Optimal Type Lifting*

Bratin Saha and Zhong Shao

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

Abstract. Modern compilers for ML-like polymorphic languages have used explicit run-time type passing to support advanced optimizations such as intensional type analysis, representation analysis and tagless garbage collection. Unfortunately, maintaining type information at run time can incur a large overhead to the time and space usage of a program. In this paper, we present an optimal type-lifting algorithm that lifts all type applications in a program to the *top* level. Our algorithm eliminates all run-time type constructions within any core-language functions. In fact, it guarantees that the number of types built at run time is strictly a static constant. We present our algorithm as a type-preserving source-to-source transformation and show how to extend it to handle the entire SML'97 with higher-order modules.

1 Introduction

Modern compilers for ML-like polymorphic languages [16, 17] usually use variants of the Girard-Reynolds polymorphic λ-calculus [5, 26] as their intermediate language (IL). Implementation of these ILs often involves passing types explicitly as parameters [32, 31, 28] at runtime: each polymorphic type variable gets instantiated to the actual type through run-time type application. Maintaining type information in this manner helps to ensure 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. Intensional type analysis [7, 31, 27], which is used by some compilers [31, 28] to support efficient data representation, also requires the propagation of type information into the target code. Run-time type information is also crucial to the implementation of tag-less garbage collection [32], pickling, and type dynamic [15].

* This research was sponsored in part by the DARPA ITO under the title "Software Evolution using HOT Language Technology", DARPA Order No. D888, issued under Contract No. F30602-96-2-0232, and in part by an NSF CAREER Award CCR-9501624, and NSF Grant CCR-9633390. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

However, the advantages of runtime type passing do 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 [32] 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 [18]. In fact, a better goal would be to guarantee that explicit type passing never blows up the execution cost of a program.

Consider the sample code below – we took some liberties with the syntax by using an explicitly typed variant of the Core-ML. Here Λ denotes type abstraction, 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 refers to 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 arising from 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[α]). But optimizing type applications such as `f[s*s]` is less obvious; `f` is nested inside `pair`, and its repeated type applications are not apparent in the `doit` function. We may type-specialize `f` to get rid of the type application but in general this may lead to substantial code duplication. Every time `doit` is called, `pair`[α] gets executed and then every time `pair` is called, `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 paper, 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 without having to worry about the run-time cost of doing so.

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 performed 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 the guarantee that all type applications 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 in turn will allow us to use a more sophisticated runtime type representation and make greater use of type information at runtime.

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 [28] and tested it on a few benchmarks. We provide the implementation results at the end of this paper.

2 The Lifting Algorithm For Core-ML

This section presents our optimal type lifting algorithm. We use an explicitly typed variant of the Core-ML calculus [6] (Figure 1) as the source and target languages. The type lifting algorithm (Figure. 2) is expressed as a type-directed program transformation that lifts all type applications to the top-level.

2.1 The language

We use an explicitly typed variant of the Core-ML calculus [6] as our source and target languages. The syntax is shown in Figure 1. The static and dynamic semantics are standard, and are given in the Appendix (Section 7).

Here, terms e consist of identifiers (x), integer constants (i), function abstractions, function applications, and let expressions. We differentiate between

[1] We are not referring to "link time" in the traditional sense. Rather, we are referring to the run time spent on module initialization and module linkage (e.g., functor application) in a ML-style module language.

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

Fig. 1. An explicit Core-ML calculus

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 $\forall t_1 \ldots \forall t_n.\mu$. The *vterms* ($e_v$) denote values – terms that are free of side-effects.

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 applications and polymorphic definitions is semantically sound [33]. 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 Figure 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` (bound to the lifted type application) is monomorphic and is not applied further to types. Finally, following SML [17, 16], 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 [4]. We also use indentation to indicate the nesting.

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 as the number of λ(value) abstractions within which it is nested. Consider the terms outside all value abstractions to be at depth zero. Obviously, terms at depth zero occur 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. To avoid repeated type applications, the algorithm therefore tries to lift all of them to depth zero. But since we want to lift type applications, we must also lift the polymorphic functions to depth zero. The algorithm scans the input program and collects all the type applications and polymorphic functions occuring at depths greater than zero and adds them to a list `H`. (In the algorithm given in Figure 2, the depth is implicitly

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

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 Section 1. In the example code, `f[s*s]` is at depth 1 since it occurs inside the λx, `Array.sub[`α`*`α`]` and `pair[`α`]` are at depth 2 since they occur inside the λa and λi. We want to lift all of these type applications to depth zero. Translating `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` (inside `pair`). This requires us to lift `f`'s definition by abstracting over its free variables. In the resulting code, all type applications occur at depth zero. Therefore when `main` is called at the beginning of execution, $\mathtt{v}_1$, $\mathtt{v}_2$ and $\mathtt{v}_3$ get evaluated. During execution, when the function `loop` runs through the array and repeatedly calls function `doit`, none of the type applications need to be performed – the type specialised functions $\mathtt{v}_1$, $\mathtt{v}_2$ and $\mathtt{v}_3$ can be used instead.

```
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)
```

2.3 Formalization

Figure 2 shows the 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', and F is the set of free variables of e'. (The set F is restricted to the monorphically typed free variables of e') The *header* H contains the polymorphic functions and type applications occuring in e at depths greater than zero. The final result of lifting a closed term e of type μ is $LET(H, e')$ where the algorithm infers $\emptyset \vdash e : \mu \Rightarrow e'; H; \emptyset$. 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 resulting term let $x_1 = e_1$ in $\ldots$ in let $x_n = e_n$ in e.

The environment Γ maps a variable to its type and to a list of the free variables in its definition. In the algorithm, we use standard notation for lists and operations on lists; 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 $@^* v L$ reduce to $\lambda x_1 : \mu_1 . \ldots \lambda x_n : \mu_n . e$ and $@(\ldots(@v x_1)\ldots) x_n$, respectively, where $L = [x_1 : \mu_1, \ldots, x_n : \mu_n]$.

Rules (*exp*) and (*app*) are just the identity transformations. Rule (*fn*) deals with abstractions. We translate the body of the abstraction and return a header H containing all the type applications and type functions in the term e.

The translation of monomorphic `let` expressions is similar. We translate each of the subexpressions replacing the old terms with the translated terms 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 from the translation of the subexpressions.

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

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

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

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

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

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

Fig. 2. The Lifting Translation

The real work is done in the last two rules which deal 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(e_2). The result of the translation is only e_2'; the polymorphic function introduced by the **let** is added to the result header H_r so that it is lifted to the top level. The polymorphic function body (in H_r) is closed by abstracting over its free variables F_1 while the header H_1 is 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 get lifted to the top level.

The (*tapp*) rule replaces the type application by an application of the newly introduced variable (v) to the free variables(L) of the corresponding function definition. The type application is added to the header and lifted to the top level where it gets bound to v. Note that the free variables of the translated term do not include the newly introduced variable v. This is because when the header is written out at the top level, the translated expression remains in the scope of the dumped header.

Proposition 1. *Suppose* $\Gamma \vdash e : \mu \Rightarrow e'; H; F$. *Then in the expression LET(H,e'), the term* e' *does not contain any type application and* H *does not contain any type application nested inside a value*(λ) *abstraction.*

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

Theorem 1 (Full Lifting). *Suppose* $\Gamma \vdash e : \mu \Rightarrow e'; H; F$. *Then the expression LET(H,e'), does not have any type application nested inside a value abstraction.*

The theorem follows from Proposition 1.
In the Appendix, we prove the type preservation and the semantic soundness theorems.

2.4 A closer look

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 [10]. However the lifting in the two cases is different. Lambda lifting converts a program with local function definitions into a program with 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* [9, 24] with the maximal free subexpressions restricted to be type applications. However, the fully-lazy transformation as described in Peyton Jones [24] 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.

Minamide [18] uses a different approach to solve this problem. He lifts the construction of type parameters from within a polymorphic function to the call sites of the function. This lifting is recursively propagated to the call sites at the top level. At runtime, type construction is replaced by projection from type parameters.

His method eliminates the runtime construction of types and replaces it by projection from type records. The transformation also does not rely on the value restriction for polymorphic definitions. However, he requires a more sophisticated type system to type-check his transformation; he uses a type system based on the qualified type system of Jones [12] and the implementation calculus for the compilation of polymorphic records of Ohori [21]. Our algorithm on the other hand is a source-to-source transformation. Moreover, Minamide's algorithm deals only with the Core-ML calculus whereas we have implemented our algorithm on the entire SML'97 language with higher-order modules.

Jones [11] has also worked on a similar problem related to dictionary passing in Haskell and Gofer. Type classes in these languages are implemented by passing

dictionaries at runtime. Dictionaries are tuples of functions that implement the methods defined in a type class.

Consider the following Haskell [8] example

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

The actual type of `f` is $Eq[a] \Rightarrow a \rightarrow a \rightarrow Bool$. Context reduction leads to the type specified in the example. 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. In a naive implementation, this function would be passed a dictionary for $Eq\ a$ and the dictionary for $Eq\ [a]$ would be constructed inside the function. Jones optimises this by constructing a dictionary for $Eq\ [a]$ at the call site of `f` and passing it in as a parameter. This is repeated for all overloaded functions so that all dictionaries are constructed statically. But this approach does not work with separately compiled modules since `f's` type in other modules does not specify the dictionaries that are constructed inside it.

In Gofer [11], instance declarations are not used to simplify the context. Therefore the type of `f` in the above example would be $Eq[a] \Rightarrow a \rightarrow a \rightarrow Bool$. Jones' optimisation of dictionary passing can now be performed in the presence of separately compiled modules. However, we now require a more complicated type system to typecheck the code. Assume two functions `f` and `g` have the same type $(\mu \rightarrow \mu')$. Both `f` and `g` can be passed as a parameter to `h` in $(h = \lambda x{:}\mu \rightarrow \mu'.e)$. However, `f` and `g` could, in general, be using different dictionaries (d_f and d_g). This implies that after the transformation, the two functions will have different types – $d_f \Rightarrow \mu \rightarrow \mu'$ and $d_g \Rightarrow \mu \rightarrow \mu'$. Therefore, we can no longer use `f` and `g` interchangeably.

3 The Lifting Algorithm for FLINT

Till now, we have considered only the Core-ML calculus while discussing the algorithm. But what happens when we take into account the module language as well?

To handle the Full-ML langauge, we compile the source code into the FLINT intermediate language. The details of the translation are given in [29]. FLINT is based upon a predicative variant of the Girard-Reynolds polymorphic λ-calculus [5, 26], with the term language written in A-normal form [4]. It contains the following four syntactic classes: kinds (κ), constructors (μ), types (σ) and terms (e), as shown in Figure 3. 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. 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). The terms are an explicitly

typed λ-calculus (but in A-normal form) with explicit constructor abstraction and application.

$$
\begin{array}{lll}
(\textit{kinds}) & \kappa & ::= \Omega \mid \kappa_1 \rightarrow \kappa_2 \\
(\textit{cons}) & \mu & ::= t \mid \mathrm{Int} \mid \mu_1 \rightarrow \mu_2 \mid \lambda t :: \kappa.\mu \mid \mu_1[\mu_2] \\
(\textit{types}) & \sigma & ::= T(\mu) \mid \sigma_1 \rightarrow \sigma_2 \mid \forall t :: \kappa.\sigma \\
(\textit{terms}) & e & ::= i \mid x \mid \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 \mid @x_1 x_2 \\
 & & \quad \mid \lambda^c x\colon T(\mu).e \mid \lambda^m x\colon \sigma.e \\
 & & \quad \mid \mathrm{let}\ x = \Lambda\overline{t_i} :: k_i.e_v\ \mathrm{in}\ e_2 \mid x[\mu_i] \\
(\textit{values}) & e_v & ::= i \mid x \mid \mathrm{let}\ x = e_v\ \mathrm{in}\ e'_v \mid \lambda^c x\colon T(\mu).e \mid \lambda^m x\colon \sigma.e \\
 & & \quad \mathrm{let}\ x = \Lambda\overline{t_i} :: k_i.e_v\ \mathrm{in}\ e'_v \mid x[\mu_i]
\end{array}
$$

Fig. 3. Syntax of the Core-FLINT calculus

In ML, structures are the basic module unit and functors abstract over structures. Polymorphic functions may now escape as part of structures and get initialized later at a functor application site. In the FLINT translation [29], functors are represented as a polymorphic definition combined with a polymorphic abstraction ($\texttt{fct} = \Lambda\overline{t_i} :: \overline{k_i}.\lambda^m x : \sigma.e$). The variable x in the functor definition is polymorphic since the parameterised structure may contain polymorphic components. In the functor body e, the polymorphic components of x are instantiated by type application. Functor applications are a combination of a type application and a term application. with the type application instantiating the type parameters ($t'_i s$). Though abstractions model both functors and functions, the translation allows us to distinguish between them. In the FLINT calculus, $\lambda^c x : T(\mu).e$ denotes functions, whereas $\lambda^m x : \sigma.e$ denotes functors. The rest of the term calculus is standard.

This calculus complicates the lifting since type applications arising from an abstracted variable (the variable x in `fct` above) can not be lifted to the top level. This also differs from the Core-ML calculus in that type applications may now be curried to model escaping polymorphic functions.

However, the module calculus obeys some nice properties. Functors in a program always occur outside any Core-ML functions. Type applications arising out of functor parameters (when the input structure contains a polymorphic component) can therefore be lifted outside all functions. Escaping polymorphic functions occur outside Core-ML functions. Therefore the corresponding curried type application is not nested inside Core-ML functions.

Therefore a FLINT source program can be converted into a *well-formed* program satisfying the following constraints –

- All functor abstractions (λ^m) occur outside function abstractions (λ^c).
- No partial type application occurs inside a function abstraction.

We now redefine the depth of a term in a program as the number of function abstractions within which it is nested with *depth 0 terms occuring outside all*

function abstractions only. Note that depth 0 terms may not occur outside all abstractions since they may be nested inside functor abstractions. We then perform type lifting as in Figure 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 function and the header returned is empty.

We illustrate the algorithm on the example code in Figure 4. The syntax is not totally faithful to the FLINT syntax in Figure 3 but it makes the code easier to understand. In the code in Figure 4, F is a functor which takes the structure

```
F = Λt0.λ^m X:S.
     f = λ^c v.
           let id = Λt1.λ^c x2.x2
               .....
               v1 = ... id[Int](3) ....
           in  v1
     v2 = (#1(X))[t0]
     ....
```

Fig. 4. Example FLINT code

X as a parameter. The type S denotes a structure type. Assume the first component of X $(\#1(X))$ is a polymorphic function which gets instantiated in the functor body(v_2). f is a locally defined function in the functor body. According to the definition of depth above, f and v_2 are at depth 0 even though they are nested inside the functor abstraction(λX). Moreover, the type application $(\#1(X))[t_0]$ is also at depth 0 and will therefore not be lifted. It is only inside the function f that the depth increases which implies that the type application $id[Int]$ occurs at $d > 0$. The algorithm will lift the type application to just outside the function abstraction (λv), it is not lifted outside the functor abstraction (λX). The resulting code is shown in Figure 5.

Is the reformulation 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 construction inside functions. Functors are top level code and are applied at "link" time. Moreover they are non-recursive. Therefore having type applications nested only inside functors 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.

To summarize, we note that depth 0 in Core-ML (according to the definition above) coincides with the top level of the program since Core-ML does not

```
F = Λt0.λ^m X:S.
     f = let
           id = Λt1.λ^c x2.x2
           z1 = id[Int]
           .. (Other type expressions in f's body)..
         in  λ^c v.
               let  ..... (type lifted body of f)
                    v1 = ... z1(3) ....
               in   v1
     v2 = (#1(X))[t0]
     ....
```

Fig. 5. FLINT code after type lifting

have functors; therefore the Core-ML translation is merely a special case of the translation for FLINT.

4 Implementation

We have implemented the type-lifting algorithm in the FLINT/ML compiler version 1.0 and the experimental version of SML/NJ v109.32. All the tests were performed on a Pentium Pro 200 Linux workstation with 64M physical RAM. Figure 6 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.

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

Fig. 6. Type Lifting Results

The current FLINT/ML and SML/NJ compilers maintain a very minimal set of type information. Types are represented by integers since the compiler only

needs to distinguish primitive types (e.g., `int`, `real`) and special record types. As a result, runtime type construction and type application are not expensive. The test results therefore yield a moderate speedup for most 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.

Our algorithm makes the simultaneous uncurrying of both value and type applications difficult. Therefore at runtime, a type application will result in the formation of a closure. However, these closures are created only once at linktime and do not represent a significant penalty.

We also need to consider the closure size of the lifted functions. The (*tapp*) rule in Figure 2 introduces new variables (*the set* L) which may increase the number of free variables of a function. Moreover after type applications are lifted, the type specialised functions become free variables of the function body. On the other hand, since all type applications are lifted, we no longer need to include the free type variables in the closure which decreases the closure size. We believe therefore that the increase in closure size, if any, does not incur a significant penalty. This is borne out by the results on the benchmark suite – none of the benchmarks slows down significantly.

The creation of closures makes function application more expensive since it involves the extraction of the environment and the code. However, in most cases, the selection of the code and the environment will be a loop invariant and can therefore be optimised.

The algorithm is implemented in a single pass by a bottom up traversal of the syntax tree. The (*tfn*) rule shown in Figure 2 simplifies the implementation considerably by reducing the type information to be adjusted. In the given rule, all the expressions in H_1 are dumped right in front of the type abstraction. Note however that we require to dump only those terms (in H_1) which contain any of the $t_i's$ as free type variables. The advantage of dumping all the expressions is that the *de Bruijn* depth of the terms in H_1 remains the same even after lifting. The algorithm needs to adjust the type information only while abstracting the free variables of a polymorphic definition. (The types of the abstracted variables have to be adjusted.) The implementation also optimises the number of variables abstracted while lifting a definition – it remembers the depth at which a variable is defined so that variables that will still remain in scope after the lifting are not abstracted.

5 Related Work and Conclusions

Tolmach [32] 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 [18] proposes a refinement of Tolmach's method to eliminate runtime construction of type parameters. The speedups

obtained in our method are comparable to the ones reported in his paper. Mark P. Jones [11] has worked on the related problem of optimising dictionary passing in the implementation of type classes.

In their study of the type theory of Standard ML, Harper and Mitchell [6] 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.* [19] in the implementation of Napier. The work of Ohori on compiling record operations [21] is similarly based on a type passing interpretation of polymorphism. Jones [12] has proposed *evidence passing*—a general framework for passing data derived from types to "qualified" polymorphic operations. Harper and Morisett [7] 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 [14] and Shao [27]. Many modern compilers like the FLINT/ML compiler [28], TIL [31] and the Glasgow Haskell compiler [22] 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 [9] showed that by doing lambda lifting in a particular way, full laziness can be preserved. Johnsson [10] describes different forms of lambda lifting and the pros and cons of each. Peyton Jones [25, 23, 24] also described a number of optimizations which are similar in spirit but have totally different aims. Appel [2] describes let hoisting in the context of ML. In general, using correctness preserving transformations as a compiler optimization [1, 2] 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 runtime construction of types inside functions. 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. We are now working on making the type representation in FLINT more comprehensive so that we can maintain complete type information at runtime.

6 Acknowledgements

We would like to thank Valery Trifonov, Chris League and Stefan Monnier for many useful discussions and comments about earlier drafts of this paper. We also thank the annonymous referees who suggested various ways of improving the presentation.

References

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

2. A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
3. 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.
4. 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.
5. J. Y. Girard. *Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur.* PhD thesis, University of Paris VII, 1972.
6. R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Trans. Prog. Lang. Syst.*, 15(2):211–252, April 1993.
7. 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.
8. 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.
9. R. Hughes. *The design and implementation of programming languages.* PhD thesis, Programming Research Group, Oxford University, Oxford, UK, 1983.
10. 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.
11. M. P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Oxford University Computing Laboratory, Oxford, july 1992. Technical Monograph PRG-106.
12. M. P. Jones. A theory of qualified types. In *The 4th European Symposium on Programming*, pages 287–306, Berlin, February 1992. Spinger-Verlag.
13. 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.
14. 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.
15. 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.
16. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.
17. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Massachusetts, 1997.
18. Y. Minamide. Full lifting of type parameters. Technical report, RIMS, Kyoto University, 1997.
19. 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.
20. G. Nadathur. A notation for lambda terms II: Refinements and applications. Technical Report CS-1994-01, Duke University, Durham, NC, January 1994.
21. 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.

22. 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.
23. S. Peyton Jones. Compiling haskell by program transformation: a report from trenches. In *Proceedings of the European Symposium on Programming*, Linkoping, April 1996.
24. S. Peyton Jones and D. Lester. A modular fully-lazy lambda lifter in haskell. *Software – Practice and Experience*, 21:479–506, 1991.
25. 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.
26. 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.
27. Z. Shao. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 85–98. ACM Press, June 1997.
28. Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.
29. Z. Shao. Typed cross-module compilation. Technical Report YALEU/DCS/RR-1126, Dept. of Computer Science, Yale University, New Haven, CT, November 1997.
30. 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.
31. 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.
32. 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.
33. 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.

7 Appendix

In this section, we give the proofs of the type preservation theorem and the semantic-soundness theorem. Figure 7 gives the typing rules. Figure 8 gives a slightly modified version of the translation algorithm. The type environment Γ_m binds monomorphic variables while the environment Γ_p binds polymorphic variables.

Notation 1 ($\lambda^* F.e$ and $@^* zF$) *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 $@(..(@zx_1)..)x_n$.*

$$(const/var) \qquad \Gamma \vdash i : \text{Int} \qquad\qquad \Gamma \vdash x : \Gamma(x)$$

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

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

$$(tfn) \qquad \frac{\Gamma \vdash e_v : \mu_1 \quad \Gamma \uplus \{x : \forall \overline{t_i}.\mu_1\} \vdash e : \mu_2}{\Gamma \vdash \text{let } x = \Lambda \overline{t_i}.e_v \text{ in } e : \mu_2}$$

$$(tapp) \qquad \frac{\Gamma \vdash x : \forall \overline{t_i}.\mu}{\Gamma \vdash x[\overline{\mu_i}] : [\mu_i/t_i]\mu}$$

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

Fig. 7. Static Semantics

Notation 2 ($T(L)$) *If L is a set of variables, then $T(L)$ refers to the types of the variables in L in 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 $\mu_1 \to (... \to (\mu_n \to \tau)..)$.*

Throughout this section, we assume unique variable bindings – variables are never redefined in the program.

7.1 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 1 (The header type environment – 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 (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 — $\Gamma_m \vdash Let\ H\ in\ e : \mu$ iff $\Gamma_m; \Gamma_H \vdash e : \mu$.

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

$$(app) \qquad \frac{\Gamma_m(x_1) = \mu_1 \to \mu_2 \qquad \Gamma_m(x_2) = \mu_1}{\Gamma_m; \Gamma_p; H \vdash @x_1x_2 : \mu_2 \Rightarrow @x_1x_2; \emptyset; \{x_1, x_2\}}$$

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

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

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

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

Fig. 8. The Lifting Translation

Definition 3 (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 $\Gamma_{h_0 ... h_i} \vdash z : [\mu_i/t_i]\mu$

Lemma 1. *Suppose $\Gamma_m; \Gamma_p; H \vdash e \Rightarrow e'; H'; F$. If $x \in \Gamma_m$ and x does not occur free in H, then x does not occur free in $H + H'$.*

Proof. This is proved by induction on the structure of e.

Theorem 2 (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 induction on the structure of e. We will consider only tfn and *tapp*.

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

Since we assume H is well typed, we need to prove H' is well typed. By the precondition on the translation $\Gamma_H \vdash x : \forall \overline{t_i}.T(F) \rightarrow \mu$. Since F consists of the free variables of x, T(F) cannot have any of the $t_i's$ as a free type variable. Therefore $\Gamma_{H+H'} \vdash z : T(F) \rightarrow [\mu_i/t_i]\mu$ which proves that H' is well-typed. This also leads to $\Gamma_m; \Gamma_{H+H'} \vdash @^* zF : [\mu_i/t_i]\mu$.

Case tfn = To prove - given H is well-typed, $H + H_1 + H_2$ is also well-typed and $\Gamma_m; \Gamma_H \vdash Let\ H_1 + H_2\ in\ e_2' : \mu_2$.

By the inductive assumption on the translation of e_1, $H + H_1'$ is well-typed and $\Gamma_m; \Gamma_H \vdash Let\ H_1'\ in\ e_1' : \mu_1$. Since the variables in F_1 are bound in Γ_m (and not in H_1'), this implies that $\Gamma_m; \Gamma_H \vdash Let\ H_1'\ in\ \lambda^* F_1.e_1' : T(F_1) \rightarrow \mu_1$. Since $\lambda^* F_1.e_1'$ is closed with respect to monomorphic variables, we no longer require the environment Γ_m. Therefore $\Gamma_H \vdash Let\ H_1'\ in\ \lambda^* F_1.e_1' : T(F_1) \rightarrow \mu_1$. This implies H_1 is well-typed.

Again by induction, if $H + H_1$ is well-typed, then $H + H_1 + H_2$ is well-typed and $\Gamma_m; \Gamma_{H+H_1} \vdash Let\ H_2\ in\ e_2' : \mu_2$. This implies that $\Gamma_m; \Gamma_{H+H_1+H_2} \vdash e_2' : \mu_2$ which leads to the type preservation theorem.

□

7.2 Semantic Soundness

The operational semantics is shown in Figure 9.

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

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

Definition 4 (Type of a Value).

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

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

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

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

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

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

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

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

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

Fig. 9. Operational Semantics

We need to define the notion of equivalence of values before we can prove that two terms are semantically equivalent.

Definition 5 (Equivalence of Values).

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

Definition 6 (Equivalence of terms).
Suppose $a : \Gamma \vdash e \to v$ *and* $a' : \Gamma' \vdash e' \to v'$. *Then the terms* e *and* e' *are semantically equivalent iff* $v \approx v'$. *We denote this as* $a : \Gamma \vdash e \approx a' : \Gamma' \vdash e'$.

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 bindings in the environment as the header gets evaluated. Well-formedness of the header ensures that the lifting of polymorphic functions and type applications is semantically sound.

Definition 7 (The header value environment – a_H).

a_H is equal to $a_{h_0} \ldots a_{h_n}$ and a_{h_j} is –

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

Definition 8 (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. From the operational semantics we get $a_m : \Gamma_m \vdash Let\ H\ in\ e \approx a_m : \Gamma_m; a_H : \Gamma_H \vdash e$.

Definition 9 (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 –

- If $h_j ::= (x = \Lambda\overline{t_i}.e, \tau)$, and $\Gamma_p(x) = (\forall\overline{t_i}.\mu, F)$ then
  $a_m : \Gamma_m; a_p : \Gamma_p \vdash x[\overline{\mu_i}] \approx a_m : \Gamma_m; a_{h_0 \ldots h_j} : \Gamma_{h_0 \ldots h_j} \vdash let\ \ z = x[\overline{\mu_i}]\ \ in\ \ @^* zF$
- If $h_j ::= (z = x[\overline{\mu_i}], \tau)$, then h_j is well-formed.

H is well-formed w.r.t. $a_m : \Gamma_m; a_p : \Gamma_p$ will be abbreviated in this section to H is well-formed.

Theorem 3 (Semantic Soundness). *Suppose* $\Gamma_m; \Gamma_p; H \vdash e : \mu \Rightarrow e'; H_1; F$. *If* $a_m : \Gamma_m; a_p : \Gamma_p \vdash e \rightarrow v$ *and* H *is well-formed w.r.t* $a_m : \Gamma_m; a_p : \Gamma_p$ *then* $a_m : \Gamma_m; a_H : \Gamma_H \vdash Let\ H_1\ in\ e' \rightarrow v'$ *and* $v \approx v'$.

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

Case tapp = To prove – If H is well-formed then
$a_m : \Gamma_m; a_p : \Gamma_p \vdash x[\overline{\mu_i}] \approx a_m : \Gamma_m; a_H : \Gamma_H \vdash Let\ H_1\ in\ @^* zF$

Substituting $Let\ H_1$ in the above equation leads to
$a_m : \Gamma_m; a_p : \Gamma_p \vdash x[\overline{\mu_i}] \approx a_m : \Gamma_m; a_H : \Gamma_H \vdash let\ \ z = x[\overline{\mu_i}]\ \ in\ \ @^* zF$

By the precondition on the translation rule $\Gamma_p(x) = (\forall\overline{t_i}.\mu, F)$ and 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 therefore by definition

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

But since we assume unique variable bindings, no h_k for $k > j$ rebinds x. This leads to –

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

which is what we want to prove.

Case tfn = To prove - given H is well-formed
$a_m : \Gamma_m; a_p : \Gamma_p \vdash \text{let } x = \Lambda \overline{t_i}.e_1 \text{ in } e_2 \approx a_m : \Gamma_m; a_H : \Gamma_H \vdash Let\ H_1 + H_2\ in\ e_2'$

which means we must prove that if
$a_m : \Gamma_m; a_p[x \mapsto Clos^t\langle \overline{t_i}, e_1, a_m + a_p\rangle] : \Gamma_p[x \mapsto \langle \forall \overline{t_i}.\mu_1, F\rangle] \vdash e_2 \rightarrow v$
and $a_m : \Gamma_m; a_{H+H_1} : \Gamma_{H+H_1} \vdash Let\ H_2\ in\ e_2' \rightarrow v'$
then $v \approx v'$.

Assume for the time being that $H + H_1$ is well-formed. Then the inductive hypothesis on the translation of e_2 leads to the above condition.

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. According to the definition we need to prove that

$$a_m' : \Gamma_m'; a_p' : \Gamma_p' \vdash x[\overline{\mu_i}] \approx a_m' : \Gamma_m'; a_{H+H_1} : \Gamma_{H+H_1} \vdash let\ \ z = x[\overline{\mu_i}]\ \ in\ \ @^* zF$$

In the above equation $a_{H_1} := x \mapsto Clos^t\langle \overline{t_i}, Let\ H_1'\ in\ \lambda^* F.e_1', a_H\rangle$, therefore the operational semantics leads to $z \mapsto Clos\langle F^{T(F)}, e_1'[\mu_i/t_i], a_H + a_{H_1'[\mu_i/t_i]}\rangle$

This implies that we must prove –

$$a_m' : \Gamma_m'; a_p' : \Gamma_p' \vdash x[\overline{\mu_i}] \approx a_m'(F) : \Gamma_m'; a_H : \Gamma_H + a_{H_1'[\mu_i/t_i]} : \Gamma_{H_1'[\mu_i/t_i]} \vdash e_1'[\mu_i/t_i]$$

In the source term $x \mapsto Clos^t\langle \overline{t_i}, e_1, a_m + a_p\rangle$ which implies that

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

Therefore we need to prove that –

$$a_m : \Gamma_m; a_p : \Gamma_p \vdash e_1[\mu_i/t_i] \approx a_m'(F) : \Gamma_m'; a_H : \Gamma_H + a_{H_1'[\mu_i/t_i]} : \Gamma_{H_1'[\mu_i/t_i]} \vdash e_1'[\mu_i/t_i] \quad (1)$$

But $a_m'(F) = a_m(F)$ since variables are bound only once. F consists of all the free variables of e_1' that are bound in a_m' and therefore in a_m. Hence evaluating e_1' in $a_m(F)$ is equivalent to evaluating it in a_m . So proving Eqn 1 reduces to proving

$$a_m : \Gamma_m; a_p : \Gamma_p \vdash e_1[\mu_i/t_i] \approx a_m : \Gamma_m; a_H : \Gamma_H + a_{H_1'[\mu_i/t_i]} : \Gamma_{H_1'[\mu_i/t_i]} \vdash e_1'[\mu_i/t_i]$$

which follows from the inductive assumption on the translation of e_1.

□