]\sigma}$$

Figure 2: The Static Semantics of Core-FLINT

```
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 3: Representing Kinds, Constructors, and Types

FLINT. We also intend to extend FLINT to cover more interesting representation types. Since most of these are just new primitive constructors and functions, the overall structure of the FLINT language remains to be simple and small.

3.5 Implementations

One 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 3 for a fragment of the FLINT definitions, written as ML datatype definitions). First, we represent all type variables as de Bruijn indices [6]. 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.

Finally, to make type reduction *lazy*, we use Nadathur's *suspension* notations [24, 25] 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 [25]. The environment e determines the bindings for the type variables.

Figure 3 gives parts of the definitions of FLINT kind (`tkind`), constructor (`tyc`), and type (`lty`) using SML datatype definitions. 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 [4], our optimized implementation is an order-of-magnitude faster (in compilation time) than naive implementations.

Representing type variables as de Bruijn indices does have its drawback. For example, the type-based manipulation becomes much harder to program. A simple beta-reduction such as $v[\mu]$ where $v = \Lambda t :: \kappa.e$ requires adjustment of all type variables occurred free in e; furthermore, if t occurs with some type abstractions, then μ must be adjusted as well.

4 Compiling FLINT

The FLINT code is compiled in two steps. First, the middle end performs a series of conventional control and data flow optimizations. All optimizations are type-preserving so the resulting FLINT code will still type-check under the same typing rules. Because FLINT terms are always in the A-normal form, all CPS-based optimizations [3] apply to FLINT as well. Apart from the presence of polymorphism and higher-order functions, the resulting FLINT code should be very close to the low-level machine languages.

After the optimizations, the back end uses flexible representation analysis [34] to compile polymorphism and safe-for-space closure conversion to compile higher-order functions [36]; it then does the standard register allocation, instruction scheduling, and machine code generation [8].

In the rest of this section, we sample several important techniques used in our compiler back end.

4.1 Type Specialization

Because polymorphic functions are often more expensive than monomorphic functions, the middle end of our compiler performs several rounds of *type specialization* to decrease the degree of polymorphism. The basic idea can be illustrated by the following example:

$$\begin{array}{l}\texttt{let } f = \Lambda t :: \Omega.\lambda x :: T(t).x \\ \quad \texttt{in let } g = \Lambda s :: \Omega.\lambda y :: s.@(f[s])y \\ \qquad \texttt{in } \ldots \text{ g[Int] } \ldots \text{ g[Int] } \ldots\end{array}$$

Here, assume function f and g are only called as shown, then we can rewrite the above programs into the following:

$$\begin{array}{l}\texttt{let } f' = \lambda x :: T(\texttt{Int}).x \\ \quad \texttt{in let } g' = \lambda y :: T(\texttt{Int}).@f'y \\ \qquad \texttt{in } \ldots \text{ g' } \ldots \text{ g' } \ldots\end{array}$$

Both f and g now become monomorphic functions. This transformation can be carried out through a

bottom up traversal: because function g is only applied to `Int`, g can be specialized to `Int` first; after this, f can be specialized in the same way.

4.2 Lambda Reduction

Type specialization will only be most effective if it is combined with conventional dataflow optimizations such as dead code elimination, common subexpression elimination, constant folding, constant propagation, and loop invariants. The middle-end optimizer does all of these. The lambda contraction phase is also a good place to carry out domain specific program analysis and program optimizations.

4.3 Representation Analysis

One novel aspect in our back end is to use the new *flexible representation analysis* technique [34] 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 [19] and SML/NJ [37] does not support unboxed representations on recursive and mutable objects; the *type-passing* approach used in TIL [38] does handle all data objects, but it involves heavy-weight runtime type analysis and code manipulations.

4.4 Closure Conversion

After the polymorphism is eliminated, we use an efficient and safe-for-space closure conversion algorithm [36] to compile the higher-order functions. Our algorithm exploits the use of compile-time control and data flow information to optimize closure representations. By extensive closure sharing and allocating as many closures in registers as possible, our closure conversion algorithm not only gives good performance but also satisfies the strong *safe for space complexity* rule [3], thus achieving good asymptotic space usage.

5 Translation into FLINT

To demonstrate the power of the FLINT language, we have built a new front end that translates the entire SML'97 [22] plus MacQueen-Tofte higher-order modules [21]) into our typed common intermediate format. This new front end and the FLINT middle end have been incorporated and released as part of the Standard ML of New Jersey compiler since version 109.24 (January 9, 1997). Translation from the Core-ML-like (or Core-Haskell-like) language to FLINT is same as the standard embedding of ML into F_ω [13]; other features such as ML datatypes are translated into FLINT type constructors. Compilation from SML higher-order modules to FLINT is quite a challenge because higher-order modules involve the use of dependent types which, in general, cannot be expressed as F_ω-like polymorphism.

Fortunately, ML-style higher-order modules have a clean phase-distinction property; the module language is completely separate from the core language. In a companion paper [35], we present a type-directed translation of the MacQueen-Tofte higher-order modules into the Core-FLINT like language. The basic idea of our algorithm is like this: we notice that every ML module can be split into a *type* part and a *value* part; the type (value) part of a structure includes all its type (value) components plus the type (value) parts of its structure and functor components; the type part of a functor is an higher-order type function from the type part of its arguments to that of its result; the value part of a functor can be viewed as a polymorphic function quantified over the type part of its arguments; functor applications can thus be expressed as a combination of type application and value application as in the Girard-Reynolds calculus. The detailed algorithm can be found in the companion paper [35].

The fact that ML-style higher-order modules can be embedded into FLINT is a good indication of FLINT's expressive power. We are currently working on translations of other source languages such as Haskell, Java, Safe C. Translating Haskell into FLINT is not much different from translating the Core-SML language. Two distinct features of Haskell are *type class* and *lazy evaluation*. Type class can be eliminated by explicit dictionary-passing, done by the type checker in the front end. Lazy evaluation requires the use of FLINT primitives, *delay* and *force*, to make the evaluation explicit. Translating Java into FLINT is less trivial, but it boils down to what kind of encodings [5] we use to model the Java objects.

We believe that FLINT is a sufficiently rich intermediate language that can be used to handle many interesting application languages. While building a

new front end will not be completely trivial, it is definitely much easier than translating into C or building a compiler from scratch. If we consider C as a common intermediate format for conventional imperative languages, FLINT plays the same role but for modern HOT languages.

6 Related Work

Common intermediate format has been an active research area in the past. Many existing compilers such as GNU's GCC, Stanford's SUIF [12], and U. Washington's Vortex [7] all use some kind of shared intermediate representations for multiple source languages. In addition, the C programming language has been used as the *de facto* common intermediate format for a long time. Of course, none of these intermediate languages are strongly typed, and neither do they support advanced HOT languages such as ML.

One example of typed intermediate format is the Java VM bytecode [20]. Like FLINT, the Java bytecode can be statically type-checked, though its type system is not as formalized as the F_ω calculus. Because the Java bytecode is originally designed for Java only, it does not directly support common HOT language features such as higher-order functions and polymorphic functions.

Typed intermediate languages have gotten a lot of attentions in the ML community lately. Several ML compilers, e.g., Gallium [19], SML/NJ [37], and TIL [38], all maintain explicit type information inside their intermediate languages. Our FLINT compiler is the first that handles the entire SML'97 plus MacQueen-Tofte higher-order modules.

Using the predicative polymorphic λ-calculus to model the type-theoretic semantics of Standard ML was pioneered by Harper and Mitchell [13]. Their XML calculus also includes dependent types to characterize ML module constructs. Harper and Morrisett [15, 23] later proposed to use a predicative variant of F_ω (but extended with `typerec`) to compile ML-like polymorphism. Recently, Harper and Stone [16] gave a new type theoretic account for the entire SML'97; the internal language they use still contain a separate module calculus and translucent types. All these work inspired us to look into the possibility of building a typed common intermediate format based on F_ω.

7 Conclusions

We have presented a new framework for constructing high-quality compilers for multiple advanced (HOT) languages. By compiling different general-purpose and application languages into a single typed intermediate format, some of the "Babel" problems associated with application languages can be nicely resolved. For example, the compiler infrastructure we are building can be quickly adapted to generate compilers for new application languages. Also, languages compiled under FLINT can interact with each other based on their static type information. They may also share a single runtime system with system-wide garbage collector and foreign function call interface.

Although the FLINT compiler has been incorporated and released with the SML/NJ compiler for a while, the design of the FLINT language is still at its very early stage. In fact, some important features such as objects and type dynamic are still not supported well. In the future, we plan to gain more experience about application languages, and to expand and evolve FLINT into a more mature intermediate language.

8 Acknowledgments

This research was sponsored in part by the DARPA 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.

9 Availability

A preliminary implementation of the FLINT intermediate language is now used by (and released with) the Standard ML of New Jersey (SML/NJ) compiler. SML/NJ is a joint work by AT&T, Lucent, Princeton and Yale; both the software and the source code are available via anonymous FTP from :

`ftp.research.bell-labs.com/pub/smlnj`

More detailed information and related papers on FLINT can be found at the following WWW page:

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

References

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

[2] A. W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3(4):343–380, 1990.

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

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

[5] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. In *Proc. Third Workshop on Foundations of Object Oriented Languages*, July 1996.

[6] 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.

[7] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proc. ACM SIGPLAN '96 Conf. on Object-Oriented Programming Systems, Languages, and applications*, pages 83–100, New York, October 1996. ACM Press.

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

[11] O. M. Group. The common object request broker: Architecture and specifications (corba). Revision 1.2., Object Management Group (OMG), Framingham,M A, December 1993.

[12] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, December 1996.

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

[14] 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.

[15] 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.

[16] 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.

[17] P. Hudak, S. P. Jones, and P. W. *et al.* Report on the programming language Haskell, a non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 21(5), May 1992.

[18] L. Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, January 1996.

[19] 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.

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

[21] D. 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.

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

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

[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] G. Nelson, editor. *Systems programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[28] S. L. 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] J. H. Reppy. CML: A higher-order concurrent language. In *Proc. ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 293–305. ACM Press, 1991.

[30] J. H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, January 1993.

[31] 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.

[32] D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[33] A. Sabry and P. Wadler. A reflection on call-by-value. In *Proc. 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 13–24. ACM Press, June 1996.

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

[36] 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.

[37] 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.

[38] 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.

[39] D. R. 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.

[40] P. Wadler. Editorial: A HOT opportunity. *Journal of Functional Programming*, 2(7), 1997.

[41] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for java. Technical Report CS-TR-546-97, Princeton University, Department of Computer Science, Princeton, NJ, April 1997.