Yale University
Department of Computer Science

Efficient Runtime Type Passing

Bratin Saha
Dept. of Computer Science
Yale University

YALEU/DCS/CS-690 Report
May 5th, 1998

Contents

List of Figures

Acknowledgements

I would like to thank my advisor Professor Zhong Shao for his help and guidance throughout the last year. We had inumerable discussions and he always found time to help out if I needed any clarifications – which often bordered on the numerous specially with regard to digging inside the FLINT/ML compiler. I also thank the other members of the FLINT group – Chris League, Stefan Monnier and Valery Trifonov – for many useful discussions.

Introduction

This report is about efficient runtime type passing. I believe that passing types at runtime is desirable and can be done efficiently.

Compilers for modern polymorphic languages like ML generate code that is not as efficient as code emitted by compilers for languages like C, Pascal etc. One of the reasons for this is that compilers for languages like C can make use of type information at compile time to determine calling conventions and allocate data efficiently. In polymorphic languages, types are not known fully at compile time and moreover the types are variable at runtime. As a result compilers for these languages have traditionally compiled by assuming a uniform data representation and a fixed calling convention. A uniform data representation implies that every value is boxed so that accessing data requires an extra level of indirection. Furthermore to support garbage collection and overloaded operators, data values must also be tagged. Therefore the advanced type system in these languages causes them to incur a runtime penalty.

One solution to the above problem is to pass types around at runtime as ordinary values. These types can then be inspected and the appropriate code executed. For example, instead of using a boxed representation for reals, we could use a natural representation. A polymorphic function can then use the type information available at runtime to access the data correctly while monomorphic code pays no extra penalty. Similarly the garbage collector could be passed type information at runtime so that it does not have to inspect tags to trace live data. Overloaded operators could also be implemented more efficiently by adopting a type passing approach. At runtime an overloaded operator could dispatch based on the type of its parameters and choose the appropriate code to execute. Runtime type passing can also be used to support various other applications like pretty printing, debugging, pickling, marshalling/unmarshalling of data etc. This shows that runtime type passing is a viable approach to compiling modern polymorphic languages like ML.

What has prevented the widespread use of this technique is the perceived runtime cost of passing types. Types in these languages specially for recursive data structures can quickly get complicated and the runtime representation of such types may be costly. Under such conditions, constructing and passing types at runtime may incur a significant overhead to the execution time of a program.

Our work therefore focusses on making type passing as efficient as possible. Through a series of transformations, we ensure that the runtime cost of passing types never blows up the execution cost. In Chapter 2, we present an optimal type lifting algorithm that eliminates runtime type construction inside functions and guarantees that all type information is constructed at linktime. In Chapter 3, we implement a common type expression elimination algorithm that ensures that we share as much of the work of constructing types as possible. The idea is similar to the common subexpression elimination done by most compilers. However since we represent types as Debruijn indices, we can not lift a conventional algorithm and apply it straight to a FLINT program. We need an algorithm that works with our representation of types. Having thus ensured that the runtime cost of passing types is optimised to a minimum, we present a new runtime representation for types in Chapter 4. This representation maintains complete type information at runtime and is therefore suitable for supporting applications like pretty printing, debugging, pickling and type dynamic. All of our work was done on the FLINT/ML compiler which served as the testbed for the entire FLINT group. We therefore give an overview of the compiler in Chapter 1.

Chapter 1

Overview of the FLINT/ML compiler

1.1 Introduction

In this chapter, we give an introduction to the FLINT/ML compiler. All our algorithms have been implemented on this compiler and it has been the testbed for the entire FLINT group.

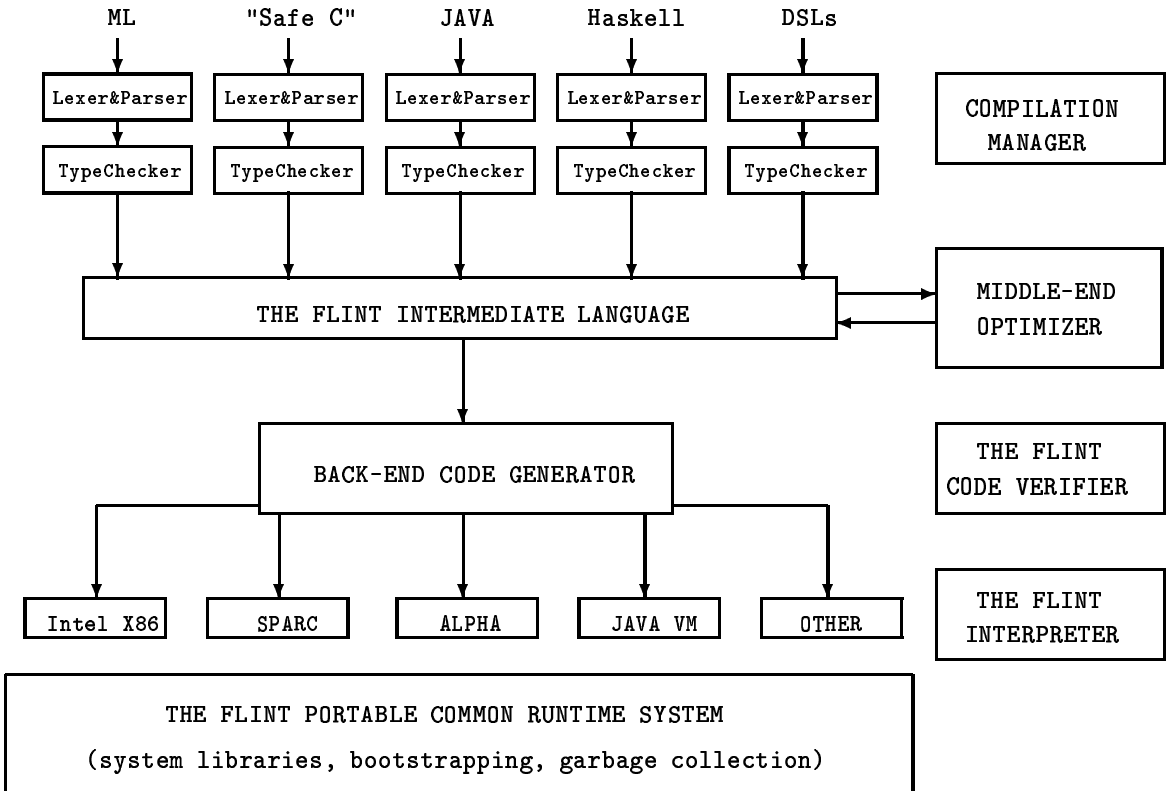

Figure 1.1: Top-Level Structure of the FLINT System

1.2 The FLINT Architecture

The FLINT system, as shown in Figure 1.1, 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, 48], 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 [43] (to compile polymorphism), safe-for-space closure conversion [46] (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 system services. Our current implementation borrows SML/NJ's runtime system [40, 2, 15] which runs under all major machine platforms. We plan to extend it to support new services such as dynamic linking and bytecode execution.

1.3 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 [10], and U. Washington's Vortex [6], 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 has proven invaluable for efficient compilation of statically typed languages [21, 47, 49]; types are also useful for debugging compilers and proving properties of programs.

1.3.1 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 collection).

- *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.

1.3.2 Background

The core language of FLINT is a predicative variant of the Girard-Reynolds polymorphic λ-calculus F_ω [9, 41], with the term language written in the A-normal form [42]. 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:

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

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 belongs 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.

1.3.3 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 [13], 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 [42]. More specifically, each function application (or type application) can only refer to values (as $@v_1v_2$). The standard F_ω function application term $@e_1e_2$ 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 [24].

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

$$
\begin{array}{llll}
(\textit{kinds}) & \kappa & ::= & \Omega \mid \kappa_1 \to \kappa_2 \mid \kappa_1 \otimes \kappa_2 \\
(\textit{con's}) & \mu & ::= & t \mid \texttt{Int} \mid \to(\mu_1, \mu_2) \\
 & & \mid & \lambda t{::}\kappa.\mu \mid \mu_1[\mu_2] \\
 & & \mid & \otimes(\mu_1, \mu_2) \mid \Pi_1\mu \mid \Pi_2\mu \\
(\textit{types}) & \sigma & ::= & T(\mu) \mid \forall t{::}\kappa.\sigma \mid \sigma_1 \to \sigma_2 \\
(\textit{terms}) & e & ::= & v \mid \lambda x{:}\sigma.e \mid @v_1v_2 \\
 & & \mid & \Lambda t{::}\kappa.e \mid v[\mu] \\
 & & \mid & \texttt{let } x = e_1 \texttt{ in } e_2 \\
(\textit{values}) & v & ::= & x \mid i
\end{array}
$$

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 $\to$. As in F_ω, the application and abstraction constructors correspond to the function kind $\kappa_1 \to \kappa_2$. 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 1.2, 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 $\Delta;\Gamma \vdash e : \sigma$ where Δ is a kind environment mapping type variables to kinds, and Γ is the type environment mapping term variables to types. Harper and Morrisett [13, 28] 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.

1.3.4 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 [45, 12] 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.

Constructor Formation and Constructor Equivalence:

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

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

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

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

Type Formation and Type Equivalence:

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

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

Term Formation:

$$(value) \qquad \frac{}{\Gamma \vdash i : \mathtt{Int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

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

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

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

Figure 1.2: The Static Semantics of Core-FLINT

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 remaining 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 [39]), suspensions (or thunks, to support lazy evaluations).

1.3.5 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 1.3 for a fragment of the FLINT definitions, written as ML datatype definitions). First, we represent all type variables as de Bruijn indices [5]. 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 subparts; (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 [30, 31] 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 [31]. The environment e determines the bindings for the type variables.

Figure 1.3 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.

1.4 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 [43] to compile polymorphism and safe-for-space closure conversion to compile higher-order functions [46]; it then does the standard register allocation, instruction scheduling, and machine code generation [8].

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

1.4.1 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:

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

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

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

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.

1.4.2 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.

1.4.3 Representation Analysis

One novel aspect in our back end is to use the new *flexible representation analysis* technique [43] 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 [21] and SML/NJ [47] does not support unboxed representations on recursive and mutable objects; the *type-passing* approach used in TIL [49] does handle all data objects, but it involves heavy-weight runtime type analysis and code manipulations.

1.4.4 Closure Conversion

After the polymorphism is eliminated, we use an efficient and safe-for-space closure conversion algorithm [46] to compile the higher-order functions. The 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, the 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.

1.5 Conclusions

To demonstrate the power of the FLINT language, we have built a new front end that translates the entire SML'97 [26] plus MacQueen-Tofte higher-order modules [24]) 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 [11]; 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.

We believe that FLINT is a sufficiently rich intermediate language. 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.

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

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

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

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

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

Figure 1.3: Representing Kinds, Constructors, and Types

Chapter 2

Optimal Type Lifting

2.1 Introduction

Modern compilers for ML-like polymorphic languages [25, 26] usually use variants of the Girard-Reynolds polymorphic λ-calculus [9, 41] as their intermediate languages (ILs). Implementation of these ILs often involves passing types explicitly as run-time parameters [50, 49, 44]: each polymorphic type variable gets instantiated to the actual type through run-time type application. Maintaining type information in this manner helps in ensuring the correctness of a compiler; more importantly, it also enables many interesting optimizations and applications. For example, both pretty-printing and debugging on polymorphic values require complete type information at runtime in order to work correctly. Intensional type analysis [13, 49, 43], which is used by some compilers [49, 44] to support efficient data representation, also requires the propagation of type information into the target code. Finally, run-time type information is crucial to the implementation of tag-less garbage collection [50], pickling, and type dynamic [22].

However, the added information available at run time as a result of type passing does not come for free. Depending on the sophistication of the type representation, run-time type passing can add a significant overhead to the time and space usage of a program. For example, Tolmach [50] implemented a tag-free garbage collector via explicit type passing; he reported that the memory allocated for type information sometimes exceeded the memory saved by the tag-free approach. Clearly, it is desirable to optimize the run-time type passing in polymorphic code [27]. In fact, a better goal would be to guarantee that explicit type passing never blows up the execution cost of a program.

Let's consider the following sample code - we took some liberties with the syntax by using an explicitly typed variant of the Core-ML. Here Λ denotes type abstraction and λ denotes value abstraction. $x[\alpha]$ denotes type application and $x(e)$ denotes term application.

```
pair = Λs.λx:s*s.
  let f = Λt.λy:t. ... (x , y)
  in  ... f[s*s](x) ...

......

main = Λα.λa:α.
  let doit = λi:Int.
        let elem = Array.sub[α*α](a,i)
        in  ... pair[α](elem) ...

      loop = λn1:Int.λn2:Int.λg:Int→Unit.
              if n1 <= n2
                (g(n1);
                 loop(n1+1,n2,g))
              else ()
   in loop(1,n,doit)
```

Here, `f` is a polymorphic function defined inside function `pair`; it references the parameter `x` of `pair` so `f` cannot be easily lifted outside `pair`. Function `main` executes a loop: in each iteration, it selects an element `elem` of the array `a` and then performs some computation (i.e, `pair`) on it. Executing the function `doit` results in three type applications, the `Array.sub` function, `pair`, and `f`. In each iteration, `sub` and `pair` are applied to types $\alpha * \alpha$ and α respectively. A clever compiler may do a *loop-invariant removal* [1] to avoid the repeated type construction (e.g., $\alpha * \alpha$) and application (e.g., `pair`$[\alpha]$). Notice that optimizing type applications such as `f[s*s]` is much less obvious; `f` is nested inside `pair`, and its repeated type applications are not apparent in the `doit` function. In other words, the loop invariant and the loop body are in different functions !! We may type-specialize `f` but in general this may lead to substantial code duplication. Every time `doit` is called, `pair`$[\alpha]$ gets executed and then every time `pair` is called with all its arguments, `f[s*s]` will be executed. Since `loop` calls `doit` repeatedly and each such call generates type applications of `pair` and `f`, we are forced to incur the overhead of repeated type construction and application. If the type representation is complicated, this is clearly expensive.

In this chapter, we present an algorithm that minimizes the cost of run-time type passing. More specifically, the optimization eliminates all type application inside any core-language function - it guarantees that the amount of type information constructed at runtime is a static constant. This guarantee is important because it allows us to use more sophisticated representations for run-time types (say, to suit the needs of certain application), yet not have to worry about the run-time cost of doing so. We know for sure that this will not increase the execution cost significantly.

The basic idea is as follows. We lift all polymorphic function definitions and type applications in a program to the "top" level. By top level, we mean "outside any core-language function." Intuitively, no type application is nested inside any function abstraction (λ); they are nested only inside type abstractions (Λ). All type applications are now top-level code and the type information is resolved once and for all at the beginning of execution of each compilation unit. In essence, the code after our type lifting would perform all of its type applications at "link" time.[1] In fact, the number of type applications performed and the amount of type information constructed can be determined statically.

This leads us to a natural question. Why do we restrict the transformation to type applications alone? Obviously the transformation could be carried out on value computations as well but what makes type computations more amenable to this transformation is that we can guarantee that all type computations can be lifted to the top level. Moreover, while the transformation is also intended to increase the runtime efficiency, a more important goal is to ensure that type passing in itself is not costly. This will allow us to use a more sophisticated type system and make greater use of type information at runtime.

[1] By "link" time, we don't really mean the link time in the traditional sense. Rather, we use it to refer to the run time spent on module initialization and module linkage (e.g., functor application) in an ML-style module language.

We describe the algorithm in later sections and also prove that it is both type-preserving and semantically sound. We have implemented it in the FLINT/ML compiler [44] and tested it on a few benchmarks. We provide the implementation results at the end of this chapter.

2.2 The Type Lifting Algorithm

This section presents our optimal type lifting algorithm. We use an explicitly typed variant of the Core-ML calculus [11], shown in Figure 2.1, as the source and target languages. The type lifting algorithm (Fig. 2.2) is expressed as a type-directed program transformation that lifts all type applications to the top-level. We illustrate the algorithm on an example program and then prove the type correctness and the semantic correctness of our translation.

2.2.1 The language

We use an explicitly typed variant of the Core-ML calculus [11] as our source and target languages. The syntax is shown in Fig 2.1. The static and dynamic semantics are all standard, and are given in (Sec 2.4) along with the proofs.

$$
\begin{array}{llll}
(\textit{con's}) & \mu & ::= & t \mid \texttt{Int} \mid \mu_1 \rightarrow \mu_2 \\
(\textit{types}) & \sigma & ::= & \mu \mid \forall \overline{t_i}.\, \mu \\
(\textit{terms}) & e & ::= & i \mid x \mid \lambda x{:}\mu.e \mid @x_1 x_2 \mid \texttt{let } x = e \texttt{ in } e' \mid \texttt{let } x = \Lambda \overline{t_i}.\, e_v \texttt{ in } e \mid x[\overline{\mu_i}] \\
(\textit{vterms}) & e_v & ::= & i \mid x \mid \lambda x{:}\mu.e \mid \texttt{let } x = e_v \texttt{ in } e'_v \mid \texttt{let } x = \Lambda \overline{t_i}.\, e_v \texttt{ in } e'_v \mid x[\overline{\mu_i}]
\end{array}
$$

Figure 2.1: An explicit Core-ML calculus

Here, terms e consist of identifiers (x), integer constants (i), function abstractions, function applications, and let expressions. We differentiate between monomorphic and polymorphic let expressions in our language. We use $\overline{t_i}$ (and $\overline{\mu_i}$) to denote a sequence of type variables $t_1, ..., t_n$ (and types) so $\forall \overline{t_i}.\, \mu$ is equivalent to $\forall t_1 \ldots \forall t_n.\mu$.

There are several aspects of this calculus that are worth noting. First, we restrict polymorphic definitions to value expressions (e_v) only so that moving type functions and type applications is semantically sound [51]. Variables introduced by normal λ-abstraction are always monomorphic, and polymorphic functions are introduced only by the **let** construct. In our calculus, type applications of polymorphic functions are never curried and therefore in the algorithm in Fig 2.2, the *exp* rule assumes that the variable is monomorphic. The *tapp* rule also assumes that the type application is not curried and therefore the newly introduced variable `v` (denoting the lifted type application) is monomorphic and is not further type applied. Finally, following SML [26, 25], polymorphic functions are not recursive. [2] *This restriction is crucial to proving that all type applications can be lifted to the top level.*

Throughout the paper we take a few liberties with the syntax: we allow ourselves infix operators, multiple definitions in a single `let` expression to abbreviate a sequence of nested `let` expressions, and term applications that are at times not in A-Normal form [7]. We also use indentation to indicate the program nesting structure.

2.2.2 Informal description

Before we move on to the formal description of the algorithm, we will present the basic ideas informally.

Define the depth of a term in a program to be equal to the number of λ(value) abstractions within which it is nested. Consider the terms outside any value abstraction to be at depth zero. Since terms at depth zero occur

[2] Our current calculus does not support recursions, but recursive functions can be easily added. As in SML, recursive functions are always monomorphic.

outside all functions, they are necessarily outside all loops in the program. In a strict language like ML, all these terms are evaluated once and for all at the beginning of program execution. This may create new bindings in the environment but since these terms never occur inside functions (and hence loops) they are never reevaluated.

We want to avoid repeated type applications and therefore the algorithm aims to move all type applications to depth zero. This will ensure as we said before that type applications are carried out once and for all at the beginning of program execution. But since we want to lift type applications, we must also lift the type functions to depth zero. Therefore as the algorithm scans the input program, it collects all the type applications and polymorphic functions occuring at depth greater than zero and adds them to a list `H`. (In the algorithm given in Fig 2.2, the depth is implicitly assumed to be greater than zero). When the algorithm returns to the top level of the program, it dumps the expressions contained in the list.

We will illustrate the algorithm on the sample code given in Sec 2.1 where the lifting is pretty straightforward. But before that, we want to clarify one remaining feature of the algorithm. The type computations contained in the body of a polymorphic definition *(rule tfn)* are dumped right in front of the type abstraction. Since this polymorphic definition will in turn also get hoisted to depth zero and type abstractions do not increase the depth, the dumped expressions also get lifted to the top level.

```
pair = Λs.λx:s*s.
  let f = Λt.λy:t. ... (x , y)
   in ... f[s*s](x) ...

......

main = Λα.λa:α.
  let doit = λi:Int.
        let elem = Array.sub[α*α](a,i)
         in ... pair[α](elem) ...

      loop = λn1:Int.λn2:Int.λg:Int→Unit.
              if n1 <= n2
                (g(n1);
                 loop(n1+1,n2,g))
              else ()
   in loop(1,n,doit)
```

In the example above (reproduced from Sec 2.1), `f[s*s]` is at depth 1 since it occurs inside the λx, `Array.sub`$[\alpha*\alpha]$ and `pair`$[\alpha]$ are at depth 2 since they occur inside the λa and λi. We want all of these type applications to occur at depth zero. Transforming `main` first, the resulting code becomes –

```
pair = Λs.λx:s*s.
  let f = Λt.λy:t. ... (x , y)
  in  ... f[s*s](x) ...

......

main =
  Λα.
     let v1 = Array.sub[α*α]
         v2 = pair[α]
     in  λa:α.
            let
              doit = λi:Int.
                   let elem = v1(a,i)
                   in ... v2(elem) ...

              loop = λn1:Int.λn2:Int.λg:Int→Unit.
                        if n1 <= n2
                          (g(n1);
                           loop(n1+1,n2,g))
                        else ()
            in loop(1,n,doit)
```

We then lift the type application of `f` which means we must also lift `f`'s definition by abstracting over its free variables. The resulting code becomes –

```
pair = Λs.
  let f = Λt.λx:s*s.λy:t. ... (x , y)
      v3 = f[s*s]
  in  λx:s*s. ... (v3(x))(x) ...

......

main =
   Λα.
      let v1 = Array.sub[α*α]
          v2 = pair[α]
      in  λa:α.
             let doit =
                   λi:Int.
                       let elem = v1(a,i)
                       in  ... v2(elem) ...

                 loop =
                   λn1:Int.λn2:Int.λg:Int→Unit.
                       if n1 <= n2
                         (g(n1);
                          loop(n1+1,n2,g))
                       else ()
             in loop(1,n,doit)
```

In the code above, all type applications occur at depth zero. Therefore when `main` is called at the beginning of execution, `v1, v2 and v3` get evaluated which results in the type applications being performed now. During execution, when `loop` runs through the array and `g`(which is really `doit`) is repeatedly applied, none of the type applications are repeated since the type specialised functions are already available.

$$(exp) \qquad \frac{\Gamma(x) = (\mu, \_)}{\Gamma \vdash x : \mu \Rightarrow x; \emptyset; \{x : \mu\}} \qquad\qquad \Gamma \vdash i : \texttt{Int} \Rightarrow i; \emptyset; \emptyset$$

$$(app) \qquad \frac{\Gamma(x_1) = \langle \mu_1 \to \mu_2, \_\rangle \qquad \Gamma(x_2) = \langle \mu_1, \_\rangle}{\Gamma \vdash @x_1x_2 : \mu_2 \Rightarrow @x_1x_2; \emptyset; \{x_1 : \mu_1 \to \mu_2, x_2 : \mu_1\}}$$

$$(fn) \qquad \frac{\Gamma[x \mapsto \langle \mu, \_\rangle] \vdash e : \mu' \Rightarrow e'; H; F}{\Gamma \vdash \lambda x : \mu.e : \mu \to \mu' \Rightarrow \lambda x : \mu.e'; H; F \backslash \{x : \mu\}}$$

$$(let) \qquad \frac{\Gamma \vdash e_1 : \mu_1 \Rightarrow e_1'; H_1; F_1 \quad \Gamma[x \mapsto \langle \mu_1, \_\rangle] \vdash e_2 : \mu_2 \Rightarrow e_2'; H_2; F_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \mu_2 \Rightarrow \texttt{let } x = e_1' \texttt{ in } e_2'; H_1 || H_2; F_1 \cup (F_2 \backslash \{x : \mu_1\})}$$

$$(tfn) \qquad \frac{\Gamma \vdash e_1 : \mu_1 \Rightarrow e_1'; H_1; F_1 \qquad L = List(F_1) \qquad \Gamma[x \mapsto \langle \forall \overline{t_i}.\mu_1, L\rangle] \vdash e_2 : \mu_2 \Rightarrow e_2'; H_2; F_2}{\Gamma \vdash \texttt{let } x = \Lambda \overline{t_i}.e_1 \texttt{ in } e_2 : \mu_2 \Rightarrow e_2'; \underbrace{\langle x, \Lambda \overline{t_i}.\, LET(H_1, \lambda^* L.e_1')\rangle}_{H_r} :: H_2; F_2}$$

$$(tapp) \qquad \frac{\Gamma(x) = \langle \forall \overline{t_i}.\mu, L\rangle \qquad v \text{ a fresh variable}}{\Gamma \vdash x[\overline{\mu_i}] : [\mu_i/t_i]\mu \Rightarrow @^* vL; \underbrace{[\langle v, x[\overline{\mu_i}]\rangle]}_{H_r}; Set(L)}$$

Figure 2.2: The Lifting Translation

2.2.3 Formalization

Figure 2.2 gives our type-directed lifting algorithm. The translation is defined as a relation of the form $\Gamma \vdash e : \mu \Rightarrow e'; H; F$, that carries the meaning that $\Gamma \vdash e : \mu$ is a derivable typing in the input program, the translation of the input term e is the term e', H is the set of type expressions generated by the translation that must be lifted to the top level and F is the set of free variables[3] of the translated term e'. H is also referred to hereafter as the header. H consists of the polymorphic function definitions and the type applications occuring in the input program that after translation are dumped at the top level. The final result of lifting a closed term e of type μ for which the algorithm infers $\emptyset \vdash e : \mu \Rightarrow e'; H; \emptyset$ is $LET(H, e')$, where the function $LET(H, e)$ expands a list of bindings $H = [\langle x_1, e_1\rangle, \ldots, \langle x_n, e_n\rangle]$ and a term e into the term $\texttt{let } x_1 = e_1 \texttt{ in } \ldots \texttt{let } x_n = e_n \texttt{ in } e$.

The environment Γ maps a variable to its type and also to a list of the free variables in its definition in the case of `let`-bound polymorphic variables. We use standard notation for lists and operations on them in the algorithm; in addition, the functions *List* and *Set* convert between lists and sets of variables using a canonical ordering. The functions λ^* and $@^*$ are defined so that $\lambda^* L.\, e$ and $@^* vL$ reduce to $\lambda x_1 : \mu_1 \ldots \lambda x_n : \mu_n.e$ and $@(\ldots(@vx_1)\ldots)x_n$, respectively, where $L = [x_1 : \mu_1, \ldots, x_n : \mu_n]$.

Rules (*exp*) and (*app*) are just the identity transformations. Since the IL is in the A-normal form [7], no type applications can occur in these terms.

Rule (*fn*) deals with abstractions. We first translate the body of the abstraction. H now contains the type applications and the type functions that were in e and is returned as the header from the translation. The resulting term is an abstraction over the translated expression.

The translation of monomorphic `let` expressions is similar. We translate each of the subexpressions replacing

[3] Actually, this includes only the momorphically typed free variables of e'.

the old terms with the new ones and return this as the result of the translation. The header H of the translation is the concatenation of the headers H_1 and H_2 obtained in the translation of the subexpressions.

The real work is done in the last two rules which are the ones dealing with type expressions. In rule (*tfn*), we first translate the body of the polymorphic function definition. H_1 now contains all the type expressions that were in e_1 and F_1 is the free variables of e_1'. We then translate the body of the `let` expression in an augmented environment that binds x to its type and its free variables. We will need this information when we encounter a type application of x. The translation of e_2 returns a type lifted expression e_2' and H_2 which contains the type expressions occuring in e_2. The result of the translation is only e_2' while the polymorphic definition introduced by the **let** is put into the result header H_r so that it is lifted to the top level and dumped there. The definition of x (in H_r) is closed by abstracting over its free variables L with the header H_1 dumped right after the type abstractions. Note that since H_r will be lifted to the top level, the expressions in H_1 will also as a result get lifted to the top level.

The (*tapp*) rule replaces the type application by a new variable (v) applied to the free variables(L) in the definition of x and adds the pair $(v, x[\overline{\mu_i}])$ to the header. When the header is dumped at the top, v will get bound to the type application. Note that the free variables of the translated term do not include the newly introduced variable v. This is because when the header obtained from translating an expression is written out at the top level, the translated expression remains in the scope of the dumped header. Therefore the new variable need not be abstracted.

Proposition 2.2.1 *Suppose* $\Gamma \vdash e : \mu \Rightarrow e'; H; F$. *Then in the expression LET(H,e'), e' does not contain any type application and H does not have any type application nested inside a value abstraction.*

This propostion can be proved by a simple structural induction on the structure of the source term e.

Theorem 2.2.2 (Full Lifting) *Suppose that the translation yields* $\Gamma \vdash e : \mu \Rightarrow e'; H; F$. *Then the expression LET(H,e'), does not have any type application nested inside a value abstraction.*

The theorem follows from Proposition 2.2.1.

2.2.4 An example

This section illustrates the algorithm on an example program fragment. We show the construction of the header and the translated expression as the algorithm proceeds. The notation used for the intermediate structures is the same as in Fig 2.2. The program fragment used for the example is shown below.

```
Λt1.Λt2.λx:t1.λy:t2.
      let f = Λt3.Λt4.λu:t3.λv:t4. (v,u,x)
  in let g = Λt5.λz:t5. @ (@ (f[t5][t1]) z) x
  in @ (g[t2]) y
```

The number at the beginning of each block of code denotes the sequence of transformations.

1. After translating `f's` body

```
e1' = λu:t3.λv:t4. (v,u,x)
H1 = []
F1 = {x:t1}
```

2. Now `g's` body is translated

```
e1' = λz:t5. @(@(@v1x)z)x
H1 = v1 = f[t5][t1]
F1 = {x:t1}
```

3. Now the body of the inner let

```
e'2 = @(@v2x)y

H2 = v2 = g[t2]

F2 = {x:t1,y:t2}
```

4. The inner let as a whole returns

```
e' = @(@v2x)y
H = g = (Λt5.
          let v1 = f[t5][t1]
          in λx:t1.λz:t5. @(@(@v1x)z)x ) ::
        [ v2 = g[t2] ]
F = {x:t1,y:t2}
```

5. For the outer let

```
e'2 = @(@v2x)y
H2 = g = (Λt5.
          let v1 = f[t5][t1]
          in λx:t1.λz:t5. @(@(@v1x)z)x) ::
        [ v2 = g[t2] ]
F2 = {x:t1,y:t2}
```

6. The outer let as a whole returns

```
e' = @(@v2x)y
F = {x:t1,y:t2}
H = f = (Λt3.Λt4.λx:t1.λu:t3.λv:t4. (v,u,x) ) ::
    g = (Λt5.
          let v1 = f[t5][t1]
          in λx:t1.λz:t5. @(@(@v1x)z)x ) ::
        [ v2 = g[t2] ]
```

7. After translating the lambda abstraction

```
e' = @(@v2x)y
F = []
H = f = (Λt3.Λt4.λx:t1.λu:t3.λv:t4. (v,u,x) ) ::
    g = (Λt5.
          let v1 = f[t5][t1]
          in λx:t1.λz:t5. @(@(@v1x)z)x) ::
        [ v2 = g[t2] ]
```

The final translated code with all type applications lifted:

```
Λt1.Λt2.
   let f = Λt3.Λt4.λx:t1.λu:t3.λv:t4. (v,u,x)
   let g = Λt5.
            let v1 = f[t5][t1]
            in λx:t1.λz:t5. @(@(@v1x)z)x
   let v2 = g[t2]
   in λx:t1.λy:t2. @(@v2x)y
```

2.3 Comparison with Jones' and Minamide's optimisations

There are two transformations taking place simultaneously. One is the lifting of type applications and the other is the lifting of polymorphic function definitions. At first glance, the lifting of function definitions may seem

similar to lambda lifting [17]. However the lifting in the two cases is different. Lambda lifting converts a program with local function definitions into a program consisting only of global function definitions whereas the lifting shown here preserves the nesting structure of the program.

The lifting of type applications is similar in spirit to the hoisting of loop invariant expressions outside a loop. It could be considered as a special case of a *fully lazy transformation* [16, 37] with the maximal free subexpressions restricted to be type computations. However, the fully-lazy transformation as described in Peyton Jones [37] will not lift all type applications to the top level. Specifically, type applications of a polymorphic function that is defined inside other functions will not be lifted to the top level. Our algorithm though is guaranteed to lift all type applications to depth zero. As an example, we show below a fully lazy transformation on the code fragment at the beginning of this subsection.

```
Λαβγ.
  let u = β*γ
  λx1.
     let
       y1 = Λt1.
            let u1 = f[t1*α]
            in  λx2. u1(x2) ...
       y2 = Λt3.
            let u2 = y1[β*t3]
            in  λx3. u2(x3) ...
       y3 = Λt5.
            let u3 = y2[γ*t5]
            in  λx4. u3(x4) ...
     in y3[u]x1
```

Minamide [27] has also worked on the same problem but has used an entirely different approach from ours. He lifts the construction of type parameters from within a polymorphic function to the call sites of the function. This lifting is propagated from the innermost type functions to the type applications at the top level. At runtime, type construction is replaced by projection from type parameters. However, this increases the number of type parameters of a polymorphic function since all the types that were previously constructed inside functions are now passed in as parameters. Minamide therefore considers the uncurried version of this transformation. As an example, consider his transformation on the code fragment shown at the beginning of this subsection. In the example code below, type information is passed by the evidence variable u. It is assumed to hold an evidence value that satisfies the predicate pr. $\#i(u)$ refers to the i^{th} field of u.

```
Λu:pr0.λx1.
  let
     y1 = Λu:pr1.λx2.f[#2(u)]x2...
     y2 = Λu:pr2.λx3.y1[#2(u)]x3...
     y3 = Λu:pr3.λx4.y2[#2(u)]x4...
  in
     y3[#4(u)]x1

pr0 = {α,β,γ,{β*γ,{γ*β*γ,{β*γ*β*γ,β*γ*β*γ*α}}}}
pr1 = {t1,t1*α}
pr2 = {t3,{β*t3,β*t3*α}}
pr3 = {t5,{γ*t5,{β*γ*t5,β*γ*t5*α}}}
```

The advantage of his method is that he eliminates the runtime construction of types and replaces it by projection from type records. Even though he mentions in his paper that his calculus obeys the value restriction, the transformation does not depend on it critically. However, the disadvantage is that he can no longer type-check his transformation with the existing type system; instead, he has to make use of an auxiliary type system based on the qualified type system of Jones [19] and the implementation calculus for the compilation of polymorphic

records of Ohori [33]. Our algorithm on the other hand is a source-to-source transformation whose output can be type-checked with the type-checker for the source program. Finally, Minamide's algorithm deals only with the Core-ML calculus and does not mention how his method may be extended to ML-style modules. In our case though, we have implemented our algorithm on the entire SML'97 language with higher-order modules.

Jones [18] has also worked on a similar problem related to type classes in the implementation of Haskell and Gofer. Type classes in these languages are implemented by dictionary passing and if done naively can lead to the same dictionaries being created repeatedly. Dictionaries are tuples of functions that implement the methods defined in a **Class**. At runtime, a dictionary for the type at which an overloaded operator will be used is created and passed to the function. The operation is then performed by selecting out the appropriate functions from the dictionary.

We will briefly compare our approach with his optimisations on dictionary passing - we will not however talk about eliminating dictionaries through partial evaluation [20]. Since the type systems and the implementation of dictionaries differs slightly in Haskell and Gofer, we will consider the two separately.

Haskell [14] performs context reduction and simplifies the set of constraints in a type. Consider the following Haskell example

```
f :: Eq a => a -> a -> Bool
f x y = ([x] == [y]) && ([y] == [x])
```

The actual type of `f` is $Eq[a] => a \rightarrow a \rightarrow Bool$ from where after context reduction we get the type as specified in the example code. Here $[a]$ means a list of elements of type a. $Eq\ a$ means that the type a must be an instance of the Equality Class. $Eq\ [a]$ means that the type $List\ of\ a's$ must be an instance of the Equality Class. Function `f` as shown above has type $a \rightarrow a \rightarrow Bool$, but a must be an instance of the Equality class. Jones optimises this by constructing a dictionary for *Eq [a]* at the call site of `f` rather than pass a dictionary for $Eq\ a$ and construct the dictionary for *Eq [a]* inside the function `f`. He repeats this for all overloaded functions so that he constructs dictionaries only once at the beginning of the program much as we perform all type applications at the beginning of the program. But this approach does not work with separately compiled modules because the type of `f` that is exported to other modules does not specify the dictionaries that are constructed inside it - therefore if `f` is called from a different module it will still be passed the dictionary for $Eq\ a$ and the dictionary for *Eq [a]* will be constructed during execution of `f`.

In Gofer [18], however, instance declarations are not used to simplify the context. Therefore the type of `f` in the above example would be $Eq[a] => a \rightarrow a \rightarrow Bool$. Jones' optimisation of dictionary parameters can now be performed even in the presence of separately compiled modules.

Dictionaries in Haskell and types in ML share a similarity of purpose - both of them are used to specialise a non-monomorphic function to values of a particular type. Therefore Jones' optimisation of dictionary passing may be adapted to type passing in ML. In that case, we would lift the construction of types from function bodies to the call sites of the function, perform this optimisation repeatedly and ensure that types are constructed only once at the beginning. In fact, Minamide's transformation is very similar to this.

However, the ML module language (which we consider in Sec 2.5) supports functors that are modelled by polymorphic abstractions - by this we mean that the abstracted variable is polymorphic. Suppose two polymorphic functions `f` and `g` have the same type(σ) but they construct different types in their bodies. If we transform the functions so that the types that are constructed are passed in as parameters, the two functions `f` and `g` will no longer have the same type and the FLINT(Sec 2.5) typechecker will not, in general, type check the code. This is because if we have a function $(\lambda x : \sigma.e)$, we could previously pass either `f` or `g` as parameters. But now since the two functions have different types, we cannot use them in the same context. So the method used by Jones for optimising dictionary passing does not extend to the Full-ML language.

$$
\begin{array}{llll}
(con's) & \mu & ::= & t \mid \texttt{Int} \mid \mu_1 \rightarrow \mu_2 \\
(types) & \sigma & ::= & \mu \mid \forall \overline{t_i}.\, \mu \\
(terms) & e & ::= & i \mid x \mid \lambda x{:}\mu.e \mid @x_1 x_2 \mid \texttt{let } x = e \texttt{ in } e' \mid \texttt{let } x = \Lambda \overline{t_i}.\, e_v \texttt{ in } e \mid x[\overline{\mu_i}] \\
(vterms) & e_v & ::= & i \mid x \mid \lambda x{:}\mu.e \mid \texttt{let } x = e_v \texttt{ in } e'_v \mid \texttt{let } x = \Lambda \overline{t_i}.\, e_v \texttt{ in } e'_v \mid x[\overline{\mu_i}]
\end{array}
$$

Figure 2.3: The Explicit Core-ML calculus

2.4 Correctness of the Algorithm

In this section, we give the proofs of the type preservation theorem and the semantic-soundness theorem. We first repeat the definitions of our source calculus (see Fig 2.1) used in the translation algorithm as in Figure 2.3.

$$
\begin{array}{ll}
(const/var) & \dfrac{}{\Gamma \vdash i : \texttt{Int}} \qquad \dfrac{}{\Gamma \vdash x : \Gamma(x)} \\[2ex]
(fn) & \dfrac{\Gamma \uplus \{x : \mu_1\} \vdash e : \mu_2}{\Gamma \vdash \lambda x : \mu_1.e : \mu_1 \rightarrow \mu_2} \\[2ex]
(app) & \dfrac{\Gamma \vdash x_1 : \mu' \rightarrow \mu \quad \Gamma \vdash x_2 : \mu'}{\Gamma \vdash @x_1 x_2 : \mu} \\[2ex]
(tfn) & \dfrac{\Gamma \vdash e_v : \mu_1 \quad \Gamma \uplus \{x : \forall \overline{t_i}.\mu_1\} \vdash e : \mu_2}{\Gamma \vdash \texttt{let } x = \Lambda \overline{t_i}.e_v \texttt{ in } e : \mu_2} \\[2ex]
(tapp) & \dfrac{\Gamma \vdash x : \forall \overline{t_i}.\mu}{\Gamma \vdash x[\overline{\mu_i}] : [\mu_i/t_i]\mu} \\[2ex]
(let) & \dfrac{\Gamma \vdash e_1 : \mu_1 \qquad \Gamma \uplus \{x : \mu_1\} \vdash e_2 : \mu_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \mu_2}
\end{array}
$$

Figure 2.4: Static Semantics

Figure 2.4 gives the typing rules which are useful in defining the type-preservation theorem. Figure 2.5 gives a slightly modified version of the translation algorithm given in Figure 2.2. More specifically, we divide the type environment Γ into two separate ones: one for monomorphic variables (Γ_m), and another for polymorphic variables (Γ_p). This is used only in proving the theorems; it doesnot change the semantics of the algorithm in any way.

Notation 1 *In the figure and the rest of this section we use $\lambda^* F.e$ and $@^* zF$ to denote repeated abstractions and applications respectively. If $F = \{x_1, ..., x_n\}$, then $\lambda^* F.e$ reduces to $\lambda x_1 : \mu_1.(...(\lambda x_n : \mu_n.e)..)$ where $\mu_1, ...\mu_n$ are the types of $x_1, ..., x_n$ in Γ_m. Similarly $@^* zF$ reduces to $@(..(@zx_1)..)x_n$.*

Throughout this section, we assume lambda-bound identifiers are always unique, and there are no variable redefinitions in the source term. In the following, we will state some lemmas that are used in proving subsequent theorems. We omit the proofs for these lemmas because they follow in a straightforward manner.

$$(exp) \qquad \frac{\Gamma_m(x) = \mu}{\Gamma_m; \Gamma_p; H \vdash x : \mu \Rightarrow x; \emptyset; \{x\}} \qquad \Gamma_m; \Gamma_p; H \vdash i : \texttt{Int} \Rightarrow i; \emptyset; \emptyset$$

$$(app) \qquad \frac{\Gamma_m(x_1) = \mu_1 \to \mu_2 \qquad \Gamma_m(x_2) = \mu_1}{\Gamma_m; \Gamma_p; H \vdash @x_1 x_2 : \mu_2 \Rightarrow @x_1 x_2; \emptyset; \{x_1, x_2\}}$$

$$(fn) \qquad \frac{\Gamma_m[x \mapsto \mu]; \Gamma_p; H \vdash e : \mu' \Rightarrow e'; H_1; F}{\Gamma_m; \Gamma_p; H \vdash \lambda x : \mu.e : \mu \to \mu' \Rightarrow \lambda x : \mu.e'; H_1; F \backslash \{x : \mu\}}$$

$$(let) \qquad \frac{\Gamma_m; \Gamma_p; H \vdash e_1 : \mu_1 \Rightarrow e_1'; H_1; F_1 \quad \Gamma_m[x \mapsto \mu_1]; \Gamma_p; H \vdash e_2 : \mu_2 \Rightarrow e_2'; H_2; F_2}{\Gamma_m; \Gamma_p; H \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \mu_2 \Rightarrow \texttt{let } x = e_1' \texttt{ in } e_2'; H_1 + H_2; F_1 \cup (F_2 \backslash \{x\})}$$

$$(tfn) \qquad \frac{\begin{array}{c}\Gamma_m; \Gamma_p; H \vdash e_1 : \mu_1 \Rightarrow e_1'; H_1'; F_1 \qquad H_1 = \langle x = \Lambda \overline{t_i}.Let\ H_1'\ in\ \lambda^* F_1.e_1', \forall \overline{t_i}.T(F_1) \to \mu_1 \rangle \\ \Gamma_m; \Gamma_p[x \mapsto \langle \forall \overline{t_i}.\mu_1, F_1 \rangle]; H + H_1 \vdash e_2 : \mu_2 \Rightarrow e_2'; H_2; F_2\end{array}}{\Gamma_m; \Gamma_p; H \vdash \texttt{let } x = \Lambda \overline{t_i}.e_1 \texttt{ in } e_2 : \mu_2 \Rightarrow e_2'; H_1 + H_2; F_2}$$

$$(tapp) \qquad \frac{\Gamma_p(x) = \langle \forall \overline{t_i}.\mu, F \rangle \qquad \Gamma_H(x) = \forall \overline{t_i}.T(F) \to \mu \qquad z \text{ a fresh variable}}{\Gamma_m; \Gamma_p; H \vdash x[\overline{\mu_i}] : [\mu_i/t_i]\mu \Rightarrow @^* z F; \underbrace{\langle z = x[\overline{\mu_i}], T(F) \to [\mu_i/t_i]\mu \rangle}_{H_1}; F}$$

Figure 2.5: The Lifting Translation

Notation 2 *In the lemmas and the rest of the section, if L is a set of variables, then $T(L)$ refers to the types of the variables in L under the environment Γ_m. If $L = \{x_1, x_2, ..., x_n\}$ and the types of the variables are respectively $\mu_1, ..., \mu_n$, then $T(L) \to \tau$ is shorthand for $\mu_1 \to (... \to (\mu_n \to \tau)..)$. Γ_m and Γ_p bind monomorphic and polymorphic variables to their types respectively while a_m and a_p bind monomorphic and polymorphic variables to their values respectively.*

Lemma 2.4.1 *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \lambda L.e : T(L) \to \tau$ where L is the set of free variables of e and $T(L)$ respects Γ.*

Lemma 2.4.2 (H is closed) *If H is closed and $\Gamma_m; \Gamma_p; H \vdash e : \mu \Rightarrow e'; H_1; F$, then $H + H_1$ is closed.*

Proof. Lemma 2.4.2 follows directly by structural induction on the structure of e.

Lemma 2.4.3 *If $\Gamma_m; \Gamma_p; H \vdash e : \mu \Rightarrow e'; H_1; F$, and the set of variables bound by Γ_m and H are disjoint, then the set of variables bound by Γ_m and $H + H_1$ are disjoint.*

Proof By structural induction on e. Notice that Γ_m binds only monomorphic variables and H binds only polymorphic or newly introduced variables.

Lemma 2.4.4 *If $\Gamma_m; \Gamma_p; H \vdash e : \mu \Rightarrow e'; H_1; F$, then the variables in F are bound in Γ_m.*

Proof. Again by straightforward induction on the structure of e.

The translation of a closed term e occurs as

$$\emptyset;\emptyset;\emptyset \vdash e : \mu \Rightarrow e';H;\emptyset$$

Therefore initially H is closed and the set of variables bound by Γ_m and H are disjoint. This leads to the following corollary —

Corollary 2.4.5 *During the translation, H is always closed and the set of variables bound by Γ_m and H are always disjoint.*

2.4.1 Type Preservation

Before we prove the type soundness of the translation, we will define a couple of predicates on the header — Γ_H and well-typedness of H. Intuitively, Γ_H denotes the type that we annotate with each expression in H during the translation and well-typedness ensures that the type we annotate is the correct type. Together these two ensure that the header formed is well typed.

Definition 2.4.6 (Let H in e)

If $H = h_0 \ldots h_n$, then **Let H in e** is shorthand for *let* h_0 *in* ... *let* h_n *in* e. The typing rule is as follows — $\Gamma_m;\Gamma_H \vdash$ *let* h *in* $e : \mu$ iff $\Gamma_m;\Gamma_H + \Gamma_h \vdash e : \mu$.

Definition 2.4.7 (Γ_H)

If $H = (h_0 \ldots h_n)$, then $\Gamma_H = \Gamma_{h_0} \ldots \Gamma_{h_n}$. If $h_i ::= (x = e, \tau)$, then $\Gamma_{h_i} := x \mapsto \tau$.

Definition 2.4.8 (H is well typed)

H is well typed if $h_0 ... h_n$ are well typed. h_i is well typed if $h_0 ... h_{i-1}$ are well typed and —

- $h_i ::= (x = \Lambda\overline{t_i}.Let\ H_1\ in\ e, \forall\overline{t_i}.\mu)$, then $\Gamma_{h_0..h_{i-1}} \vdash Let\ H_1\ in\ e : \mu$.
- $h_i ::= (z = x[\overline{\mu_i}], [\mu_i/t_i]\mu)$, then $\Gamma_{h_0 \ldots h_{i-1}} \vdash x : \forall\overline{t_i}.\mu$

Theorem 2.4.9 (Type Preservation) *Suppose $\Gamma_m;\Gamma_p;H \vdash e : \mu \Rightarrow e';H_1;F$. If H is well typed then $H + H_1$ is well typed and if $\Gamma_m;\Gamma_p \vdash e : \mu$ then $\Gamma_m;\Gamma_H \vdash Let\ H_1\ in\ e' : \mu$.*

Proof. The proof is by structural induction on the structure of e. We will consider the two interesting cases here — tfn and *tapp*.

Case tapp = To prove - given H is well-typed, then $H + \overbrace{(z = x[\overline{\mu_i}], T(F) \rightarrow [\mu_i/t_i]\mu)}^{H'}$ is well-typed and if $\Gamma_m;\Gamma_p \vdash x[\overline{\mu_i}] : \tau$ then $\Gamma_m;\Gamma_H \vdash Let\ H'\ in\ @^*zF : \tau$

Since we assume H is well typed, we need to prove H' is well typed. To prove H' is well typed we have to prove $\Gamma_H \vdash x : \forall\overline{t_i}.T(F) \rightarrow \mu$ which is true by the precondition on the translation. Therefore H' is well typed.

From the well-typedness of $H + H'$ we get $\Gamma_H \vdash x : \forall\overline{t_i}.T(F) \rightarrow \mu$. From this we get $\Gamma_{H+H'} \vdash z : T(F) \rightarrow [\mu_i/t_i]\mu$. Since F is the free variables of x, it cannot have t_i as a free type variable. Therefore, $\Gamma_m;\Gamma_{H+H'} \vdash @^*zF : [\mu_i/t_i]\mu$. Note that $T(F)$ is the type of the variables in F under the environment Γ_m which remains the same throughout the program since we assume that identifiers are unique. Also the type of z remains unchanged since Γ_m and H bind a disjoint set of variables. Therefore, the type preservation theorem follows.

Case tfn = To prove - given H is well-typed , then $H + H_1 + H_2$ is well-typed and if $\Gamma_m;\Gamma_p \vdash$ `let` $x = \Lambda\overline{t_i}.e_1$ `in` $e_2 : \mu_2$ then $\Gamma_m;\Gamma_H \vdash Let\ H_1 + H_2\ in\ e_2' : \mu_2$.

$$(const/var) \qquad \overline{a \vdash i \to i} \quad \overline{a \vdash x \to a(x)}$$

$$(fn) \qquad \overline{a \vdash \lambda x{:}\mu.e \to Clos\langle x^{mu}, e, a\rangle}$$

$$(app) \qquad \frac{a \vdash x_1 \to Clos\langle x^{\mu}, e, a'\rangle \quad a \vdash x_2 \to v' \quad a' + x \mapsto v' \vdash e \to v}{a \vdash @x_1 x_2 \to v}$$

$$(tfn) \qquad \overline{a \vdash \Lambda \overline{t_i}.e_v \mapsto Clos^t\langle \overline{t_i}, e_v, a\rangle}$$

$$(let) \qquad \frac{a \vdash e_1 \to v_1 \qquad a + x \mapsto v_1 \vdash e_2 \to v}{a \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \to v}$$

$$(tapp) \qquad \frac{a \vdash x \mapsto Clos^t\langle \overline{t_i}, e_v, a'\rangle \quad a' \vdash e_v[\mu_i/t_i] \to v}{a \vdash x[\overline{\mu_i}] \mapsto v}$$

Figure 2.6: Operational Semantics

By induction, if H is well-typed , then $H + H_1'$ is well-typed and $\Gamma_m; \Gamma_H \vdash Let\ H_1'\ in\ e_1' : \mu_1$. This implies that $\Gamma_m; \Gamma_H \vdash Let\ H_1'\ in\ \lambda^* F_1.e_1' : T(F_1) \to \mu_1$ where $T(F_1)$ is the type of the variables F_1 in Γ_m. Since we assume unique identifiers, these types always remain the same. That is, the variable name unambigously denotes a type and this denotation remains the same at all times in the program. Now since $\lambda^* F_1.e_1'$ is closed with respect to monomorphic variables, we no longer require the environment Γ_m. Therefore we get
$\Gamma_H \vdash Let\ H_1'\ in\ \lambda^* F_1.e_1' : T(F_1) \to \mu_1$. By definition, this implies H_1 is well-typed and therefore $H + H_1$ is well-typed.

Again by induction we have if $H + H_1$ is well-typed, then $H + H_1 + H_2$ is well-typed and if
$\Gamma_m; \Gamma_p + x \mapsto \langle \forall \overline{t_i}.\mu_1, F_1\rangle \vdash e_2 : \mu_2$ then $\Gamma_m; \Gamma_{H+H_1} \vdash Let\ H_2\ in\ e_2' : \mu_2$. From this we get that $\Gamma_m; \Gamma_{H+H_1+H_2} \vdash e_2' : \mu_2$ which leads easily to the type preservation theorem.

2.4.2 Semantic Soundness

Before proving the semantic soundness, we first give the operational semantics of our calculus in Fig 2.6.

There are only three kinds of values - integers, function closures and type function closures.

$$(values) \quad v \quad ::= \quad i \mid Clos\langle x^{\mu}, e, a\rangle \mid Clos^t\langle \overline{t_i}, e, a\rangle$$

Notation 3 *The relation $a : \Gamma \vdash e \to v$ means that in a value environment a respecting Γ, e evaluates to v. a respects Γ means that if $a(x) = v$ and $\Gamma(x) = \mu$, then $\Gamma \vdash v : \mu$.*

Notation 4 *The notation $a(x \mapsto ..)$ means that in the environment a, x has the given value. Whereas $a[x \mapsto ..]$ means that the environment a is augmented with the given binding. Continuing from above we get,*

Definition 2.4.10 (Type of a Value)

- $\Gamma \vdash i : int$

- if $\Gamma \vdash \lambda x{:}\mu.e : \mu \to \mu'$, then $\Gamma \vdash Clos\langle x^{\mu}, e, a\rangle : \mu \to \mu'$
- if $\Gamma \vdash \Lambda\overline{t_i}.e_v : \forall\overline{t_i}.\mu$, then $\Gamma \vdash Clos^t\langle\overline{t_i}, e_v, a\rangle : \forall\overline{t_i}.\mu$

Throughout the proofs we assume that the subject reduction lemma holds. That is if $a : \Gamma \vdash e \to v$ and $\Gamma \vdash e : \mu$, then $\Gamma \vdash v : \mu$. This lemma can be proved from the given operational semantics by structural induction on the syntactic structure of e.

Definition 2.4.11 (Equivalence of Values)

- **Equivalence of Int** — Suppose $\Gamma \vdash i : int$ and $\Gamma' \vdash i' : int$. Then $i \approx i'$ iff $i = i'$.
- **Equivalence of Closures**
  - Suppose $\Gamma \vdash Clos\langle x^{\mu}, e, a\rangle : \mu \to \mu'$ and $\Gamma' \vdash Clos\langle x^{\mu}, e', a'\rangle : \mu \to \mu'$.
  - Suppose further that $\Gamma \vdash v_1 : \mu$ and $\Gamma' \vdash v_1' : \mu$ and $v_1 \approx v_1'$.

  Then $Clos\langle x^{\mu}, e, a\rangle \approx Clos\langle x^{\mu}, e', a'\rangle$ iff $\quad \forall v_1, v_1' \quad a : \Gamma + x \mapsto v_1 \vdash e \to v$ and $a' : \Gamma' + x \mapsto v_1' \vdash e' \to v'$ and $v \approx v'$
- **Equivalence of Type Closures** Suppose $\Gamma \vdash Clos^t\langle\overline{t_i}, e_v, a\rangle : \forall\overline{t_i}.\mu$ and $\Gamma' \vdash Clos^t\langle\overline{t_i}, e_v', a'\rangle : \forall\overline{t_i}.\mu$. Then $Clos^t\langle\overline{t_i}, e_v, a\rangle \approx Clos^t\langle\overline{t_i}, e_v', a'\rangle$ iff $a : \Gamma \vdash e_v[\mu_i/t_i] \to v$ and $a' : \Gamma' \vdash e_v'[\mu_i/t_i] \to v'$ and $v \approx v'$.

Definition 2.4.12 (Equivalence of terms) *Suppose $a : \Gamma \vdash e \to v$ and $a' : \Gamma' \vdash e' \to v'$ with $\Gamma \vdash e : \mu$ and $\Gamma' \vdash e' : \mu$. If $v \approx v'$, then we say that the terms e and e' are semantically equivalent and denote this by $a : \Gamma \vdash e \approx a' : \Gamma' \vdash e'$.*

Lemma 2.4.13 *if $a : \Gamma \vdash e \approx a' : \Gamma' \vdash e'$, then $a : \Gamma \vdash e[\mu_i/t_i] \approx a' : \Gamma' \vdash e'[\mu_i/t_i]$*

Before we get into the proof, we want to define a couple of predicates on the header - a_H and well-formedness of H. Intuitively a_H represents the addition of new variable-value bindings in the environment as the header gets evaluated. Well-formedness of the header will ensure that the lifting of the type function and the type application does not change the semantics of the program. We will semantically equate the old unlifted type application and the lifted type application applied to the free variables of the definition, both of them in the same environment.

Definition 2.4.14 (Let H in e)

Suppose $H = h_1 ... h_n$. Then **Let H in e** is shorthand for *let* h_1 *...* *in let* h_n *in* e. If $h_j ::= (x = e, \tau)$, then *let* h_j is shorthand for let x = e. The evaluation rule is $a_m : \Gamma_m \vdash Let\ H\ in\ e \approx a_m : \Gamma_m; a_H : \Gamma_H \vdash e$. Note that since we assume subject reduction holds a_H always respects Γ_H.

Definition 2.4.15 (a_H)

a_H is equal to $a_{h_0 ... h_n}$ and a_{h_j} is —

- $h_j ::= (x = \Lambda\overline{t_i}.e, \tau)$ then
  $a_{h_j} := x \mapsto Clos^t\langle\overline{t_i}, e, a_{h_0 ... h_{j-1}}\rangle$
- $h_k ::= (z = x[\overline{\mu_i}], \tau)$ and $h_j ::= x \mapsto Clos^t\langle\overline{t_i}, e, a_h\rangle$ $j < k$, then $a_{h_k} := z \mapsto v$ where
  $a_h : \Gamma_h \vdash e[\mu_i/t_i] \to v$

Definition 2.4.16 (H is well-formed w.r.t $a_m : \Gamma_m; a_p : \Gamma_p$)

H is well-formed w.r.t. $a_m : \Gamma_m; a_p : \Gamma_p$, if $h_0, \ldots, h_n$ are well-formed. A header entry h_j is well-formed if all its predecessors $h_0, \ldots, h_{j-1}$ are well-formed, and furthermore,

- If $h_j ::= (x = \Lambda\overline{t_i}.e, \tau)$, then h_j is well-formed w.r.t $a_m : \Gamma_m; a_p : \Gamma_p$ iff suppose $\Gamma_p(x) = (\forall\overline{t_i}.\mu, F)$ and

$$a_m : \Gamma_m; a_p : \Gamma_p \vdash x[\overline{\mu_i}] \rightarrow v \text{ and}$$
$$a_m : \Gamma_m; a_{h_0 \ldots h_j} : \Gamma_{h_0 \ldots h_j} \vdash let\ \ z = x[\overline{\mu_i}]\ \ in\ \ @^* zF \rightarrow v'$$

  then $v \approx v'$ is true.

- If $h_j ::= (z = x[\overline{\mu_i}], \tau)$, then h_j is well-formed.

We will sometimes abbreviate — H is well-formed w.r.t. $a_m : \Gamma_m; a_p : \Gamma_p$ —by simply writing H is well-formed.

Theorem 2.4.17 (Semantic Soundness) *Suppose we have* $\Gamma_m; \Gamma_p; H \vdash e : \mu \Rightarrow e'; H_1; F$ *and* $a_m : \Gamma_m; a_p : \Gamma_p \vdash e \rightarrow v$. *If* H *is well-formed with respect to* $a_m : \Gamma_m; a_p : \Gamma_p$ *and* $a_m : \Gamma_m; a_H : \Gamma_H \vdash Let\ H_1\ in\ e' \rightarrow v'$ *then* $v \approx v'$.

Proof. The proof is by induction on the structure of e. The only interesting cases are *tapp* and tfn.

Case tapp = To prove — $a_m : \Gamma_m; a_p : \Gamma_p \vdash x[\overline{\mu_i}] \rightarrow v$ and H is well-formed with respect to $a_m : \Gamma_m; a_p : \Gamma_p$ then if $a_m : \Gamma_m; a_H : \Gamma_H \vdash Let\ H_1\ in\ @^* zF \rightarrow v'$ implies $v \approx v'$.

Substituting the value of $Let\ H_1$ the required equivalence gets converted to — given H is well-formed $a_m : \Gamma_m; a_p : \Gamma_p \vdash x[\overline{\mu_i}] \rightarrow v$ iff $a_m : \Gamma_m; a_H : \Gamma_H \vdash \texttt{let}\ z = x[\overline{\mu_i}]\ \texttt{in}\ @^* zF \rightarrow v'$ and $v \approx v'$.

By the precondition on the translation rule $\Gamma_p(x) = (\forall\overline{t_i}.\mu, F)$ and since Γ_H also binds x to a polymorphic type, there exists some $h_j \in H$ such that $h_j ::= (x = \Lambda\overline{t_i}.e, \tau)$

Since H is well-formed, h_j is well-formed as well and so we get —

$$a_m : \Gamma_m; a_p : \Gamma_p \vdash x[\overline{\mu_i}] \rightarrow v \text{ and}$$
$$a_m : \Gamma_m; a_{h_0 \ldots h_j} : \Gamma_{h_0 \ldots h_j} \vdash let\ \ z = x[\overline{\mu_i}]\ \ in\ \ @^* zF \rightarrow v'$$

then $v \approx v'$ is true.

But we assume that variables are bound only once. Therefore there exists no other $h_k \in H$ such that h_k binds x and specifically $h_{j+1}, ..., h_n$ do not affect x. Therefore the definition of well-formedness can be reduced to

$$a_m : \Gamma_m; a_p : \Gamma_p \vdash x[\overline{\mu_i}] \rightarrow v \text{ and}$$
$$a_m : \Gamma_m; a_H : \Gamma_H \vdash let\ \ z = x[\overline{\mu_i}]\ \ in\ \ @^* zF \rightarrow v'$$

then $v \approx v'$ is true.

which is what we want to prove.

Case tfn = To prove - given H is well-formed and also that $a_m : \Gamma_m; a_p : \Gamma_p \vdash \texttt{let}\ x = \Lambda\overline{t_i}.e_1\ \texttt{in}\ e_2 \rightarrow v$ then if $a_m : \Gamma_m; a_H : \Gamma_H \vdash Let\ H_1 + H_2\ in\ e_2' \rightarrow v'$ implies that $v \approx v'$.

By induction we have that — given H is well-formed and if $a_m : \Gamma_m; a_p : \Gamma_p \vdash e_1 \rightarrow v_1$ and $a_m : \Gamma_m; a_H : \Gamma_H \vdash Let\ H_1'\ in\ e_1' \rightarrow v_1'$ then $v_1 \approx v_1'$.

Assume for the time being that $H + H_1$ is well-formed. Then if $a_m : \Gamma_m; a_p[x \mapsto Clos^t\langle\overline{t_i}, e_1, a_m + a_p\rangle] : \Gamma_p[x \mapsto \langle\forall\overline{t_i}.\mu_1, F\rangle] \vdash e_2 \rightarrow v_2$ and $a_m : \Gamma_m; a_{H+H_1} : \Gamma_{H+H_1} \vdash Let\ H_2\ in\ e_2' \rightarrow v_2'$ implies that $v_2 \approx v_2'$.

But from the operational semantics we can easily deduce that $v_2 = v$ and that $v_2' = v'$. This therefore leads to the semantic soundness theorem.

We are therefore left with proving that $H + H_1$ is well-formed. By assumption, H is well-formed, therefore we must prove that H_1 is well-formed. From the definition, equating h_j to H_1 we need to prove that

$a'_m : \Gamma'_m; a'_p : \Gamma'_p \vdash x[\overline{\mu_i}] \rightarrow v$ and
$a'_m : \Gamma'_m; a_{H+H_1} : \Gamma_{H+H_1} \vdash let\ \ z = x[\overline{\mu_i}]\ \ in\ \ @^* zF \rightarrow v'$

then $v \approx v'$ is true.

But from the operational semantics we get that in the untranslated expression $x \mapsto Clos^t\langle \overline{t_i}, e_1, a_m + a_p \rangle$ and therefore

$$a'_m : \Gamma'_m; a'_p : \Gamma'_p \vdash x[\overline{\mu_i}] \approx a_m : \Gamma_m; a_p : \Gamma_p \vdash e_1[\mu_i/t_i]$$

Again by definition,
$a_{H_1} := x \mapsto Clos^t\langle \overline{t_i}, Let\ H'_1\ in\ \lambda^* F.e'_1, a_H \rangle$.

Therefore $z \mapsto Clos\langle F^{T(F)}, e'_1[\mu_i/t_i], a_H + a_{H'_1[\mu_i/t_i]} \rangle$ (skipping a couple of steps) and so we get –

$$a'_m : \Gamma'_m; a_{H+H_1} : \Gamma_{H+H_1} \vdash let\ \ z = x[\overline{\mu_i}]\ \ in\ \ @^* zF \approx a'_m(F) : \Gamma'_m; a_H : \Gamma_H + a_{H'_1[\mu_i/t_i]} : \Gamma_{H'_1[\mu_i/t_i]} \vdash e'_1[\mu_i/t_i]$$

So the equivalence reduces to

$$a_m : \Gamma_m; a_p : \Gamma_p \vdash e_1[\mu_i/t_i] \approx a'_m(F) : \Gamma'_m; a_H : \Gamma_H + a_{H'_1[\mu_i/t_i]} : \Gamma_{H'_1[\mu_i/t_i]} \vdash e'_1[\mu_i/t_i]$$

But $a'_m(F) = a_m(F)$ since variables are unique. Moreover since F consists of all the free variables of e'_1 that are bound in a'_m and by extension in a_m, evaluating e'_1 in $a_m(F)$ and in a_m is equivalent. So the equivalence simplifies further to

$$a_m : \Gamma_m; a_p : \Gamma_p \vdash e_1[\mu_i/t_i] \approx a_m : \Gamma_m; a_H : \Gamma_H + a_{H'_1[\mu_i/t_i]} : \Gamma_{H'_1[\mu_i/t_i]} \vdash e'_1[\mu_i/t_i]$$

But the above equivalence follows from the inductive assumption on the translation of e_1 and by applying Lemma 2.4.13 to it. This proves that H_1 is well-formed.

2.5 The Lifting Algorithm for FLINT

Till now, we have only considered the Core-ML calculus while discussing the algorithm. But what happens when we take into account the module language as well? Does the algorithm extend to the full language in an obvious way?

To compile the Full-ML langauge, we compile the source code into the FLINT intermediate language. The details of the translation to FLINT are given in [45]. In ML, structures are the basic module unit and functors are parameterised structures. Polymorphic functions may escape as part of structures and be initialized later at a functor application site. We will not get into the details of the ML module calculus, but what is of relevance here is that the module language does not fit into the constraints of the Core-ML calculus. (which is why we compile it to FLINT). Thus, to model functors, abstractions are now allowed to be polymorphic. Therefore type applications involving abstracted variables cannot be lifted above the abstraction binding the variable. Moreover, type applications may be curried since we may have escaping polymorphic functions. Since all the application sites of an escaping polymorphic function cannot be determined statically, we cannot arrange to pass in the free variables at each function call. Therefore an escaping polymorphic function with free variables in its definition cannot be lifted to the top level and hence its type applications also cannot be lifted to the top level. As a result, the algorithm cannot be extended to FLINT in a straightforward manner.

But there is a silver lining to the dark clouds. Functors in a program always occur outside any Core-ML functions. Furthermore, only functor parameters give rise to partial type applications. Therefore, we can lift the partial type applications to where the functor parameter is bound and hence outside all Core-ML functions. We will formalise these constraints in terms of the FLINT calculus later in this section.

The core language of FLINT is based upon a predicative variant of the Girard-Reynolds polymorphic λ-calculus F_ω [9, 41], with the term language written in A-normal form [7]. It contains the following four syntactic

classes: kinds (κ), constructors (μ), types (σ), terms (e), as shown in Figure 2.7. Here, kinds classify 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 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 terms are an explicitly typed λ-calculus (but in A-normal form) with explicit constructor abstraction and application forms.

$$
\begin{array}{llll}
(\textit{kinds}) & \kappa & ::= & \Omega \mid \kappa_1 \rightarrow \kappa_2 \\
(\textit{constructors}) & \mu & ::= & t \mid \texttt{Int} \mid \mu_1 \rightarrow \mu_2 \mid \lambda t{::}\kappa.\mu \mid \mu_1[\mu_2] \\
(\textit{types}) & \sigma & ::= & T(\mu) \mid \sigma_1 \rightarrow \sigma_2 \mid \forall t{::}\kappa.\sigma \\
(\textit{terms}) & e & ::= & i \mid x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \mid @x_1 x_2 \\
 & & & \mid fix[x_i : \sigma_i, e_i] \\
 & & & \mid \texttt{let } x = \Lambda\overline{t_i} :: k_i.e_v \texttt{ in } e_2 \mid x[\mu_i] \\
(\textit{values}) & e_v & ::= & i \mid x \mid \texttt{let } x = e_v \texttt{ in } e'_v \mid fix[x_i : \sigma_i, e_i] \\
 & & & \texttt{let } x = \Lambda\overline{t_i} :: k_i.e_v \texttt{ in } e'_v \mid x[\mu_i]
\end{array}
$$

Figure 2.7: Syntax of the Core-FLINT calculus

In the FLINT calculus, abstractions model both functors and functions. However the type of the abstracted variable is used to distinguish between functors (polymorphic) and functions (monomorphic). In a translated program, this implies that polymorphic abstractions are never nested inside a monomorphic abstraction since functors are never nested inside functions. Moreover, since partial type applications can involve only polymorphic variables, monomorphic abstractions cannot prevent their lifting.

Therefore with a preprocessing phase, any input FLINT program can be converted into a *well-formed* program which is defined to satisfy the following constraints

- No polymorphic abstraction is nested inside a monomorphic abstraction.
- No partial type application is nested inside a monomorphic abstraction.

We now redefine the depth of a term in a program as the number of function abstractions (monomorphic abstractions) within which it is nested with *depth 0 terms being the ones outside all function abstractions.* Note that depth 0 terms now may not occur outside all abstractions since they may be nested inside polymorphic abstractions (functors in the source language). We then perform type lifting as in Fig 2.2 for terms at depth greater than zero and lift the polymorphic definitions and type applications to depth 0. For terms already at depth zero, the translation is just the identity translation and the header returned is empty.

We present the algorithm formally in Fig 2.8. The translation rules are expressed as sequents of the form

$$\Gamma; d; td \vdash e \Rightarrow e'; H; F$$

d refers to the number of λ abstractions within which we are nested. td refers to the number of Λ abstractions within which we are nested. e is the input term and e' is the output term. H is the header as defined before – it contains the list of type expressions in e that must be lifted to depth 0. F as in the previous algorithm is the list of free variables of e'.

Γ is the type environment that maps a variable to its type, the depth (no. of λ) at which it was defined, type depth (no. of Λ) at which it was defined, and the free variables in its definition. Note that the free variable information is irrelevant in the case of monomorphic variables.

The final result of lifting a closed term e of type μ for which the algorithm infers $\emptyset; 0; 0 \vdash e : \mu \Rightarrow e'; H; \emptyset$ is $LET(H, e')$, where the function $LET(H, e)$ expands a list of bindings $H = [\langle x_1, e_1\rangle, \ldots, \langle x_n, e_n\rangle]$ and a term e into the term $\texttt{let } x_1 = e_1 \texttt{ in } \ldots \texttt{let } x_n = e_n \texttt{ in } e$.

We will briefly explain the algorithm here. Firstly note that the second *tapp* rule deals with partial type applications and the fct rule deals with functors. In the fn rule, we deal only with non-recursive functions – the general case follows easily. At depth $d = 0$, no lifting takes place and consequently the header returned is always nil. Therefore in the second tfn rule and the fct rule, the headers from the translation are empty lists. The case for full type application is similar to the Core-ML case. In the case of partial type applications, we need to know the remaining type parameters before we can pass in the free variables of the polymorphic function. This results in the simultaneous introduction of the polymorphic abstraction and the type application. In case we are at depth 0, the transformation is just the identity function since no lifting takes place at $d = 0$. In the second tfn rule, the polymorphic function need not be lifted since it is already at the top level. Therefore the translation is just a combination of the translations for the subterms. Moreover since we are already at depth 0, the result header is also empty.

The rule for functors is slightly different. We want to ensure that all type applications are nested only inside functors. We do not want to lift them outside functors. Therefore while translating the body of the functor, we reinitialize the depth to zero. This ensures that all type applications inside the functor body will be lifted outside functions and nested only inside the functor. And since we set the depth to 0, the resulting header from the translation is empty. The case for functions is similar in essence to the Core-ML case. We process the function body and on returning if we find that we are at depth 0, then we dump the result header. (The **LET** function is defined above). Otherwise the translation continues in the normal way.

We illustrate the lifting algorithm on the example code shown below. The syntax is not totally faithful to the FLINT syntax shown before but it makes the code clearer.

```
Λt0.λX1:S.
 y = #1(X1)
 f = λv.
       let id = Λt1.λx2.x2
           .....
           v1 = ... id[Int](3) ....
       in  v1
 v2 = y[t0]f
```

which gets translated to

```
Λt0.λX1:S.
 y = #1(X1)
 f = let
       id = Λt1.λx2.x2
       z1 = id[Int]
       .. (Other type expressions in f's body)..
     in  λv.
           let  ..... (type lifted body of f)
                v1 = ... z1(3) ....
           in   v1
 v2 = y[t0]f
```

In the code above, the parameter X_1 is a higher-order functor whose first component is another functor. The type S denotes a structure type. Suppose f is just a single-element structure. As we explained above, y, f and v_2 are at depth 0 even though they are nested inside the functor abstraction(λX_1). This also means that the type application $y[t_0]$ is at depth 0 and therefore we will not attempt to lift it. It is only inside the function f that the depth increases. The algorithm dumps all the type applications just outside the function abstraction (λv), they are not lifted outside the functor abstraction (λX_1).

Is the reformulation above merely an artifice to get around the problems posed by FLINT ? No, the main aim of the type lifting transformation is to perform all the type applications during "link" time—when the top level code is being executed—and eliminate runtime type passing inside functions. Functors are top level code

as well and are applied at "link" time. Moreover they are non-recursive and do not occur inside loops. Therefore having type applications nested only inside functors still results in the type applications being performed once and for all at the beginning of program execution. As a result, we still eliminate runtime type passing inside functions.

In passing, we note that depth 0 in Core-ML (according to the definition above) coincides with the top level of the program since we do not have functors and hence polymorphic abstractions in Core-ML; therefore the Core-ML translation is merely a special case of the translation for FLINT.

2.6 Implementation Results

We would first like to discuss a few performance issues before we actually examine the runtime figures.

Our algorithm lifts type applications to the top level and this makes the simultaneous uncurrying of both value and type applications difficult. At runtime, type applications result in the formation of closures. But all of these closures are created only at the top level and are never created repeatedly. We therefore believe that this is not a significant penalty. Related to this is the cost of function application because this involves selecting the environment and the code from the closure before the function can actually be applied. However, in most cases, the selection of the code and the environment will be a loop invariant expression and can therefore be optimised. Secondly we need to address the issue of closure size of the lifted functions. Our optimal type lifting does not introduce any free type variables. And since the body of the function after lifting does not use the type variables any more these type variables do not need to be included in the closure. However the *tapp* rule in our algorithm introduces new variables*(the Set L)* which may increase the number of free variables of the function body. Moreover local polymorphic definitions are now lifted from function bodies which also increases the closure size.

We speculate that the increase in closure size, if any, and hence in the closure creation time does not incur a significant runtime penalty. This is borne out by the results on the benchmark suite. None of the benchmarks slows down significantly - some of the benchmarks show a moderate speedup. Note that the type information currently maintained in the FLINT compiler is very minimal. Types are represented by integers and the type information is just sufficient to distinguish between integers, reals and records. As a result, presently type construction and type application are not expensive. However we intend to make the type representation more sophisticated in the near future. The main motivating goal of the transformation is to ensure that in the presence of a more complicated type representation we do not incur a significant runtime penalty - yet at the same time make use of the enhanced type information at runtime.

We have implemented the type-lifting algorithm in the FLINT/ML compiler version 1.0. All the tests were performed on a Pentium Pro 200 Linux workstation with 64M physical RAM.

The algorithm is implemented in a single pass by a bottom up traversal of the syntax tree. An earlier stage of the compiler performs type specialization. This phase also checks for duplicate type applications and performs "common type-application elimination". We use *de Bruijn* notations [5, 30] to represent types. But the type information to be manipulated is kept to a minimum by the algorithm. In Figure 2.2 Rule (*tfn*), when we lift polymorphic function definitions, we dump all the expressions in H_1 in front of the type abstraction even though we need only dump those terms (in H_1) whose set of free type variables contain any of the $t_i's$. The advantage of dumping all the expressions at that point is that the *de Bruijn* depth of the terms in H_1 does not change. Hence we do not have to change the de Bruijn indices of the type expressions in H_1. The only time we need to manipulate the type information is when we abstract the free variables of a polymorphic definition—we need to adjust the de Bruijn indices of the types associated with the variables bound by the newly introduced abstractions. The type environment used in the implementation also remembers the depth (defined in Section 2.5) at which a variable was defined. This is used to ensure that the number of variables abstracted when a polymorphic definition is closed is kept to a minimum; variables that will still remain in scope after the lifting are not abstracted.

$$(exp)\qquad \frac{\Gamma(x) = (\mu, \_, \_, \_)}{\Gamma; d; td \vdash x \Rightarrow x; \emptyset; [x : \mu]} \qquad \Gamma; d; td \vdash i \Rightarrow i; \emptyset; \emptyset$$

$$(app)\qquad \frac{\Gamma(x_1) = \langle \mu_1 \to \mu_2, \_, \_, \_\rangle \qquad \Gamma(x_2) = \langle \mu_1, \_, \_, \_\rangle}{\Gamma; d; td \vdash @x_1 x_2 \Rightarrow @x_1 x_2; \emptyset; [x_1 : \mu_1 \to \mu_2, x_2 : \mu_1]}$$

$$(let)\qquad \frac{\Gamma; d; td \vdash e_1 \Rightarrow e_1'; H_1; F_1 \quad \Gamma[x \mapsto \langle \mu_1, d, td, F_1\rangle]; d; td \vdash e_2 \Rightarrow e_2'; H_2; F_2}{\Gamma; d; td \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Rightarrow \texttt{let } x = e_1' \texttt{ in } e_2'; H_1 || H_2; F_1 \cup (F_2 \setminus [x : \mu_1])}$$

$$(tap)\qquad \frac{\Gamma(x) = \langle \forall \overline{t_i}.\mu, \_, \_, L\rangle \qquad v \text{ fresh variable}}{\begin{array}{c}\Gamma; d > 0; td \vdash x[\overline{\mu_i}] \Rightarrow @vL; (v, x[\overline{\mu_i}]); L \\ \Gamma; d = 0; td \vdash x[\overline{\mu_i}] \Rightarrow x[\overline{\mu_i}]; nil; nil\end{array}}$$

$$(tap)\qquad \frac{\Gamma(x) = \langle \forall \overline{t_i}.\mu, \_, \_, L\rangle \qquad v \text{ fresh variable}}{\begin{array}{c}\Gamma; d > 0; td \vdash x[\overline{\mu_i}] \Rightarrow \Lambda\overline{t_i} :: k_i.@(v[\overline{t_i}])L; (v, x[\overline{\mu_i}]); L \\ \Gamma; d = 0; td \vdash x[\overline{\mu_i}] \Rightarrow x[\overline{\mu_i}]; nil; nil\end{array}}$$

$$(tfn)\qquad \frac{\begin{array}{c}\Gamma; d; td + 1 \vdash e_1 \Rightarrow e_1'; H_1; F_1 \\ \Gamma[x \mapsto \langle \forall \overline{t_i}.\mu, d, td, F_1\rangle]; d; td \vdash e_2 \Rightarrow e_2'; H_2; F_2\end{array}}{\begin{array}{c}\Gamma; d > 0; td \vdash \texttt{let } x = \Lambda\overline{t_i} :: k_i.e_1 \texttt{ in } e_2 \Rightarrow e_2'; exp :: H_2; F_2 \\ exp = let\ x = \Lambda\overline{t_i} :: k_i.LET(H_1, fix[\overline{F_1}, e_1'])\end{array}}$$

$$(tfn)\qquad \frac{\begin{array}{c}\Gamma; d; td + 1 \vdash e_1 \Rightarrow e_1'; nil; F_1 \\ \Gamma[x \mapsto \langle \forall \overline{t_i}.\mu, d, td, F_1\rangle]; d; td \vdash e_2 \Rightarrow e_2'; nil; F_2\end{array}}{\Gamma; d = 0; td \vdash \texttt{let } x = \Lambda\overline{t_i} :: k_i.e_1 \texttt{ in } e_2 \Rightarrow \texttt{let } x = \Lambda\overline{t_i} :: k_i.e_1' \texttt{ in } e_2'; nil; F_1 \cup F_2}$$

$$(fct)\qquad \frac{\Gamma[x \mapsto \langle \sigma, 0, td, \_\rangle]; d = 0; td \vdash e \Rightarrow e'; nil; F}{\Gamma; d; td \vdash fix[x : \sigma, e] \Rightarrow fix[x : \sigma, e']; nil; F_1 \setminus [x : \sigma]}$$

$$(fn)\qquad \frac{\Gamma[x \mapsto \langle \mu, d + 1, td, \_\rangle]; d + 1; td \vdash e \Rightarrow e'; H; F}{\begin{array}{c}\Gamma; d > 0; td \vdash fix[x : \mu, e] \Rightarrow fix[x : \mu, e']; H; F \setminus [x : \mu] \\ \Gamma; d = 0; td \vdash fix[x : \mu, e] \Rightarrow LET(H, (fix[x : \mu, e'])); nil; F \setminus [x : \mu]\end{array}}$$

Figure 2.8: The Lifting Translation for FLINT

Benchmark	Description	New Time	Old Time	Ratio
Simple	A spherical fluid-dynamics program	7.04	9.78	0.72
Vliw	A VLIW instruction scheduler	4.22	4.31	0.98
lexgen	A lexical-analyzer generator	2.38	2.36	1.01
ML-Yacc	The ML-yacc	1.05	1.11	0.95
Mandelbrot	The Mandelbrot curve construction	4.62	4.62	1.0
Kb-comp	Knuth-Benedix Completion Algorithm	2.98	3.11	0.96
Ray	A ray-tracer	10.68	10.66	1.01
Life	Runs 10,000 generations of the Life Simulation	2.80	2.80	1.0
Boyer	A simple theorem prover	0.49	0.52	0.96

Figure 2.9: Type Lifting Results

Our algorithm is a source-to-source transformation and the output from it is again a FLINT program. We do not need any auxiliary type system to type-check the transformation, the FLINT type-checker suffices which is a big gain. This helped us immensely in implementing the algorithm and fixing the bugs that cropped up during the implementation.

Figure 2.9 shows CPU times for executing the Standard ML benchmark suite with type lifting turned on and turned off. The third column (New Time) indicates the execution time with lifting turned on and the next column (Old Time) indicates the execution time with lifting turned off. The last column gives the ratio of the new time to the old time. We get moderate speedups for some of the benchmarks and a good speedup for one benchmark—an average of about 5% for the polymorphic benchmarks. Simple has a lot of polymorphic function calls occuring inside loops and therefore benefits greatly from lifting. Boyer and mandelbrot are monomorphic benchmarks (involving large lists) and predictably do not benefit from the optimization. Even though life is a heavily polymorphic benchmark, most of its time is spent in the polymorphic equality function [47]. Type lifting does not obviate this and hence the speedup is negligible.

2.7 Related Work and Conclusions

Tolmach [50] has worked on a similar problem and proposed a method based on the lazy substitution on types. He used the method in the implementation of the tag-free garbage collector. Minamide [27] has also attacked the same problem but has used an entirely different approach from ours. He proposes a refinement of Tolmach's method to eliminate runtime construction of type parameters. We have elaborated on this difference in Section 2.3. The speedups obtained in our method are comparable to the ones reported in his paper. Mark P. Jones [18] has worked on the related problem of optimising dictionary passing in the implementation of type classes. We elaborated on this in Section 2.3.

In their study of the type theory of Standard ML, Harper and Mitchell [11] argued that an explicitly typed interpretation of ML polymorphism has better semantic properties and scales more easily to cover the full language. The idea of passing types to polymorphic functions is exploited by Morrison *et al.* [29] in the implementation of Napier. The work of Ohori on compiling record operations [33] is similarly based on a type passing interpretation of polymorphism. Jones [19] has proposed *evidence passing*—a general framework for passing data derived from types to "qualified" polymorphic operations. Harper and Morisett [13] proposed an alternative approach for compiling polymorphism where types are passed as arguments to polymorphic routines in order to determine the representation of an object. The boxing interpretation of polymorphism which applies the appropriate coercions based on the type of an object was studied by Leroy [21] and Shao [43]. Many modern compilers like the FLINT/ML compiler [44], TIL [49] and the Glasgow Haskell compiler [34] use an explicitly typed language as the intermediate language for the compilation.

Lambda lifting and full laziness are part of the folklore of functional programming. Hughes [16] showed that by doing lambda lifting in a particular way, full laziness can be preserved. Johnsson [17] describes different forms of lambda lifting and the pros and cons of each. Peyton Jones [38, 35, 37] also described a number of optimizations which have similar spirits but have totally different aims. Appel [3] describes let hoisting in the context of ML. In general, using correctness preserving transformations as a compiler optimization [1, 3] is a well established technique and has received quite a bit of attention in the functional programming area.

We have proposed a method for minimizing the cost of runtime type passing. Our algorithm lifts all type applications out of functions and therefore eliminates the construction of types inside functions at runtime. The amount of type information constructed at run time is a static constant. We can guarantee that in Core-ML programs, all type applications will be lifted to the top level.

Chapter 3

Common Type Expression Elimination

3.1 Introduction

In the last chapter, we presented a method of eliminating runtime type constructions and ensuring that all type information was resolved once and for all at linktime. This ensures that the cost of manipulating types at runtime never blows up. However, while constructing types during linktime, we want to ensure that the sharing between the types that is maintained during compilation is also preserved during linktime – we want to make sure that we expend as little effort as possible on constructing types. More formally, we want to ensure that if the compilation of a program resulted in the creation of k types, then during linktime we construct $\Theta(k)$ types. We know from our experience in compiling that preserving the sharing between types is critical to compilation time which leads us to believe that even if we were to construct types only at the beginning, we could still incur a significant overhead if we don't preserve the sharing. Therefore eliminating common type expressions was essential in ensuring efficient runtime type passing.

We had several options about how to proceed. One was to reify the types and then apply common subexpression elimination to the resultant code. This would involve no extra effort since both the phases already exist in our compiler. However, this would imply that there would be one stage in the compiler when the type information would blow up which was unacceptable. *The FLINT/ML compiler guarantees that all type preserving stages – including the execution phase if types are passed at runtime – will preserve the asymptotic time and space usage in representing and manipulating types.* This condition would have been violated if we had followed the above approach.

The other option was to perform the common type expression elimination as part of the reify stage – in some ways combine the type expression elimination and the reification into a single stage. We did not favour this approach in the interest of modularity. If we changed the runtime type representation at some later point and therefore needed to modify the reification stage, we might have had to redo some of the elimination code. We believed it was better to have it as a separate stage in the compiler.

The fact that we use Debruijn indices to represent types made the algorithm and the implementation non-trivial. FLINT does not support an explicit **lettype** construct by which we may bind type expressions to type variables. We therefore used a combination of a polymorphic definition and a type application for the same purpose. So for example, the following code *lettype* $t = \mu$ *in exp end* is represented as $(\Lambda t.e)[\mu]$ and a sequence of such lettypes is represented as a nested sequence of polymorphic abstractions and type applications. This was sufficient for our purpose since in FLINT only constructors are passed at runtime – therefore we are actually interested in eliminating common constructor expressions only. But the introduction of polymorphic abstractions meant that the Debruijn index of terms was going to change and was going to change in a pretty drastic and random manner which in turn made the implementation pretty hairy. We talk more about this in the implementation section.

The next section(Fig 3.2) describes the algorithm formally. Here we will just informally explain it. The only non-trivial cases are the type application and the polymorphic abstraction. In the type application, we look up the constructor in the environment. The function **I** first checks to see whether μ exists in the environment. If it does, it then returns the type variable bound to it. Otherwise, **I** finds out all the common type expressions inside μ and replaces them with the corresponding type variables. It then enters μ into the environment and binds a newly introduced type variable to it. In the code, μ is then replaced with this variable. When we encounter a polymorphic abstraction, we add a new layer to the environment. This layer will hold all the type expressions that involve the type variable t_i but none of the type variables introduced by abstractions inside e. When we return, we pop off this layer and bind the type expressions contained in this layer to the corresponding type variables.

The question that arises naturally at this point is – why wasn't common type expression elimination performed till now in the FLINT/ML compiler? The reason being that till now types were only represented as small integers at runtime. Therefore passing types at runtime was never costly. However as we shall see in chapter 4, we now intend to make the type representation in FLINT more sophisticated and therefore the cost of constructing types now becomes an issue.

3.2 The CTE Algorithm

In this section, we present the algorithm formally and prove the type preservation and the semantic-soundness theorems. We also show that every common type expression is eliminated.

3.2.1 Formal Description

The source language for the algorithm is shown in Fig 3.1 and is the standard predicative variant of the Girard-Reynolds polymorphic λ-calculus F_ω.

$$
\begin{array}{llll}
(\textit{kinds}) & \kappa & ::= & \Omega \mid \kappa_1 \rightarrow \kappa_2 \\
(\textit{constructors}) & \mu & ::= & t \mid \texttt{Int} \mid \mu_1 \rightarrow \mu_2 \mid \lambda t{::}\kappa.\mu \mid \mu_1[\mu_2] \\
(\textit{types}) & \sigma & ::= & T(\mu) \mid \sigma_1 \rightarrow \sigma_2 \mid \forall t{::}\kappa.\sigma \\
(\textit{terms}) & e & ::= & i \mid x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \mid \lambda x{:}\mu.e \mid @x_1 x_2 \mid \Lambda t{::}\kappa.e \mid x[\mu]
\end{array}
$$

Figure 3.1: Syntax of the Core-FLINT calculus

Figure 3.2 shows the common type-expression elimination algorithm. The translation is defined as a relation of the form $H; \triangle; \Gamma \vdash e \Rightarrow e'; H'$, that carries the meaning that the translation of the input term e is the term e', H is the type expression environment at the beginning of the translation, H' is the environment after the translation. Γ is the type environment mapping variables to types and $\triangle$ is the kind environment mapping type variables to their kinds. The type expression environment H is actually a list of environments $h_1 \ldots h_n$. Each h_i is added as we enter a new type abstraction Λt_i and is popped off as we leave the abstraction. Each h_i in turn is a mapping between newly introduced type variables and the type expressions that they denote. All the type expressions contained in a particular h_i involve the type variable t_i – that is the type variable introduced by the corresponding Λ abstraction – and do not include any type variable introduced by a Λ abstraction nested inside it.

In the presentation of the algorithm and later on in the proofs, we will often ignore the fact that H is actually a two dimensional environment and will instead linearise it. This is only to simplify the presentation. We will also use a few more notational shortcuts to keep the presentation uncluttered and we will state them as we proceed.

$$(exp) \quad H;\triangle;\Gamma \vdash i \Rightarrow i;H \qquad H;\triangle;\Gamma \vdash x \Rightarrow x;H \qquad H;\triangle;\Gamma \vdash @x_1x_2 \Rightarrow @x_1x_2;H$$

$$(abs) \quad \frac{H;\triangle;\Gamma[x:\mu] \vdash e \Rightarrow e';H'}{H;\triangle;\Gamma \vdash \lambda x{:}\mu.e \Rightarrow \lambda x{:}\mu.e';H'}$$

$$(let) \quad \frac{H;\triangle;\Gamma \vdash e_1 \Rightarrow e_1';H' \qquad H';\triangle;\Gamma[x:\mu] \vdash e_2 \Rightarrow e_2';H''}{H;\triangle;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2';H''}$$

$$(tap) \quad \frac{x:\forall t.\mu' \qquad I(\mu) = (t_1, H_1)}{H;\triangle;\Gamma \vdash x[\mu] \Rightarrow x[t_1];H_1}$$

$$(tfn) \quad \frac{H + h[t_i \mapsto t_i, k];\triangle[t_i :: k];\Gamma \vdash e \Rightarrow e';H' + h'}{H;\triangle;\Gamma \vdash \Lambda t_i :: k.e \Rightarrow \Lambda t_i :: k.Tlet(h', e');H'}$$

Figure 3.2: The CTE algorithm

The $Tlet(h, e)$ construct used in the algorithm above is akin to the **lettype** construct. If $h = [t_1 \mapsto \mu_1] \ldots [t_n \mapsto \mu_n]$ then $Tlet(h, e)$ is expanded as

$$(\Lambda t_1(\Lambda t_2 \ldots (\Lambda t_n e)\mu_n \ldots)\mu_2)\mu_1$$

so that in the expression e, the type variables $t_1 \ldots t_n$ get bound to the type expressions $\mu_1 \ldots \mu_n$.

The **I** function used in the algorithm does the actual common type expression elimination and is shown in Fig 3.3. The **S** function actually searches for the type expression in the environment and if it already finds the expression in the environment returns the type variable bound to it. Before we describe the **S** function we define the following terms –

- ndepth = nesting depth of the innermost bound type variable in a type expression.
- adj(μ) = converts the type expression μ to μ' such that ndepth(μ') is equal to 1. We assume a familiarity with Debruijn indices and how they are used to represent types. The *adj* function basically manipulates the Debruijn indices.

The **S** function is defined as follows –

1. d = ndepth(μ)
2. μ' = adj(μ) and i = hash(μ')
3. let h = H_d. Search in h for the expression with hash value i.

3.2.2 Elimination of Common Type Expressions

Lemma 3.2.1 *A common type expression μ will always be eliminated if μ is stored in H_d where d = ndepth(μ).*

Proof. Suppose there is another type expression μ' equal to μ in the program. Since we assume unique bindings, we can also assume that two common type expressions will be structurally equal and vice versa (if we were to

$$(tvar) \qquad \frac{t \mapsto (\mu, k) \in H}{I(\mu, H) = (H, t, k)}$$

$$(int) \qquad I(int, H) = (H, int, \Omega)$$

$$(\mu_1 \to \mu_2) \qquad \frac{S(\mu_1 \to \mu_2) = \phi \quad I(\mu_1, H) = (H_1, t_1, \_) \quad I(\mu_2, H_1) = (H_2, t_2, \_) \quad t \text{ a new typevar}}{I(\mu_1 \to \mu_2, H) = (H[t \mapsto t_1 \to t_2, \Omega], t, \Omega)}$$

$$(\mu_1[\mu_2]) \qquad \frac{S(\mu_1[\mu_2]) = \phi \quad I(\mu_1, H) = (H_1, t_1, k_1) \quad I(\mu_2, H_1) = (H_2, t_2, k_2) \quad t \text{ a new typevar}}{I(\mu_1[\mu_2], H) = (H[t \mapsto t_1[t_2], k_1[k_2]], t, k_1[k_2])}$$

$$(\lambda t :: k.\mu) \qquad \frac{S(\lambda t_1 :: k_1.\mu) = \phi \qquad \triangle(\lambda t_1 :: k_1.\mu) = k_1 \to k_2 \qquad t \text{ a new typevar}}{I(\lambda t_1 :: k_1.\mu, H) = (H[t \mapsto \lambda t_1 :: k_1.\mu, k_1 \to k_2], t, k_1 \to k_2)}$$

Figure 3.3: The algorithm I

replace the Debruijn indices by the corresponding type variable). Therefore $ndepth(\mu) = ndepth(\mu')$. Since we use Debruijn indices, $adj(\mu) = adj(\mu')$. This implies that $hash(adj(\mu)) = hash(adj(\mu'))$. And since μ is stored in H_d and we also search in H_d, the common type expression μ' will be eliminated.

Again to simplify the actual presentation, we assume in Fig 3.3 that when the environment H is augmented with a new binding, the new binding is added at the correct place. We do not show explicitly the process of finding out the *ndepth* and then the calling of *adj* before adding a type expression to the environment.

Notation 5 *To simplify things we will use the following notation for lettype. When we have the following expression* $(\Lambda t_1(\Lambda t_2 \ldots (\Lambda t_n.e)\mu_n \ldots)\mu_2)\mu_1$, *it means that* μ_1 *is first substituted right through for* t_1 *and then* μ_2 *is substituted right through for* t_2 *and so on. We will abuse notation and use* $e[t_1/\mu_1]\ldots[t_n/\mu_n]$ *to denote this process of performing the substitutions in a nested manner but performing them completely at each stage.*

Definition 3.2.2 *I preserves well kindedness if* $I(H, \mu) = (H', t, k)$ *and* μ *also has kind* k *in the kind environment* $\triangle$.

Lemma 3.2.3 *The algorithm I above preserves well kindedness.*

Proof. The proof can be done easily by induction over the structure of μ - the only non-trivial case being that of constructor application.

Definition 3.2.4 *Suppose* $I(H, \mu) = (H', t, k)$ *and* $H' = [t_1 \mapsto \mu_1]\ldots[t_n \mapsto \mu_n]$. *Then I preserves well formedness if* $(t)[t_1/\mu_1]\ldots[t_n/\mu_n]$ *evaluates to* μ.

Lemma 3.2.5 *The algorithm I above preserves well formedness.*

Proof. This can be shown by induction over the structure of μ. The only non-trivial case is that of constructor abstraction. where we need to note that the type variables introduced by polymorphic abstractions are mapped only to themselves. It is only newly introduced type variables that are mapped to type expressions. Therefore the body of the constructor abstraction will remain unchanged.

3.2.3 Type Preservation

We will now state and prove the type preservation theorem. The static semantics for the language is shown in Fig 3.4. The only addition from the ones we have already seen is the rule for $Tlet$ which says that to typecheck the body, we first substitute the "letType bounded variable" in the body with the type expression.

$$(const/var) \qquad \frac{}{\triangle;\Gamma \vdash i : \mathtt{Int}} \qquad \frac{}{\triangle;\Gamma \vdash x : \Gamma(x)}$$

$$(fn) \qquad \frac{\triangle;\Gamma \uplus \{x : \mu_1\} \vdash e : \mu_2}{\triangle;\Gamma \vdash \lambda x : \mu_1.e : \mu_1 \to \mu_2}$$

$$(app) \qquad \frac{\triangle;\Gamma \vdash x_1 : \mu' \to \mu \quad \triangle;\Gamma \vdash x_2 : \mu'}{\triangle;\Gamma \vdash @x_1x_2 : \mu}$$

$$(tfn) \qquad \frac{\triangle[t :: k];\Gamma \vdash e : \mu}{\triangle;\Gamma \vdash \Lambda t{::}k.e : \forall t{::}k.\mu}$$

$$(tapp) \qquad \frac{\triangle;\Gamma \vdash x : \forall t{::}k.\mu \qquad \Delta \triangleright \mu' :: k}{\triangle;\Gamma \vdash x[\mu'] : [\mu'/t]\mu}$$

$$(let) \qquad \frac{\triangle;\Gamma \vdash e_1 : \mu_1 \qquad \triangle;\Gamma \uplus \{x : \mu_1\} \vdash e_2 : \mu_2}{\triangle;\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \mu_2}$$

$$(Tlet) \qquad \frac{\triangle;\Gamma \vdash e[\mu'/t] : \mu}{\triangle;\Gamma \vdash (\Lambda t.e)[\mu'] : \mu}$$

Figure 3.4: Static Semantics

Theorem 3.2.6 (Type Preservation) *If $H;\triangle;\Gamma \vdash e \Rightarrow e';H'$ and H is well kinded and well formed and $\triangle;\Gamma \vdash e : \mu$, then $\triangle;\Gamma \vdash Tlet(H', e') : \mu$ and H' is well kinded and well formed.*

Proof. The proof is by induction on the structure of e. We will only consider the tfn case here. The other cases are not difficult.

Case tfn = To prove that if H is well formed and kinded and $\Lambda t{::}k.e : \forall t.\mu$, then $Tlet(H', \Lambda t{::}k.Tlet(h', e')) : \forall t.\mu$ and H' is well formed and well kinded.

By induction we know that $H' + h'$ is well kinded and well formed and therefore H' is also well kinded and well formed. Also by induction $\Lambda t{::}k.Tlet(H' + h', e') : \forall t.\mu.$ which implies that $\Lambda t{::}k.Tlet(H', Tlet(h', e')) : \forall t.\mu.$

Suppose $H' = [t_1 \mapsto \mu_1] \ldots [t_n \mapsto \mu_n]$. In the expression $Tlet(H', \Lambda t :: k.Tlet(h', e'))$, notice that t does not occur in any of the $\mu_i's$ and is not equal to any of the $t_i's$. Therefore the expression may be rewritten as $\Lambda t{::}k.Tlet(H', Tlet(h', e'))$ which is what we get from the inductive assumption.

3.2.4 Semantic Soundness

We will now prove the semantic soundness property of the algorithm. The operational semantics is shown in Fig 3.5, the only addition being the rule for $Tlet$. To evaluate such an expression, we first substitute for the "lettype bounded variable" in the expression and then evaluate the resulting expression.

There are only three kinds of values - integers, function closures and type function closures.

$$(const/var) \qquad a \vdash i \to i \qquad a \vdash x \to a(x)$$

$$(fn) \qquad a \vdash \lambda x{:}\mu.e \to Clos\langle x^{\mu}, e, a\rangle$$

$$(app) \qquad \frac{a \vdash x_1 \to Clos\langle x^{\mu}, e, a'\rangle \quad a \vdash x_2 \to v' \quad a' + x \mapsto v' \vdash e \to v}{a \vdash @x_1 x_2 \to v}$$

$$(tfn) \qquad a \vdash \Lambda t :: k.e \mapsto Clos^t\langle t, e, a\rangle$$

$$(let) \qquad \frac{a \vdash e_1 \to v_1 \qquad a + x \mapsto v_1 \vdash e_2 \to v}{a \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \to v}$$

$$(tapp) \qquad \frac{a \vdash x \mapsto Clos^t\langle t, e, a'\rangle \quad a' \vdash e[\mu/t] \to v}{a \vdash x[\mu] \mapsto v}$$

$$(Tlet) \qquad \frac{a \vdash e[\mu/t] \to v}{a \vdash (\Lambda t :: k.e)[\mu] \to v}$$

Figure 3.5: Operational Semantics

$$(values) \quad v \quad ::= \quad i \mid Clos\langle x^{\mu}, e, a\rangle \mid Clos^t\langle \overline{t_i}, e, a\rangle$$

Notation 6 *The relation $a : \Gamma \vdash e \to v$ means that in a value environment a respecting Γ, e evaluates to v. a respects Γ means that if $a(x) = v$ and $\Gamma(x) = \mu$, then $\Gamma \vdash v : \mu$.*

Notation 7 *The notation $a(x \mapsto ..)$ means that in the environment a, x has the given value. Whereas $a[x \mapsto ..]$ means that the environment a is augmented with the given binding. Continuing from above we get,*

Definition 3.2.7 (Type of a Value)

- $\Gamma \vdash i : int$
- if $\Gamma \vdash \lambda x{:}\mu.e : \mu \to \mu'$, then $\Gamma \vdash Clos\langle x^{\mu}, e, a\rangle : \mu \to \mu'$
- if $\Gamma \vdash \Lambda\overline{t_i}.e : \forall\overline{t_i}.\mu$, then $\Gamma \vdash Clos^t\langle \overline{t_i}, e, a\rangle : \forall\overline{t_i}.\mu$

Throughout the proofs we assume that the subject reduction lemma holds. That is if $a : \Gamma \vdash e \to v$ and $\Gamma \vdash e : \mu$, then $\Gamma \vdash v : \mu$. This lemma can be proved from the given operational semantics by structural induction on the syntactic structure of e.

Definition 3.2.8 (Equivalence of Values)

- **Equivalence of Int** — Suppose $\Gamma \vdash i : int$ and $\Gamma' \vdash i' : int$. Then $i \approx i'$ iff $i = i'$.
- **Equivalence of Closures**
  - Suppose $\Gamma \vdash Clos\langle x^{\mu}, e, a\rangle : \mu \to \mu'$ and $\Gamma' \vdash Clos\langle x^{\mu}, e', a'\rangle : \mu \to \mu'$.

– Suppose further that $\Gamma \vdash v_1 : \mu$ and $\Gamma' \vdash v_1' : \mu$ and $v_1 \approx v_1'$.

Then $Clos\langle x^\mu, e, a\rangle \approx Clos\langle x^\mu, e', a'\rangle$ iff $\forall v_1, v_1' \quad a : \Gamma + x \mapsto v_1 \vdash e \to v$ and $a' : \Gamma' + x \mapsto v_1' \vdash e' \to v'$ and $v \approx v'$

- **Equivalence of Type Closures** Suppose $\Gamma \vdash Clos^t\langle \overline{t_i}, e, a\rangle : \forall \overline{t_i}.\mu$ and $\Gamma' \vdash Clos^t\langle \overline{t_i}, e', a'\rangle : \forall \overline{t_i}.\mu$. Then $Clos^t\langle \overline{t_i}, e, a\rangle \approx Clos^t\langle \overline{t_i}, e', a'\rangle$ iff $a : \Gamma \vdash e[\mu_i/t_i] \to v$ and $a' : \Gamma' \vdash e'[\mu_i/t_i] \to v'$ and $v \approx v'$.

Definition 3.2.9 (Equivalence of terms) *Suppose $a : \Gamma \vdash e \to v$ and $a' : \Gamma' \vdash e' \to v'$ with $\Gamma \vdash e : \mu$ and $\Gamma' \vdash e' : \mu$. If $v \approx v'$, then we say that the terms e and e' are semantically equivalent and denote this by $a : \Gamma \vdash e \approx a' : \Gamma' \vdash e'$.*

Lemma 3.2.10 *if $a : \Gamma \vdash e \approx a' : \Gamma' \vdash e'$, then $a : \Gamma \vdash e[\mu_i/t_i] \approx a' : \Gamma' \vdash e'[\mu_i/t_i]$*

Theorem 3.2.11 (Semantic Soundness) *Suppose $H; \triangle; \Gamma \vdash e \Rightarrow e'; H'$. Then if a respects Γ, and $a : \Gamma \vdash e \to v$, then $a : \Gamma \vdash Tlet(H', e') \to v'$ and $v \approx v'$.*

Proof. The proof is by induction on the structure of e. Again we will consider only the tfn case. The other cases follow without much difficulty by a similar process.

We need to prove that $a : \Gamma \vdash e[\mu/t] \approx a : \Gamma \vdash Tlet(H', (Tlet(h', e')[\mu/t]))$. But suppose that $H' = [t_1 \mapsto \mu_1] \ldots [t_n \mapsto \mu_n]$. Then t is not equal to any of the $t_i's$ and is not contained in any of the $\mu_i's$. Therefore we need to prove that $a : \Gamma \vdash e[\mu/t] \approx a : \Gamma \vdash (Tlet(H', Tlet(h', e')))[\mu/t]$ or that $a : \Gamma \vdash e[\mu/t] \approx a : \Gamma \vdash (Tlet(H' + h', e'))[\mu/t]$ or that $a : \Gamma \vdash e \approx a : \Gamma \vdash Tlet(H' + h', e')$ which follows by the inductive hypothesis.

3.3 Implementation Results

We will now talk about the actual implementation of the algorithm on the FLINT/ML compiler version 110.5. Since we represent types by Debruijn indices and the CTE algorithm involves a lot of manipulation of types, the implementation gets pretty tricky.

We implemented the algorithm as a two pass algorithm with each pass taking time proportional to the length of the input program. In the first pass, we collect the common type expressions and in the second pass we substitute the type expressions with the newly introduced type variables. Doing the whole thing in a single pass is pretty tricky. Consider a code fragment

$$\Lambda t_1 .... e_1 ... \quad \Lambda t_2 .... e_2 ... \quad \Lambda t_3 .... e_3 ..$$

The common type expressions involving t_1 will be dropped as a **letType** just in front of the corresponding abstraction. But the problem is that we donot know a-priori how many such expressions will be collected. So for example, when we are processing e_2 we do not know how many expressions will be dropped at the t_1 abstraction. This is because we could encounter expressions involving t_1 while processing the code fragment e_3. Therefore while processing e_2 during the first pass, we have no idea of what its final Debruijn depth will be.

Another complication relates to the fact that the required adjustment of the Debruijn indices is not uniform. Consider the type expression $(t_1 * t_2)$ occuring in e_2. Previously this would be represented as $((2,0) * (1,0))$. Now this gets represented as $((m,0) * (n,0))$ with no relation between m, n. This is because the values of m, n depend on the number of common type expressions dumped at the two abstractions which are not related to each other.

We implemented the algorithm as follows. In the first pass we collect all the common type expressions in the code. We use the Debruijn indices to compute hash values and for easy equality testing. But in this pass, we replace the common type expressions by named type variables – the code at the end of the first pass

therefore contains named type variables instead of Debruijn indices. During the second pass, we maintain a mapping between the names of type variables and the depth at which they are defined and replace the named type variables with their corresponding Debruijn indices.

In the algorithm shown in Fig 3.2, we do not eliminate common type expressions inside type functions. This was done to make the algorithm simpler – otherwise it introduces a few extra parameters and complicates the proofs. In the actual implementation of course, we catch common type expressions inside type functions.

In Fig 3.6, we show some performance figures for the CTE eliminator. It shows the number of common type expressions eliminated and the size of the eliminated type expressions. For some heavily polymorphic benchmarks like **vliw, kb-comp, boyer**, the savings are considerable while in the case of monomorphic benchmarks like **fft**, the gains are negligible.

The algorithm presented here is of course based on the idea of common subexpression elimination [1, 3] that is done by any standard compiler. The TIL compiler [49] uses an explicit lettype construct in the language to eliminate common type expressions.

Benchmark	Description	No. of Exps eliminated	Size of Exps eliminated
Vliw	A VLIW instruction scheduler	5	1
		92	2
		2	3
		4	4
		51	613
		2	614
		10	742
		1	7
		140	105
		11	1848
lexgen	A lexical-analyzer generator	12	1
		7	2
		8	3
		8	6
		5	9
		12	15
		7	19
		5	23
Mandelbrot	The Mandelbrot curve construction	0	0
Kb-comp	Knuth-Benedix Completion Algorithm	1	1
		115	13
		14	14
		14	26
		1	27
		27	28
		8	29
Ray	A ray-tracer	1	16
Life	Runs 10,000 generations of the Life Simulation	105	2
Boyer	A simple theorem prover	3	44
		3	53
		14	22
		3	26
fft	A Fast-Fourier Transform	0	0

Figure 3.6: CTE Results

Chapter 4

Runtime Type Representation

4.1 Introduction

In this chapter, we show the new runtime representation of types in FLINT. This is yet to be fully implemented. Unlike the previous two phases, the correctness of this phase is critical to the compiler. The optimisations mentioned before can be turned off and the most we end up paying is a performance penalty. But this phase is closely linked to the reification stage of the compiler. Any bugs in this phase renders the compiler unusable and is therefore disastrous.

Presently the FLINT compiler represents all types by small integers. We need to distinguish between a few types only – integers, floats, a pair of floats, a record of floats. This is because we use a more efficient data representation for these cases and therefore the functions to access these data types can not assume a standard boxed representation. We assign a unique integer to each of these types and a different integer value to all other types so that at runtime we can check whether a particular object uses a boxed representation or a natural representation. However, in applications such as pretty printing, debugging and type dynamic we want a more sophisticated type representation. The new runtime type representation that we are implementing now maintains full type information at runtime with a view to supporting these applications.

4.2 Description of the Algorithm

We need to define two mappings – one from the kinds to the types and the other from the constructors to the terms. This is because FLINT is an explicitly typed calculus and therefore the translated terms must be annotated with type information as well. The kind and the constructor calculus in FLINT is repeated in Fig 4.1 Monotypes like `Int` and `Real` are of kind Ω. The arrow constructor $(\mu_1 \rightarrow \mu_2)$ and the tuple constructor $[\mu_1 \times \ldots \times \mu_n]$ are also of kind Ω. The constructor function has the function kind while the sequence of constructors $(\mu_1, \ldots, \mu_n)$ has the sequence kind. The projection operator projects from a sequence of constructors.

$$
\begin{array}{llll}
(\textit{kinds}) & \kappa & ::= & \Omega \mid \kappa_1 \rightarrow \kappa_2 \mid (\kappa_1, \ldots, \kappa_n) \\
(\textit{constructors}) & \mu & ::= & t \mid \texttt{Int} \mid \texttt{Real} \mid \mu_1 \rightarrow \mu_2 \mid \lambda t{::}\kappa.\mu \\
 & & & \mid [\mu_1 \times \ldots \times \mu_n] \mid \mu_1[\mu_2] \mid (\mu_1, \ldots, \mu_n) \mid \pi_i \mu
\end{array}
$$

Figure 4.1: The Kind and Constructor Calculus

The target type and term language is shown in Fig 4.2 and is mostly standard. It has a dedicated type `Rtype` which is needed to type the runtime representation of the constructors. The kind Ω maps to this type. The

term language also includes a bunch of primops. These primops which are predefined in the language have the function type. They operate upon values of the argument type and yield a term having the result type.

$$\begin{array}{llll} (\textit{types}) & \sigma & ::= & T(\mu) \mid \texttt{Rtype} \mid \sigma_1 \to \sigma_2 \mid \forall t{::}\kappa.\sigma \mid (\sigma_1, \ldots, \sigma_n) \\ (\textit{terms}) & e & ::= & i \mid x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \mid @e_1 e_2 \mid (e_1, \ldots, e_n) \mid \pi_i e \\ & & & \mid \lambda x{:}\sigma.e \mid \Lambda t{::}k.e \mid x[\mu] \mid \text{prim(e)} \end{array}$$

Figure 4.2: The target term and type language

The kind translation is given in Fig 4.2. The kind Ω maps to the type `Rtype` and the other kind translations are defined recursively.

The constructor translation is defined in Fig 4.4. One of the problems of the translation is that the representation of the base constructors such as `Int`, `Real` and the representation of the tuple and arrow constructors must all have the same type since they all have the same kind. We could represent them all as a record. But firstly this is grossly inefficient. Secondly this would lead to typechecking problems under the given kind translation. Thirdly, we need a representation for a sequence of constructors. The kind checking rules for a sequence of constructors suggest that it maps naturally to a record of the runtime representations of the constituent constructors. On the other hand, to maintain complete type information, we need to retain the representation of every constituent of a tuple constructor or an arrow constructor.

Our solution is to use *primops* to build the runtime representation. The types of the primops are defined so that the result has the proper type – in essence we cast the result to the proper type. In the algorithm in Fig 4.4, *primty* is a primop that takes an integer and casts it to the type `Rtype`. In Fig 4.4 i and j are the integers that will be used to denote `Int` and `Real`. The primops *primarr* and *primtup* similarly construct a record of the runtime representations and cast the result to the type `Rtype`.

We therefore represent primitive type constructors like `Int` and `Real` as small integers. A tuple of constructors or the arrow constructor is represented as a record of the representations of the corresponding constructors. A sequence of constructors is also represented as a record. We use a tag to distinguish between the different cases. A projection constructor is represented as a selection from a record. A constructor function is represented as a function at the term level with the kind translation determining the argument type of the function. Finally, a constructor application is represented as an application at the term level.

$$\begin{array}{ll} (\Omega) & \Omega \Rightarrow \texttt{Rtype} \\ (\kappa_1 \to \kappa_2) & \dfrac{k_1 \Rightarrow \sigma_1 \qquad k_2 \Rightarrow \sigma_2}{k_1 \to k_2 \Rightarrow \sigma_1 \to \sigma_2} \\ ((\kappa_1, \ldots, \kappa_n)) & \dfrac{k_i \Rightarrow \sigma_i}{(\kappa_1, \ldots, \kappa_n) \Rightarrow (\sigma_1, \ldots, \sigma_n)} \end{array}$$

Figure 4.3: The Kind Translation - Algorithm ($\mathcal{K}$)

$$(tvar) \qquad \frac{\Gamma(t) = x}{\Gamma \vdash t \Rightarrow x}$$

$$(prims) \qquad \Gamma \vdash Int \Rightarrow primty^{\dagger}(i) \qquad \Gamma \vdash Real \Rightarrow primty^{\dagger}(j)$$

$$(arrow) \qquad \frac{\Gamma \vdash \mu_1 \Rightarrow e_1 \qquad \Gamma \vdash \mu_2 \Rightarrow e_2}{\Gamma \vdash \mu_1 \rightarrow \mu_2 \Rightarrow primarr^{\dagger}(e_1, e_2)}$$

$$(tuple) \qquad \frac{\Gamma \vdash \mu_i \Rightarrow e_i}{\Gamma \vdash [\mu_1 \times \ldots \times \mu_n] \Rightarrow primtup^{\dagger}(e_1 \ldots e_n)}$$

$$(seq) \qquad \frac{\Gamma \vdash \mu_i \Rightarrow e_i}{\Gamma \vdash (\mu_1, \ldots, \mu_n) \Rightarrow (e_1, \ldots, e_n)}$$

$$(sel) \qquad \frac{\Gamma \vdash \mu \Rightarrow e}{\Gamma \vdash \pi_i \mu \Rightarrow \pi_i e}$$

$$(fn) \qquad \frac{\Gamma[t \mapsto x] \vdash \mu \Rightarrow e \qquad x \text{ a new variable}}{\Gamma \vdash \lambda t :: k.\mu \Rightarrow \lambda x : \mathcal{K}(k).e}$$

$$(app) \qquad \frac{\Gamma \vdash \mu_1 \Rightarrow e_1 \qquad \Gamma \vdash \mu_2 \Rightarrow e_2}{\Gamma \vdash \mu_1[\mu_2] \Rightarrow @e_1 e_2}$$

† Their implementation is described in the text

Figure 4.4: The Translation of the Constructors

4.3 Implementation of the Algorithm

As we said at the beginning of the chapter, this is still very much work in progress. We have tried to be very conservative in making changes to the existing type representation and reification code. Presently, we are working by representing everything as a record and tagging the first field of the record according to the constructor. This makes the implementation a lot easier and once we are confident about the current implementation, we will move to a more efficient one. The final implementation should closely match the algorithm given in this chapter except that we intend to represent a record of floats as a small integer as well – this would make testing for a floating point record a little more efficient.

The other matter which we didn't address in this chapter and haven't handled in the implementation yet is the representation of recursive datatypes. We will represent recursive types as a fix point of a function where the function is derived inductively in an obvious way. This in turn leads to the question of testing two recursive datatypes for equality – how do we compare the respective functions for equality. For this we will be comparing the two functions for structural equality.

Conclusions

This report deals with efficient runtime type passing. We presented the optimal type lifting algorithm in Chapter 2 which eliminates runtime type construction inside functions and guarantees that all type information is computed at link time. We saw that it provides moderate speedup for the polymorphic benchmarks. We presented the common type expression elimination algorithm in Chapter 3 and saw that it eliminates a lot of type construction for the polymorphic benchmarks. We then presented the new runtime type representation in Chapter 4 which maintains complete type information at runtime so that we can support applications like pretty printing, debugging, pickling and type dynamic.

Bibliography

[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] 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.

[6] 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.

[7] 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.

[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] 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.

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

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

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

[14] 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.

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

[16] R. Hughes. *The design and implementation of programming languages.* PhD thesis, Programming Research Group, Oxford University, Oxford, UK, 1983.

[17] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *The Second International Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, New York, September 1985. Springer-Verlag.

[18] M. P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Oxford University Computing Laboratory, Oxford, july 1992. Technical Monograph PRG-106.

[19] M. P. Jones. A theory of qualified types. In *The 4th European Symposium on Programming*, pages 287–306, Berlin, February 1992. Spinger-Verlag.

[20] M. P. Jones. Dictionary-free overloading by partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 107–117. University of Melbourne TR 94/9, June 1994.

[21] 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.

[22] X. Leroy and M. Mauny. Dynamics in ML. In *The Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 406–426, New York, August 1991. Springer-Verlag.

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

[24] 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.

[25] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

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

[27] Y. Minamide. Full lifting of type parameters. Technical report, RIMS, Kyoto University, 1997.

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

[29] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Trans. Prog. Lang. Syst.*, 13(3), July 1991.

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

[31] 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.

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

[33] A. Ohori. A compilation method for ML-style polymorphic record calculi. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1992. ACM Press.

[34] 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.

[35] S. Peyton Jones. Compiling haskell by program transformation: a report from trenches. In *Proceedings of the European Symposium on Programming*, Linkoping, April 1996.

[36] 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.

[37] S. Peyton Jones and D. Lester. A modular fully-lazy lambda lifter in haskell. *Software – Practice and Experience*, 21:479–506, 1991.

[38] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. International Conference on Functional Programming (ICFP'96)*, New York, June 1996. ACM Press.

[39] 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.

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

[41] 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.

[42] 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.

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

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

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] A. K. Wright. Polymorphism for imperative languages without imperative types. Technical Report Tech Report TR 93-200, Dept. of Computer Science, Rice University, Houston, Texas, February 1993.