Yale University
Department of Computer Science

Optimal Type Lifting
Bratin Saha       Zhong Shao
Dept. of Computer Science
Yale University

YALEU/DCS/TR-1159
August 24, 1998

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. A preliminary version of this paper appeared in the second International Workshop on Types in Compilation, March 1998.
Optimal Type Lifting

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

August 24, 1998

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

Recent compilers for ML-like polymorphic languages [25, 27] have begun to use variants of the Girard-Reynolds polymorphic λ-calculus [5, 22] as their intermediate language (IL). Implementation of these ILs often involves passing types explicitly as parameters [29, 28, 24] 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, 28, 23], which is used by some compilers [28, 24] 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 [29], pickling and marshalling [3], and type dynamic [13].

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

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. A preliminary version of this paper appeared in the second International Workshop on Types in Compilation, March 1998.
a program. For example, Tolmach [29] 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 [15]. 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[α] denotes type application and x(e) denotes term application.

\[
\text{pair} = \Lambda x.\lambda x:s:ss.
\]

\[
\begin{aligned}
&\text{let } f = \lambda t.\lambda y:t. \ldots (x, y) \\
&\text{in } \ldots f[s:ss](x) \ldots \\
\\
&\ldots \\
\end{aligned}
\]

\[
\text{main} = \Lambda a.\lambda a:a. \\
\text{let } doit = \lambda i:\text{Int}. \\
\text{let } elem = \text{Array}.\text{sub}[\alpha*\alpha](a,i) \\
\text{in } \ldots \text{pair}[\alpha](elem) \ldots \\
\text{loop} = \lambda n1:\text{Int}.\lambda n2:\text{Int}.\lambda g:\text{Int} \rightarrow \text{Unit}. \\
\text{if } n1 <\leq n2 \\
\quad (g(n1); \\
\quad \text{loop}(n1+1,n2,g)) \\
\text{else } () \\
\text{in } \text{loop}(1,n,doit)
\]

Here, \( f \) is a polymorphic function defined inside function \( \text{pair} \); it refers to the parameter \( x \) of \( \text{pair} \), so \( f \) cannot be easily lifted outside \( \text{pair} \). Function \( \text{main} \) executes a loop: in each iteration, it selects an element \( \text{elem} \) of the array \( a \) and then performs some computation (i.e., \( \text{pair} \)) on it. Executing the function \( \text{doit} \) results in three type applications arising from the \( \text{Array}.\text{sub} \) function, \( \text{pair} \), and \( f \). In each iteration, \( \text{sub} \) and \( \text{pair} \) are applied to types \( \alpha * \alpha \) and \( \alpha \) 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., \( \text{pair}[\alpha] \)). But optimizing type applications such as \( f[s:ss] \) is less obvious; \( f \) is nested inside \( \text{pair} \), and its repeated type applications are not apparent in the \( \text{doit} \) function (specially, if one considers that \( \text{pair} \) could be defined in a separate module). We may type-specialize \( f \) to get rid of the type application but in general this may lead to substantial code duplication. Every time \( \text{doit} \) is called, \( \text{pair}[\alpha] \) gets executed and then every time \( \text{pair} \) is called, \( f[s:ss] \) will be executed. Since \( \text{loop} \) calls \( \text{doit} \) repeatedly and each such call generates type applications of \( \text{pair} \) 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 allows us to use a more sophisticated representation 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 (\( \lambda \)); they are nested only inside type abstractions (\( \Lambda \)). All type applications are now performed once and for all at the beginning of execution of each compilation unit. In essence, the code performs all of its type applications at “link” time.\(^1\) In fact, the number of type applications

\(^1\)We are not referring to “link time” in the traditional sense. Rather, we are referring to the run time spent on module
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 does aim to reduce the runtime overhead, a more important goal is to ensure that type passing, in itself, is not costly. This 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 [24] 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.

\[
\begin{align*}
\text{(con's)} & \quad \mu & := & \text{t} \mid \text{Int} \mid \mu_1 \rightarrow \mu_2 \\
\text{(types)} & \quad \sigma & := & \mu \mid \forall t. \mu \\
\text{(terms)} & \quad e & := & i \mid x \mid \lambda x : \mu . e \mid \forall x_1 : x_2 . e \mid \text{let } x = e \text{ in } e' \mid \text{let } x = \Lambda \sigma . e_v \text{ in } e \mid x \mid [\mu]
\end{align*}
\]

Figure 1: Syntax of the 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 \(\bar{t}\) (and \(\bar{\sigma}\)) to denote a sequence of type variables \(t_1, \ldots, t_n\) (and type constructors) so \(\forall \bar{t}. \mu\) is equivalent to \(\forall t_1 \ldots \forall t_n. \mu\). The \(\text{vterms} (e_v)\) 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 [30]. Variables introduced by normal \(\lambda\)-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 which implies that polymorphic functions do not escape. Therefore in the algorithm in Figure 2, the \(\text{exp}\) rule assumes that the variable is monomorphic. The \(\text{tapp}\) 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 instantiated by further type application. Finally, following SML [14], polymorphic functions are not initialized and module linkage (e.g., functor application) in a ML-style module language.
recursive. 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.

![Diagram](image)

Figure 2: The Lifting Translation

We first 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(λ) 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. Therefore to avoid repeated type applications, the algorithm tries to lift all of them to depth zero. But in order 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 occurring at depths greater than zero and adds them to a list H. (In the algorithm given in Figure 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.

---

2Our current calculus does not support recursive functions but they can be easily added. As in SML, recursive functions are always monomorphic.
We will illustrate the algorithm on the sample code given in Section 1. In the example code, the type application \( f[s*s] \) is at depth 1 since it occurs inside the \( \lambda x \) (of the \( \text{pair} \) function), and the type applications \( \text{Array.sub}[\alpha*\alpha] \) and \( \text{pair}[\alpha] \) are at depth 2 since they occur inside the \( \lambda a \) and \( \lambda i \) (of the \( \text{main} \) function). We want to lift all of these type applications to depth zero. Translating \( \text{main} \) first, the resulting code becomes —

\[
\begin{align*}
\text{pair} &= \lambda s. \lambda x:s*s.s.
\end{align*}
\]

\[
\begin{align*}
\text{let } f &= \lambda t. \lambda y:t. \ldots (x, y) \\
\text{in } \ldots f[s*s](x) \ldots
\end{align*}
\]

\[
\begin{align*}
\ldots = \lambda \alpha.
\end{align*}
\]

\[
\begin{align*}
\text{let } v_1 &= \text{Array.sub}[\alpha*\alpha]
\end{align*}
\]

\[
\begin{align*}
\text{let } v_2 &= \text{pair}[\alpha]
\end{align*}
\]

\[
\begin{align*}
in \lambda a: \alpha.
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{doit} &= \lambda a: \text{Int}.
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{elem} &= v_1(a,i)
\end{align*}
\]

\[
\begin{align*}
in \ldots v_2(\text{elem}) \ldots
\end{align*}
\]

\[
\begin{align*}
\text{loop} &= \lambda n_1: \text{Int}. \lambda n_2: \text{Int}. \lambda g: \text{Int} \rightarrow \text{Unit}.
\end{align*}
\]

\[
\begin{align*}
\text{if } n_1 &= n_2 \\
(g(n_1));
\end{align*}
\]

\[
\begin{align*}
\text{loop}(n_1+1,n_2,g)
\end{align*}
\]

\[
\begin{align*}
\text{else } ()
\end{align*}
\]

\[
\begin{align*}
in \text{loop}(1,n,\text{doit})
\end{align*}
\]

We then lift the type application of \( f \) (inside \( \text{pair} \)). 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 \( v_1, v_2 \) and \( v_3 \) get evaluated now at the beginning of execution. When the function \( \text{loop} \) runs through the array and calls function \( \text{doit} \), none of the type applications are repeated — the type specialised functions \( v_1, v_2 \) and \( v_3 \) are used instead.

\[
\begin{align*}
\text{pair} &= \lambda s.
\end{align*}
\]

\[
\begin{align*}
\text{let } f &= \lambda t. \lambda z:s*s. \lambda y:t. \ldots (x, y)
\end{align*}
\]

\[
\begin{align*}
v_3 &= f[s*s]
\end{align*}
\]

\[
\begin{align*}
in \lambda x:s*s. \ldots (v_3(x))(x) \ldots
\end{align*}
\]

\[
\begin{align*}
\ldots = \lambda \alpha.
\end{align*}
\]

\[
\begin{align*}
\text{let } v_1 &= \text{Array.sub}[\alpha*\alpha]
\end{align*}
\]

\[
\begin{align*}
\text{let } v_2 &= \text{pair}[\alpha]
\end{align*}
\]

\[
\begin{align*}
in \lambda a: \alpha.
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{doit} &=
\end{align*}
\]

\[
\begin{align*}
\lambda i: \text{Int}.
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{elem} &= v_1(a,i)
\end{align*}
\]

\[
\begin{align*}
in \ldots v_2(\text{elem}) \ldots
\end{align*}
\]

\[
\begin{align*}
\text{loop} &= \lambda n_1: \text{Int}. \lambda n_2: \text{Int}. \lambda g: \text{Int} \rightarrow \text{Unit}.
\end{align*}
\]

\[
\begin{align*}
\text{if } n_1 &= n_2 \\
(g(n_1));
\end{align*}
\]

\[
\begin{align*}
\text{loop}(n_1+1,n_2,g)
\end{align*}
\]

\[
\begin{align*}
\text{else } ()
\end{align*}
\]

\[
\begin{align*}
in \text{loop}(1,n,\text{doit})
\end{align*}
\]
2.3 Formalization

Figure 2 shows the type-directed lifting algorithm. The translation is defined as a relation of the form $\Gamma \vdash e : \sigma \Rightarrow e'; H; F$, that carries the meaning that $\Gamma \vdash e : \sigma$ 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 monomorphically 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 $\mu$ is \textsc{let}(H, e') when the algorithm infers $\emptyset \vdash e : \mu \Rightarrow e'; H; \emptyset$. The function \textsc{let}(H, e) expands a list of bindings $H = [(x_1, e_1), \ldots, (x_n, e_n)]$ and a term $e$ into the resulting term \textsc{let} $x_1 = e_1 \ \text{in} \ldots \text{in} \ \text{let} \ x_n = e_n \ \text{in} \ e$.

The environment $\Gamma$ 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. The functions $\lambda^* \text{ and } @^*$ are defined so that $\lambda^*L.e \text{ and } @^*vL$ reduce to $\lambda x_1 : \mu_1 \ldots \lambda x_n : \mu_n.e$ and $\emptyset((@v x_1) \ldots )x_n$, respectively, where $L = \{x_1 : \mu_1, \ldots, x_n : \mu_n\}$. We use $::$ to denote the append operation and $::$ to denote the consing operation.

Rules (\textsc{exp}) and (\textsc{app}) are just the identity transformations. Rule (\textsc{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 \textsc{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 result header, $H_1$, is the concatenation of the headers, $H_1$ and $H_2$, formed during the translation of the subexpressions.

The real work is done in the last two rules which deal with type expressions. In rule (\textsc{fn}), we first translate the polymorphic function body. $H_1$ 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 \textsc{let} expression($e_2$). The result of the translation is only $e_2'$. The polymorphic function introduced by the \textsc{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 $F_1$ 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 get lifted to the top level.

The (\textsc{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, after the header is written out at the top level, the translated expression remains in the scope of the header.

**Proposition 1**

Suppose $\Gamma \vdash e : \mu \Rightarrow e'; H; F$. Then in the expression \textsc{let}(H, e'), the term $e'$ does not contain any type application and $H$ does not contain any type application nested inside a \textsc{value}(\lambda) abstraction.

**Proof** This is proved by induction on the structure of the source term $e$. Most of the cases follow directly from the inductive assumption. We will consider only the (\textsc{fn}) and (\textsc{tapp}) cases here.

- **case fn** By induction on the translation of $e_1$, the header $H_1$ does not have any nested type application and the term $e_1'$ does not have any type application. This implies that $H_r$ does not have any nested type application. By induction on the translation of $e_2$, the result term $e_2'$ does not have any type application and the header $H_2$ does not have any nested type application. The required result follows from here.

- **case tapp** In the translated term, the type application gets replaced by the newly introduced variable $v$; in
the result header, the type application is clearly not nested inside an abstraction.

\[\square\]

**Theorem 2 (Full Lifting)**

Suppose \( \Gamma \vdash e : \mu \Rightarrow e'; H; F \). Then the expression \( \text{LET}(H, e') \), does not have any type application nested inside a value abstraction.

**Proof** The theorem follows from Proposition 1.

\[\square\]

### 2.4 An Example Transformation

This section illustrates the algorithm on a code 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 Figure 2. The program fragment used for the example is shown below.

\[
\Lambda t_1. \lambda t_2. \lambda x: t_1. \lambda y: t_2. \\
\text{let } f = \Lambda t_3. \lambda t_4. \lambda v: t_3. \lambda w: t_4. (v, u, x) \\
\text{in let } g = \Lambda t_5. \lambda z: t_5. \Theta (\Theta (f[t_2][t_1]) z) x \\
\text{in } \Theta (g[t_3]) y
\]

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

1. After translating \( f \)'s body

\[
e'_{1} = \lambda u: t_3. \lambda v: t_4. (v, u, x) \\
H_1 = [] \\
F_1 = \{x: t_1\}
\]

2. Now \( g \)'s body is translated

\[
e'_{1} = \lambda z: t_5. \Theta (\Theta (f[t_2][t_1]) z) x \\
H_1 = v_1 = f[t_2][t_1] \\
F_1 = \{x: t_1\}
\]

3. Now the body of the inner let

\[
e'_{2} = \Theta (\Theta v_2 x) y \\
H_2 = v_2 = g[t_2] \\
F_2 = \{x: t_1, y: t_2\}
\]

4. The inner let as a whole returns

\[
e' = \Theta (\Theta v_2 x) y \\
H = g = (\Lambda t_5. \\
\text{let } v_1 = f[t_2][t_1] \\
\text{in } \lambda x: t_1. \lambda z: t_5. \Theta (\Theta (v_1 x) z) x ) :: [v_2 = g[t_2]] \\
F = \{x: t_1, y: t_2\}
\]
5. For the outer let

\[ e' = \theta_1(v_{2x})y \]
\[ H_2 = g = (\; \lambda t_0. \]
\[ \text{let } v_1 = f[t_0][t_1] \]
\[ \text{in } \lambda x : t_1. \lambda z : t_0. \theta_1(\theta_1 z)z \quad :: \quad v_2 = g[t_2] \]
\[ F_2 = \{ x : t_1, y : t_2 \} \]

6. The outer let as a whole returns

\[ e' = \theta_1(v_{2x})y \]
\[ H = f = (\; \lambda t_3. \lambda t_4. \lambda z : t_1. \lambda u : t_3. \lambda v : t_4. (v, u, x) \quad :: \quad g = (\; \lambda t_0. \]
\[ \text{let } v_1 = f[t_0][t_1] \]
\[ \text{in } \lambda x : t_1. \lambda z : t_0. \theta_1(\theta_1 z)z \quad :: \quad v_2 = g[t_2] \]
\[ F = \{ \} \]

7. After translating the lambda abstraction

\[ e' = \theta_1(v_{2x})y \]
\[ H = f = (\; \lambda t_3. \lambda t_4. \lambda z : t_1. \lambda u : t_3. \lambda v : t_4. (v, u, x) \quad :: \quad g = (\; \lambda t_0. \]
\[ \text{let } v_1 = f[t_0][t_1] \]
\[ \text{in } \lambda x : t_1. \lambda z : t_0. \theta_1(\theta_1 z)z \quad :: \quad v_2 = g[t_2] \]
\[ F = \{ \} \]

The final translated code with all type applications lifted:

\[ \lambda t_1. \lambda t_2. \]
\[ \text{let } f = \lambda t_3. \lambda t_4. \lambda z : t_1. \lambda u : t_3. \lambda v : t_4. (v, u, x) \]
\[ \text{let } g = \lambda t_0. \]
\[ \text{let } v_1 = f[t_0][t_1] \]
\[ \text{in } \lambda x : t_1. \lambda z : t_0. \theta_1(\theta_1 z)z \]
\[ \text{let } v_2 = g[t_2] \]
\[ \text{in } \lambda x : t_1. \lambda y : t_2. \theta_1(\theta_1 y) \]

3 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 SML module language [14] as well?

To handle the Full-ML language, we compile the source code into the FLINT intermediate language 3. The details of the translation are given in [26]. FLINT is based upon a predicative variant of the Girard-Reynolds polymorphic \( \lambda \)-calculus [5, 22], with the term language written in A-normal form [4]. The monotypes are generated from variables, from Int, and through the \( \rightarrow \) constructor. 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 \( \mu \). The terms are an explicitly typed \( \lambda \)-calculus (but in A-normal form) with explicit constructor abstraction and application.
\[
\begin{align*}
\text{(cons)} \quad \mu & := t \mid \text{Int} \mid \mu_1 \rightarrow \mu_2 \\
\text{(types)} \quad \sigma & := T(\mu) \mid \sigma_1 \rightarrow \sigma_2 \mid \forall t.\sigma \\
\text{(terms)} \quad e & := i \mid x \mid \text{let} \ x = e_1 \ \text{in} \ e_2 \mid \otimes x_1 \ x_2 \mid \lambda^c x : T(\mu). e \mid \lambda^m x : \sigma. e \mid \text{let} \ x = \Lambda t_i e_v \ \text{in} \ e_2 \mid x[\pi_i]
\end{align*}
\]

Figure 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 instantiated later at a functor application site. In the FLINT translation [26], functors are represented as a combined type and value abstraction \((fct = \Lambda t_i : t_i, \lambda^m x : \sigma.e)\). 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 \(x\) may be instantiated by type application. Functor application itself consists of a type application and a term application [26], 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 allow for escaping polymorphic functions.

However, the ML 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 (and therefore the corresponding curried type applications) are also not nested inside Core-ML functions. This leads to the notion of a well-formed FLINT program (Figure 4), satisfying the following constraints —

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

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 occurring outside all function abstractions only. Note that depth 0 terms may not occur outside all abstractions since they may be nested inside functor abstractions. As before, we perform type lifting only for terms at depth greater than zero.

We illustrate the algorithm on the example code in Figure 5. 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 5, \(F\) is a functor which takes the structure \(X\) as a parameter. The type \(S\) denotes a structure type. Assume the first component of \(X\), that is \((\#1(X))\), is a polymorphic function which gets instantiated in the functor body (and gets bound to \(v_2\)). \(f\) is a locally defined function in the functor body. According to the above definition of depth, \(f\) and \(v_2\) are at depth 0 even though they are nested inside the functor abstraction \((\lambda X)\). Moreover, the type application \((\#1(X))[t_0]\) is also at depth 0. 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 \((\lambda v)\), it is not lifted outside the functor abstraction \((\lambda X)\). The resulting code is shown in Figure 6.
(int) \[ \Gamma; d \vdash w \ i : \text{Int} \]

(var) \[ \Gamma; d \vdash w \ x : T(\mu) \]

(app1) \[ \frac{\Gamma(x_1) = \sigma_1 \rightarrow \sigma_2 \quad \Gamma(x_2) = \sigma_1}{\Gamma; d = 0 \vdash w \ @x_1 x_2 : \sigma_2} \]

(app2) \[ \frac{\Gamma(x_1) = T(\mu_1 \rightarrow \mu_2) \quad \Gamma(x_2) = T(\mu_1)}{\Gamma; d = 0 \vdash w \ @x_1 x_2 : T(\mu_2)} \]

(fn) \[ \frac{\Gamma[x \mapsto T(\mu)]; d + 1 \vdash w \ e : T(\mu')}{\Gamma; d \vdash w \ \lambda^c x : T(\mu).e : T(\mu \rightarrow \mu')} \]

(fct) \[ \frac{\Gamma[x \mapsto \sigma]; d = 0 \vdash w \ e : \sigma'}{\Gamma; d = 0 \vdash w \ \lambda^m x : \sigma.e : \sigma \rightarrow \sigma'} \]

(tfn) \[ \frac{\Gamma; d \vdash w \ e_v : \sigma \quad \Gamma[x \mapsto \forall i.\sigma]; d \vdash w \ e' : \sigma']}{\Gamma; d \vdash w \ \text{let } x = \Lambda_i e_v \ \text{in } e' : \sigma'} \]

(tapp) \[ \frac{\Gamma(x) = \forall i.\sigma \quad j < i}{\Gamma; d \vdash w \ x[\overline{m}_j] : \sigma[\mu_j/t_i]} \]

(let1) \[ \frac{\Gamma; d = 0 \vdash w \ e_1 : \sigma_1 \quad \Gamma[x \mapsto \sigma_1]; d = 0 \vdash w \ e_2 : \sigma_2}{\Gamma; d = 0 \vdash w \ \text{let } x = e_1 \ \text{in } e_2 : \sigma_2} \]

(let2) \[ \frac{\Gamma; d \vdash w \ e_1 : T(\mu_1) \quad \Gamma[x \mapsto T(\mu_1)]; d \vdash w \ e_2 : \sigma_2}{\Gamma; d \vdash w \ \text{let } x = e_1 \ \text{in } e_2 : \sigma_2} \]

Figure 4: The Well Formedness Relation \( \vdash_w \)

\[
F = \Lambda t_0. \lambda^m x : S.
\]

\[ f = \lambda^c v. \]

\[ \text{let } \text{id} = \Lambda t_1. \lambda^c x_2. x_2 \]

\[ \ldots \]

\[ \nu_1 = \ldots \text{id[Int](3)} \ldots \]

\[ \text{in } \nu_1 \]

\[ \nu_2 = (\#1(\chi))[t_0] \]

\[ \ldots \]

Figure 5: Example FLINT code
\[ F = \text{let } z_1 = \text{id}[^0\text{Int}]
\quad \text{in } \text{let } \ldots \text{ (type lifted body of } f) \ldots \text{ in } v_1
\quad \text{v}_2 = (\#1(x))[t_0]
\ldots \]

Figure 6: FLINT code after type lifting

The algorithm is shown in Figure 7. It does not assume that the input program is well formed. The translation rules are expressed as sequents of the form

\[ \Gamma; d \vdash e : \sigma \Rightarrow e' ; H ; F \]

Here \( e \) is the input term; \( e' \) is the output term; \( d \) denotes the depth of the term \( e \); \( H \) is as defined before, it contains the list of type expressions in \( e \) that will be lifted; \( F \) consists of the monomorphic free variables of \( e' \); \( \Gamma \) is the type environment that maps a variable to its type, and the free variables in its definition.

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

The \textit{(tapp)} rule deals with both partial and full type applications. (for full type applications, \( k = 0 \)). If \( d = 0 \), the type application is included in the header and bound to \( v \) at \( d = 0 \). When the header is dumped at \( d = 0 \), the type application gets bound to \( v \). The set \( F \) contains the free variables in the definition of \( x \).

The first \textit{(tfn)} rule deals with polymorphic functions at depth greater than zero. We close the function body by abstracting over its free variables, (only the monomorphically typed variables), and add it to the header. As in the Core-ML case, the result of the translation is simply the translation of \( e_2 \). The second \textit{(tfn2)} rule deals with polymorphic functions at \( d = 0 \). Since the function definition is already at the top level, the algorithm simply translates each subexpression.

Note that at \( d = 0 \), the header formed is empty. The algorithm adds expressions to the header only at \( d > 0 \). The \textit{(fct)} rule deals with functors. Since the algorithm restricts itself to lifting expressions only outside functions, (type applications are allowed to remain nested inside functors), the depth is reset to zero while translating the functor body.

The \textit{(fn)} rule deals with functions. The algorithm first collects the type applications and polymorphic functions occurring in the body. When the algorithm returns to depth zero, it writes out the header. This ensures that while translating a term at depth zero, the header returned is always empty. This invariant is useful in formulating the algorithm and in the proofs of type preservation and semantic soundness.

In the algorithm, we donot abstract over polymorphic variables. (Hence we donot add them to the free variable list). \textit{let} introduced polymorphic variables are lifted to the top, and therefore, remain in scope. Polymorphic variables introduced by functors remain in scope as well, since no expressions are lifted outside
(int)
\[ \Gamma; d \vdash i : \text{Int} \Rightarrow i \ \text{nil} ; \emptyset \]

(var)
\[ \Gamma; d \vdash x : T(\mu) \Rightarrow x \ \text{nil} ; \{x : T(\mu)\} \quad \Gamma; d \vdash x : \sigma \Rightarrow x \ \text{nil} ; \emptyset \]

(app)
\[ \Gamma; d \vdash x_1 : T(\mu \rightarrow \mu_1) \Rightarrow x_1 \ \text{nil} ; \{x_1 : T(\mu \rightarrow \mu_1)\} \quad \Gamma; d \vdash x_2 : T(\mu) \Rightarrow x_2 \ \text{nil} ; \{x_2 : T(\mu)\} \\
\quad \Gamma; d \vdash @x_1 x_2 : T(\mu) \Rightarrow @x_1 x_2 \ \text{nil} ; \{x_1 : T(\mu \rightarrow \mu_1), x_2 : T(\mu)\} \]

(app2)
\[ \Gamma; d \vdash x_1 : \sigma_1 \Rightarrow x_1 \ \text{nil} ; \emptyset \quad \Gamma; d \vdash x_2 : \sigma_2 \Rightarrow x_2 \ \text{nil} ; \emptyset \\
\quad \Gamma; d \vdash @x_1 x_2 : \sigma_1 \Rightarrow @x_1 x_2 \ \text{nil} ; \emptyset \]

(let)
\[ \Gamma; d \vdash e_1 : T(\mu) \Rightarrow e'_1 ; H_1 ; F_1 \quad \Gamma[x \mapsto (T(\mu), \text{nil})]; d \vdash e_2 : \sigma_2 \Rightarrow e'_2 ; H_2 ; F_2 \quad \Gamma; d \vdash \text{let} \ x = e_1 \ \text{in} \ e_2 : \sigma_2 \Rightarrow \text{let} \ x = e'_1 \ \text{in} \ e'_2 ; H_1 \parallel H_2 ; F_1 \cup F_2 \setminus \{x : T(\mu)\} \]

(let2)
\[ \Gamma; d \vdash e_1 : \sigma_1 \Rightarrow e'_1 ; H_1 ; F_1 \quad \Gamma[x \mapsto (\sigma_1, \text{nil})]; d \vdash e_2 : \sigma_2 \Rightarrow e'_2 ; H_2 ; F_2 \quad \Gamma; d \vdash \text{let} \ x = e_1 \ \text{in} \ e_2 : \sigma_2 \Rightarrow \text{let} \ x = e'_1 \ \text{in} \ \text{LET}(H_2, e'_2) ; H_1 ; F_1 \cup F_2 \]

(tfn)
\[ \Gamma; d \vdash e_1 : \sigma_1 \Rightarrow e'_1 ; H_1 ; F_1 \quad \Gamma[x \mapsto (\forall \sigma_1, \sigma_1, \text{nil})]; d \vdash e_2 : \sigma_2 \Rightarrow e'_2 ; H_2 ; F_2 \quad H' = \{x = \Lambda \sigma_1 . \text{LET}(H_1, \lambda \sigma_1 . e'_1)\} \]

(tfn2)
\[ \Gamma; d = 0 \vdash e_1 : \sigma_1 \Rightarrow e'_1 ; H_1 ; F_1 \quad \Gamma[x \mapsto (\forall \sigma_1, \sigma_1, \text{nil})]; d = 0 \vdash e_2 : \sigma_2 \Rightarrow e'_2 ; H_2 ; F_2 \quad H_2 = \text{nil} \quad \Gamma; d = 0 \vdash \text{let} \ x = \Lambda \sigma_1 . e_1 \ \text{in} \ e_2 : \sigma_2 \Rightarrow \text{let} \ x = \Lambda \sigma_1 . e'_1 \ \text{in} \ e'_2 ; \text{nil} ; F_1 \cup F_2 \]

(tapp)
\[ \Gamma; d > 0 \vdash x[Z_1] ; \mu_j/t_j \forall \sigma_k . \Rightarrow \text{let} \ x = \Lambda \sigma_k . \forall (v[Z_k])F \ \text{in} \ x ; \mu = x[Z_1] ; F \quad \Gamma; d = 0 \vdash x[Z_1] ; \mu_j/t_j \forall \sigma_k . \Rightarrow \text{let} \ x = \Lambda \sigma_k . \forall (x[Z_1][Z_k])F \ \text{in} \ x ; \text{nil} ; F \]

(fct)
\[ \Gamma[x \mapsto (\sigma, \theta_f)]; d = 0 \vdash e : \sigma' \Rightarrow e' ; H ; F \quad H = \text{nil} \quad \Gamma; d \vdash \lambda^m x : \sigma.e : \sigma' \Rightarrow \lambda^m x : \sigma.e' ; \text{nil} ; F \]

(fn)
\[ \Gamma; d > 0 \vdash \lambda^c x : T(\mu), e : T(\mu) \Rightarrow e' ; H ; F \quad \Gamma; d = 0 \vdash \lambda^c x : T(\mu); e : T(\mu) \Rightarrow \text{LET}(H, \lambda^c x : T(\mu).e') ; \text{nil} ; F \setminus \{x : T(\mu)\} \]

Figure 7: The Lifting Algorithm For FLINT
functor abstractions.

**Proposition 3**

In a well formed expression $e$, terms at $d = 0$ occur outside all Core-ML functions ($\lambda^c$).

**Proof** In Figure 4, the depth is zero either initially, before any ($\lambda^c$) is encountered, or is reset to zero by the (fct) rule. Assume that the above proposition does not hold. Then there exists a term $e'$, such that, $e'$ is nested inside a ($\lambda^c$), but the depth of $e'$ is zero. This is possible only if an occurrence of rule (fct) inside the ($\lambda^c$) resets the depth to zero. But this implies an occurrence of the (fct) rule at $(d > 0)$, and since $e$ is well formed, this cannot happen. Therefore the above proposition is true.

**Proposition 4**

Suppose $\Gamma; d \vdash e : \sigma = e'; H; F$ and $e$ is a well formed expression. If $x$ has type $\forall i. \sigma$ and $x[j]$ occurs in $e$ at $d > 0$, then $j = i$.

**Proof** Since $e$ is well formed, every subterm must satisfy the constraints in Figure 4. Rule (tapp) restricts partial type applications to $d = 0$. Therefore the above proposition must be true.

**Proposition 5**

Suppose $\Gamma; d \vdash e : \sigma \Rightarrow e'; H; F$. Suppose further that $e$ is well formed and contains the subterm $\text{let } x = e_1 \text{ in } e_2$ at some depth $d'$. Then $x$ must be of type $T(\mu)$ if $d' > 0$.

**Proof** Since $e$ is well formed, every subterm must satisfy the constraints in Figure 4. Therefore, if $x$ is of type $\sigma$, the term $\text{let } x = e_1 \text{ in } e_2$ must satisfy rule (letI) which restricts the Let definition to $d = 0$.

**Lemma 6**

Suppose $\Gamma; d \vdash e : \sigma \Rightarrow e'; H; F$. Suppose further that $e$ is well formed. Then the term $e'$ does not contain any type application if $d > 0$.

**Proof** This is proved by induction over the structure of $e$. We will consider only the cases that do not follow directly from the inductive assumption.

- **case let2** Since $e$ is well formed and $d > 0$, by Proposition 5, this case does not arise.

- **case tfn** Since $e$ is well formed, $e_2$ must be well formed as well. By induction on the translation of $e_2$, the result term $e'_2$ does not contain any type applications.

- **case tapp** By Proposition 4, $k = 0$ for $d > 0$. Therefore the result term reduces to $\@^*\nu F$.

- **case fct** Since $e$ is well formed and $d > 0$, from Figure 4, rule (fct), this case does not arise.

□
Lemma 7

Suppose \( \Gamma; d \vdash e : \sigma \Rightarrow e' \); \( H \); \( F \) and \( e \) is well formed. Then both \( H \) and \( e' \) do not contain any type application nested inside a function abstraction \((\lambda^c)\).

**Proof** The proof is by induction on the structure of \( e \). We consider only the cases that do not follow directly from the inductive hypothesis.

- **case tfn** By Lemma 6, \( e'_1 \) does not contain any type applications. By induction, \( H_1 \) does not contain type applications nested inside a \((\lambda^c)\). Therefore \( H' \) does not contain any type application nested inside a \((\lambda^c)\). By induction on \( e_2 \), both \( H_2 \) and \( e'_2 \) do not contain nested type application; the lemma follows from this.

- **case tfn2** The result follows directly from the inductive assumptions on the translation of \( e_1 \) and \( e_2 \).

- **case tapp** The lemma is clearly satisfied for this case since the type applications in either the header or the translated term are not nested inside a \((\lambda^c)\).

- **case fc** Since \( e \) is well formed, this rule occurs only for \( d = 0 \). Therefore, the inductive assumption leads directly to the lemma.

- **case fn** By Lemma 6, \( e' \) does not contain any type applications. Therefore the term \( \lambda^c x : T(\mu).e' \) does not contain any type application nested inside a \((\lambda^c)\). By induction on the translation of \( e \), the header \( H \) does not contain any nested type application; the lemma follows from this.

\( \square \)

Theorem 8 (Full Lifting)

Suppose \( \Gamma; d \vdash e : \sigma \Rightarrow e' \); \( H \); \( F \). Suppose further that \( e \) is well formed. Then the expression \( \text{LET}(H, e') \) does not have a type application nested inside a function abstraction \((\lambda^c)\).

**Proof** The theorem follows from Lemma 7.

\( \square \)

In the Appendix, we prove the type preservation property and the semantic soundness of the algorithm.

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 part of the 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 only once 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 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 1.7 and the SML/NJ compiler v110.7. All the tests were performed on a Pentium Pro 200 Linux workstation with 64M physical
RAM. Figure 8 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.

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

Figure 8: 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 4% for the polymorphic benchmarks. Simple has a lot of polymorphic function calls occurring inside loops and therefore benefits greatly from lifting. Life, (after specialisation), and mandelbrot are monomorphic benchmarks (involving large lists) and predictably do not benefit from the optimization.

What are the tradeoffs involved in the algorithm? 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 the lifting algorithm introduces new variables 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. 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 simplifies the implementation considerably by reducing the amount of 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 need to dump only those terms (in $H_1$) which contain any of the $\ell_i$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. Therefore, 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
ensures that the number of variables abstracted while lifting a definition is kept to a minimum – by recording the depth at which a variable is defined.

Our algorithm is a source-to-source transformation and the output 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.

5 Related Work

The algorithm performs two transformations 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 consisting only of global function definitions whereas the lifting shown here preserves the nesting structure of the program.

Consider the program fragment given below. (assuming $f$ is externally defined)

$$\Lambda \alpha \beta \gamma. \lambda x_1.
\begin{align*}
&\text{let} \\
&y_1 = \Lambda t_1. \lambda x_2. \Theta(f[t_1*\alpha])x_2 \\
&y_2 = \Lambda \alpha t_2. \lambda x_3. \Theta(y_1[\alpha][\beta*t_3])x_3 \\
&y_3 = \Lambda \alpha \beta \gamma t_4. \lambda x_4. \Theta(y_2[\alpha][\beta][\gamma*t_4])x_4 \\
&\text{expr} = \Lambda \alpha \beta \gamma. \lambda x_1. \Theta(y_3[\alpha][\beta][\gamma][\beta*\gamma])x_1
\end{align*}$$

Johnsson-style lambda-lifting [10] converts this into:

$$\begin{align*}
&y_1 = \Lambda \alpha t_1. \lambda x_2. \Theta(f[t_1*\alpha])x_2 \\
&y_2 = \Lambda \alpha \beta \gamma t_3. \lambda x_3. \Theta(y_1[\alpha][\beta*\gamma])x_3 \\
&y_3 = \Lambda \alpha \beta \gamma t_4. \lambda x_4. \Theta(y_2[\alpha][\beta][\gamma*t_4])x_4 \\
&\text{expr} = \Lambda \alpha \beta \gamma. \lambda x_1. \Theta(y_3[\alpha][\beta][\gamma][\beta*\gamma])x_1
\end{align*}$$

But we translate this into (ignoring the lifting of type applications for the time being as it is not relevant to our argument):

$$\Lambda \alpha \beta \gamma.
\begin{align*}
&\text{let} \\
&y_1 = \Lambda t_1.
&\text{let } v_1 = f[t_1*\alpha] \\
&\text{in } \lambda x_2. \Theta v_1 x_2 \\
&y_2 = \Lambda t_2.
&\text{let } v_2 = y_1[\beta*t_3] \\
&\text{in } \lambda x_3. \Theta v_2 x_3 \\
&y_3 = \Lambda t_3.
&\text{let } v_3 = y_2[\gamma*t_4] \\
&\text{in } \lambda x_4. \Theta v_3 x_4 \\
&v_4 = y_3[\beta*\gamma] \\
&\text{in } \lambda x_1. \Theta v_4 x_1
\end{align*}$$

Retaining the lexical nesting of the program has some important implications. The conventional lambda lifting has to abstract over the free type variables of the function definition. As nested functions are lifted, the number
of free type variables may start increasing in a cascading manner (when the set of lifted functions have distinct free type variables). In our case, type abstractions are never introduced.

The lifting of type applications is similar in spirit to the hoisting of loop invariant expressions. It could be considered as a special case of a fully lazy transformation [9, 20] with the maximal free subexpressions restricted to be type applications. However, the fully-lazy transformation as described in Peyton Jones [20] will not lift all type applications to the top level. Specifically, type applications of a polymorphic function 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.

$$\Lambda \alpha \beta \gamma .$$
let $$u = \beta \gamma$$
$$\lambda x_1 .$$
  let
  $$y_1 = \Lambda t_1 .$$
  let $$u_1 = f[ t_1 \alpha ]$$
  in $$\lambda x_2 . \ u_1 x_2 \ldots$$
  $$y_2 = \Lambda t_3 .$$
  let $$u_2 = y_1 [ \beta t_3 ]$$
  in $$\lambda x_3 . \ u_2 x_3 \ldots$$
  $$y_3 = \Lambda t_5 .$$
  let $$u_3 = y_2 [ \gamma t_5 ]$$
  in $$\lambda x_4 . \ u_3 x_4 \ldots$$
in $$\Theta(y_3[u])x_1$$

Minamide [15] has also worked on the same problem but uses an entirely different approach from ours. Instead of constructing types inside functions, he constructs them at the call sites and passes them in as parameters. This transformation is recursively propagated to the call sites at the top level. However, this increases the number of type parameters of a polymorphic function since all the types that were previously being constructed inside functions are now passed in as parameters. He therefore passes in a record of types and replaces type construction by projections from this record. As an example, consider his transformation on the code fragment shown at the beginning of this subsection. In the example code, 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 $$u$$.

$$\Lambda u : pr_0 . \lambda x_1 .$$
let
  $$y_1 = \Lambda u : pr_1 . \lambda x_2 . \ \Theta( f[ \# 2(u) ]) x_2 \ldots$$
  $$y_2 = \Lambda u : pr_2 . \lambda x_3 . \ \Theta( y_1[ \# 2(u) ]) x_3 \ldots$$
  $$y_3 = \Lambda u : pr_3 . \lambda x_4 . \ \Theta( y_2[ \# 2(u) ]) x_4 \ldots$$
in $$\Theta(y_3[ \# 4(u) ]) x_1$$

$$pr_0 = \{ \alpha, \beta, \gamma, \{ \beta \gamma, \{ \beta \gamma, \beta \gamma, \beta \gamma, \beta \gamma \alpha \} \} \}$$
$$pr_1 = \{ t_1, t_1 \alpha \}$$
$$pr_2 = \{ t_2, \{ t_2, t_2 \beta t_3 \} \}$$
$$pr_3 = \{ t_3, \{ t_3, \{ t_3, t_3, t_3 \beta \gamma t_5 \} \} \}$$

The advantage of his method is that he eliminates the runtime construction of types. However, the disadvantage is that he can no longer type-check his transformation with the existing type system; instead, he
has to use an auxiliary type system based on the qualified type system of Jones [12], and the implementation calculus for the compilation of polymorphic records of Ohori [17]. Our algorithm on the other hand is a source-to-source transformation. 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. On the other hand, 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 the implementation of type classes in Haskell [8] and Gofer [11]. Type classes in these languages are implemented by dictionary passing and if done naively, can lead to the same dictionaries being created repeatedly.

We will briefly compare our approach with his optimisations on dictionary passing. Since the type systems and the implementation of dictionaries differs slightly in Haskell and Gofer, we will consider the two separately.

Haskell [8] performs context reduction and simplifies the set of constraints in a type. Consider the following Haskell example

\[
\begin{align*}
f :: & \text{Eq a} \Rightarrow a \rightarrow a \rightarrow \text{Bool} \\
f \ x \ y \ = \ ([x]) \ == \ [y] \ \&\& \ ([y]) \ == \ [x]
\end{align*}
\]

The actual type of \(f\) is \(\text{Eq a} \Rightarrow a \rightarrow a \rightarrow \text{Bool}\), but after context reduction, we get the type shown in the example. Here \([a]\) means a list of elements of type \(a\). \(\text{Eq a}\) means that the type \(a\) must be an instance of the Equality Class. \(\text{Eq a}\) means that the type \(\text{List of } a's\) must be an instance of the Equality Class. Function \(f\) in the example above, has type \(a \rightarrow a \rightarrow \text{Bool}\), with \(a\) being an instance of the Equality class. Jones optimises this by constructing a dictionary for \(\text{Eq a}\) at the call site of \(f\); rather than pass a dictionary for \(\text{Eq a}\) and construct the dictionary for \(\text{Eq a}\) in the function \(f\). He repeats this for all overloaded functions so that all dictionaries are created statically. But this approach does not work with separately compiled modules since the type of \(f\) being exported to other modules does not specify the dictionaries that are constructed inside it.

In Gofer [11], however, instance declarations are not used to simplify the context. Therefore the type of \(f\) in the above example would still be \(\text{Eq a} \Rightarrow a \rightarrow a \rightarrow \text{Bool}\). Jones’ optimisation can now be performed even in the presence of separately compiled modules. In fact, Minamide’s transformation is very similar to this.

However, the ML module language (which we considered in Section 3) supports functors that get translated into polymorphic abstractions – meaning thereby, that the abstracted variable is polymorphic. Suppose two polymorphic functions \(f\) and \(g\) have the same type\((\sigma_1)\) but the types constructed in their bodies are different. 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. Suppose we had a function \((h = \lambda x : \sigma_1.e)\). Previously, we could 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.

Tolmach [29] 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’s [15] method, in fact, is a refinement of Tolmach’s method to eliminate runtime construction of type parameters. Peyton Jones [21, 19, 20] 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.

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. [16] in the implementation of Napier. The work of Ohori on compiling record operations [17] 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 Morrisett [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. Many modern compilers like the FLINT/ML compiler [24], TIL [28] and the Glasgow Haskell compiler [18] use an explicitly typed language as the intermediate language in the compilation.

6 Conclusions

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. The method we have proposed can also be used in optimising dictionary passing in Haskell. However, since Haskell supports polymorphic recursion, we cannot guarantee that all dictionaries will be constructed statically. We are now working towards a more comprehensive runtime type-representation in FLINT; so that we can maintain complete type information, yet, not incur a significant penalty at runtime.

7 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.

References


A Proofs For The Core-ML Algorithm

In this section, we prove the semantic soundness and the type preservation property of the algorithm.

\[
\begin{align*}
\text{(exp)} & \quad \Gamma_m(x) = \mu \\
& \quad \frac{\Gamma_m; \Gamma_i; H \vdash x : \mu \Rightarrow x; \text{nil}; \{x : \mu\}}{\Gamma_m; \Gamma_i; H \vdash i : \text{Int} \Rightarrow i; \text{nil}; \emptyset}
\end{align*}
\]

\[
\begin{align*}
\text{(app)} & \quad \Gamma_m(x_1) = \mu_1 \rightarrow \mu_2 \\
& \quad \frac{\Gamma_m; \Gamma_i; H \vdash @x_1 x_2 : \mu_2 \Rightarrow @x_1 x_2; \text{nil}; \{x_1 : \mu_1 \rightarrow \mu_2, x_2 : \mu_1\}}{\Gamma_m; \Gamma_i; H \vdash e : \mu \Rightarrow e'; H'; F}
\end{align*}
\]

\[
\begin{align*}
\text{(fn)} & \quad \Gamma_m[x \mapsto \mu]; \Gamma_i; H \vdash e : \mu \Rightarrow e'; H'; F \\
& \quad \frac{\Gamma_m; \Gamma_i; H \vdash \lambda x : \mu.e : \mu \rightarrow \mu' \Rightarrow \lambda x : \mu.e' ; H'; F \setminus \{x : \mu\}}{\Gamma_m; \Gamma_i; H \vdash e \in e_2 : \mu_2 \Rightarrow e'_2 ; H_2 ; F_2}
\end{align*}
\]

\[
\begin{align*}
\text{(let)} & \quad \Gamma_m; \Gamma_i; H \vdash e_1 : \mu_1 \Rightarrow e'_1 ; H_1 ; F_1 \\
& \quad \frac{\Gamma_m; \Gamma_i; H \vdash e_2 : \mu_2 \Rightarrow e'_2 ; H_2 ; F_2}{\Gamma_m; \Gamma_i; H \vdash \text{let } x = e_1 \text{ in } e_2 ; \mu_2 \Rightarrow e'_2 ; H' \| H_2 ; F_1 \cup (F_2 \setminus \{x : \mu_1\})}
\end{align*}
\]

\[
\begin{align*}
\text{(tfn)} & \quad \Gamma_m; \Gamma_i; H \vdash \text{let } x = \Lambda_f e_1 \text{ in } e_2 ; \mu_2 \Rightarrow e'_2 ; H' \| H_2 ; F_2 \\
& \quad \frac{\Gamma_m; \Gamma_i; H \vdash \Lambda_f e_1}{\Gamma_m; \Gamma_i; H \vdash (\forall x. \mu_i. F) ; h \mid h \vdash z \in \mu_i} ; F
\end{align*}
\]

\[
\begin{align*}
\text{(topp)} & \quad \Gamma_i(x) = (\forall \mu_i. F) \\
& \quad \frac{\Gamma_m; \Gamma_i; H \vdash x[\mu_i] : [\mu_i / t_i]\mu \Rightarrow \@z F ; ((z = x[\mu_i])); F}{H_1}
\end{align*}
\]

Figure 9: The Lifting Translation

**Notation** \(\lambda^* F.e\) and \(@^* zF\)

We use \(\lambda^* F.e\) and \(@^* zF\) to denote multiple abstractions and applications respectively. If \(F = \{x_1 : \mu_1, \ldots, x_n : \mu_n\}\), then \(\lambda^* F.e\) denotes \(\lambda x_1 : \mu_1.(\ldots(\lambda x_n : \mu_n.e)\ldots)\). Similarly \(@^* zF\) denotes \(@(\ldots(@z x_1)\ldots)x_n\).

**Notation** \(\text{Ty}(L)\)

If \(L = \{x_1, x_2, \ldots, x_n\}\) and the types of the variables are respectively \(\mu_1, \ldots, \mu_n\), then \(\text{Ty}(L) \rightarrow \mu\) is shorthand for \(\mu_1 \rightarrow (\ldots \rightarrow (\mu_n \rightarrow \mu)\ldots)\).

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

The variable environment is split up into 2 environments – \(\Gamma_m\) which binds the monomorphic variables, and \(\Gamma_i\) which binds the variables introduced by polymorphic let expressions.

**Lemma 9**

Suppose \(\Gamma_m; \Gamma_i; H \vdash e : \sigma \Rightarrow e' ; H' ; F\). If \(x\) occurs free in \(e'\), and \(x \in \Gamma_m\), then \(x \in F\).
**Proof** The proof is by induction on the structure of $e$. Most of the cases follow directly from the inductive hypothesis; the only cases of interest being *tapp* and *tfn*.

**case tapp** The only variables occurring free in $e'$ are $z$ and the variables in $F$. The variable $z$ is newly introduced and hence cannot occur in $\Gamma_m$, while $F$ is returned.

**case tfn** By the inductive hypothesis, if $y \in \Gamma_m$, and $y$ occurs free in $e'_2$, then $y \in F_2$. The free variables of the translated term are the free variables of $e'_2$ and therefore the lemma is satisfied. \[\Box\]

**Lemma 10**

Suppose $\Gamma_m; \Gamma_1; H \vdash e : \sigma \Rightarrow e' ; H' ; F$. Then if $x \in \Gamma_m$, then $x$ does not occur free in $H'$.

**Proof** The proof is by induction on the structure of $e$. We will only consider the cases that don’t follow directly from the inductive assumption.

**case tapp** The only free variable in the header is $x$ which occurs in $\Gamma_1$.

**case tfn** By Lemma 9, we know that $\lambda^{*}F_{1}.e'_{1}$ is closed with respect to variables in $\Gamma_m$. By induction on $e_{1}$, we know that $H_{1}$ is closed with respect to variables in $\Gamma_m$. This implies that $H'$ is closed with respect to variables in $\Gamma_m$. By induction on $e_{2}$, we know that $H_{2}$ is closed with respect to variables in $\Gamma_m$, which proves the lemma. \[\Box\]

**Lemma 11**

The variables bound in $\Gamma_m$ and $\Gamma_1$ are mutually disjoint; and the variables bound in $\Gamma_m$ and $\Gamma_H$ are mutually disjoint.

**Proof** The proof is again a straightforward induction over the translation rules. In each rule, only one of the environments is augmented with a variable binding. \[\Box\]

### A.1 Type Preservation

In this section, we prove that the transformation preserves the type of the original expression.

**Definition 1** (*LET($H, e$))*

If $H ::= [h_1, \ldots, h_n]$, and each $h_i ::= (y_i = e_i)$, then LET($H, e$) is shorthand for **let** $y_1 = e_1$ **in** $\ldots$ **in** **let** $y_n = e_n$ **in** $e$.

**Definition 2** (*$\Gamma_H$ Header Type Environment*)

Suppose $\Gamma_m; \Gamma_1; H \vdash e : \sigma \Rightarrow e' ; H' ; F$. If $H = [h_1, \ldots, h_n]$, then $\Gamma_H = \Gamma_{h_1} \uplus \ldots \uplus \Gamma_{h_n}$. If $h_i ::= (y_i = e_i)$, and $\Gamma_m; \Gamma_{h_1 \ldots h_{i-1}} \vdash e_i : \sigma_i$; then $\Gamma_{h_i} ::= y_i \mapsto \sigma_i$. 

22
\[
\begin{align*}
\text{(const/var)} & \quad \Gamma \vdash i : \text{Int} \quad \Gamma \vdash x : \Gamma(x) \\
\text{(fn)} & \quad \frac{}{\Gamma \vdash \lambda x : \Gamma_1 e : \Gamma_2 \rightarrow \Gamma_3} \\
\text{(app)} & \quad \frac{\Gamma \vdash x_1 : \Gamma_3 \rightarrow \Gamma_4 \quad \Gamma \vdash x_2 : \Gamma_5}{\Gamma \vdash x_1 x_2 : \Gamma_5} \\
\text{(ifn)} & \quad \frac{\Gamma \vdash e_1 : \Gamma_3 \quad \Gamma \vdash \{x : \Gamma_1 \} \vdash e : \Gamma_2 \quad t_i \text{ not free in } \Gamma}{\Gamma \vdash \textbf{let } x = \Lambda t_i e_i \textbf{ in } e : \Gamma_2} \\
\text{(tapp)} & \quad \frac{\Gamma \vdash x : \forall \Gamma_1, \mu}{\Gamma \vdash x[\mu] : [\mu_1/t_1] \mu} \\
\text{(let)} & \quad \frac{\Gamma \vdash e_1 : \Gamma_1 \quad \Gamma \vdash \{x : \mu_1\} \vdash e_2 : \Gamma_2}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \Gamma_2}
\end{align*}
\]

Figure 10: Static Semantics

**Definition 3 (H is well formed with respect to \( \Gamma_i \))**

\( H \) is well formed iff \( \Gamma_i(x) = (\forall \Gamma_1, \mu, F) \) implies that \( \Gamma_H(x) = \forall \Gamma_1, Ty(F) \rightarrow \mu \).

The static semantics is shown in Figure 10. The environment \( \Gamma \) is a generic environment which means that for the source terms, \( \Gamma \) is substituted with \( \Gamma_m; \Gamma_1 \), and for the target terms, \( \Gamma \) is substituted with \( \Gamma_m; \Gamma_H \).

**Lemma 12**

Suppose \( \Gamma_m; \Gamma_H \vdash \textbf{let } x = \text{LET}(H_1, e') \textbf{ in } \text{LET}(H_2, e'') : \mu_2 \) and \( x \) does not occur free in \( H_1 \) or \( H_2 \). Then \( \Gamma_m; \Gamma_H \vdash \text{LET}(H_1 || H_2, \textbf{let } x = e' \textbf{ in } e'') : \mu_2 \)

**Proof** From the typing rules, \( \Gamma_m; \Gamma_H \vdash \text{LET}(H_1, e') : \mu \) (for some \( \mu \)) and \( \Gamma_m[x \mapsto \mu] ; \Gamma_H \vdash \text{LET}(H_2, e'') : \mu_2 \). This implies that \( \Gamma_m ; \Gamma_H \not\supset \Gamma_{H_1} \vdash e' \mapsto \mu \). Since variables bound in \( H_1 \) do not occur in \( \text{LET}(H_2, e'') \), we get that \( \Gamma_m[x \mapsto \mu] ; \Gamma_H \not\supset \Gamma_{H_1} \vdash \text{LET}(H_2, e'') : \mu_2 \). Therefore \( \Gamma_m; \Gamma_H \not\supset \Gamma_{H_1} \vdash \textbf{let } x = e' \textbf{ in } \text{LET}(H_2, e'') : \mu_2 \).

The header \( H_2 \) is \( [h_1, \ldots, h_n] \) where each \( h_i \) is \( y_i = e_i \). Therefore write \( \text{LET}(H_2, e'') \) as \( \textbf{let } y_1 = e_1 \textbf{ in } e \) where \( e = \text{LET}([h_2, \ldots, h_n], e'') \). When we consider the typing rules for this expression, we get that \( \Gamma_m[x \mapsto \mu] ; \Gamma_H \not\supset \Gamma_{H_1} \vdash e_1 : \mu_1 \) (for some \( \mu_1 \)) and \( \Gamma_m[x \mapsto \mu] ; \Gamma_H \not\supset \Gamma_{H_1} \vdash [y_1 \mapsto \mu] \vdash e : \mu_2 \). We know that \( x \) does not occur in \( e_1 \), therefore we can remove it from the environment while typing \( e_1 \). Further \( y_1 \) does not occur in \( e' \) and can therefore be introduced into the environment while typing \( e' \). So we get the following typing equations - \( \Gamma_m; \Gamma_H \not\supset \Gamma_{H_1} \vdash e_1 : \mu_1 \) and \( \Gamma_m; \Gamma_H \not\supset \Gamma_{H_1} \vdash [y_1 \mapsto \mu] \vdash e' : \mu \) and \( \Gamma_m[x \mapsto \mu] ; \Gamma_H \not\supset \Gamma_{H_1} \vdash [y_1 \mapsto \mu] \vdash e : \mu_2 \). This implies that \( \Gamma_m; \Gamma_H \not\supset \Gamma_{H_1} \vdash \textbf{let } y_1 = e_1 \textbf{ in } \text{let } x = e' \textbf{ in } e : \mu_2 \). By continuing this process of breaking \( H_2 \) into its components, in the end, we get the following typing equations - \( \Gamma_m; \Gamma_H \not\supset \Gamma_{H_1} \vdash e' : \mu \) and \( \Gamma_m[x \mapsto \mu] ; \Gamma_H \not\supset \Gamma_{H_1} \vdash e'' : \mu_2 \). which proves the lemma.

\( \square \)
Lemma 13

Suppose that $\Gamma_m; \Gamma_H \vdash \lambda x : \mu. \text{LET}(H', e) : \mu \rightarrow \mu'$. Suppose further that $x$ does not occur free in $H'$. Then $\Gamma_m; \Gamma_H \vdash \text{LET}(H', \lambda x : \mu.e) : \mu \rightarrow \mu'$.

Proof The header $H' ::= [h_1, \ldots, h_n]$ with each $h_i ::= y_i = e_i$. Therefore write $\text{LET}(H', e)$ as $\text{let } y_1 = e_1 \text{ in } e''$ where $e'' := \text{LET}([h_2, \ldots, h_n], e)$. The typing rule for the expression $\lambda x : \mu. \text{LET}(H', e)$ can be written as $\Gamma_m[x \mapsto \mu]; \Gamma_H \vdash e_1 : \mu_1$ (for some type $\mu_1$) and $\Gamma_m[x \mapsto \mu]; \Gamma[H[y_1 \mapsto \mu_1] \vdash e'' : \mu'$. Since $x$ does not occur free in $e_1$, we can remove it from the environment while typing $e_1$. Therefore the typing equations now become $\Gamma_m; \Gamma_H \vdash e_1 : \mu_1$ and $\Gamma_m[x \mapsto \mu]; \Gamma[H[y_1 \mapsto \mu_1] \vdash e'' : \mu'$. This implies that $\Gamma_m; \Gamma_H \vdash \text{let } y_1 = e_1 \text{ in } \lambda x : \mu.e'' : \mu \rightarrow \mu'$. Continuing to break $H'$ like this, we finally reach $\Gamma_m; \Gamma_H \vdash \text{LET}(H', \lambda x : \mu.e) : \mu \rightarrow \mu'$.

\[\Box\]

Theorem 14 (Type Preservation)

Suppose $\Gamma_m; \Gamma_1; H \vdash e : \sigma \Rightarrow e'; H'; F$. Suppose further that $H$ is well formed with respect to $\Gamma_1$. If $\Gamma_m; \Gamma_1 \vdash e : \sigma$, then $\Gamma_m; \Gamma_H \vdash \text{LET}(H', e') : \sigma$.

Proof The proof is by induction on the structure of $e$.

\textbf{case int} Both the source and target terms have type $\text{Int}$.

\textbf{case var} $x$ occurs in $\Gamma_m$ and a variable does not occur in more than one environment(Lemma 11). Therefore both the source and the target terms have type $\Gamma_m(x)$.

\textbf{case app} By an argument similar to the (var) case, we can deduce that $\Gamma_m; \Gamma_H \vdash x_1 : \mu_1 \rightarrow \mu_2$ and $\Gamma_m; \Gamma_H \vdash x_2 : \mu_1$. This implies that $\Gamma_m; \Gamma_H \vdash \ast x_1 x_2 : \mu_2$.

\textbf{case let} By the inductive hypothesis, $\Gamma_m; \Gamma_H \vdash \text{LET}(H_1, e'_1) : \mu_1$ and $\Gamma_m[x \mapsto \mu_1]; \Gamma_H \vdash \text{LET}(H_2, e'_2) : \mu_2$. Since $H_2$ and $H_1$ are closed with respect to variables in $\Gamma_m$ (Lemma 10), $x$ does not occur free in $H_2$ and $H_1$. Applying Lemma 12 leads to the preservation theorem.

\textbf{case tapp} From the antecedent we know that $\Gamma_m(x) = (\forall t_i, \mu, F)$. By the well formedness of $H$, we know that $\Gamma_H(x) = \forall t_i, T_y(F) \rightarrow \mu$. Therefore $\Gamma_m; \Gamma_H \vdash x[\mu_i] : T_y(F) \rightarrow \mu[\mu_i/t_i]$. Since $F$ consists of the free variables of $x$, the type $T_y(F)$ cannot contain any of the $t_i$ as a free type variable. Therefore the translated term has type $\mu[\mu_i/t_i]$ which is the same as the source term.

\textbf{case tfn} We need to prove that $\Gamma_m; \Gamma_H \vdash \text{LET}(H' || H_2, e'_2) : \mu_2$ which implies that $\Gamma_m; \Gamma_H \vdash \text{LET}(H', e'_2) : \mu_2$ must be true. By definition, $\Gamma_H' ::= x \mapsto \sigma'$ where $\Gamma_m; \Gamma_H \vdash \text{LET}(H_1, \lambda^*F_1, e'_1) : \sigma'$. By the inductive hypothesis, we know that $\Gamma_m; \Gamma_H \vdash \text{LET}(H_1, e'_1) : \mu_1$. This implies that $\Gamma_m; \Gamma_H \vdash \lambda^*F_1.\text{LET}(H_1, e'_1) : T_y(F_1) \rightarrow \mu_1$ where $F_1$ is the set of free variables of $e'_1$ bound in $\Gamma_m$. By generalising Lemma 13, we get that $\Gamma_m; \Gamma_H \vdash \text{LET}(H' \parallel H_2, \lambda^*F_1, e'_1) : T_y(F_1) \rightarrow \mu_1$ since none of the $F_i$ occur free in $H_1$ (Lemma 10). By the inductive assumption, we knew that $H'$ was well formed with respect to $\Gamma_1$; we now showed that $H' || H_2$ is well formed with respect to $\Gamma_1[x \mapsto (\forall t_i, \mu_1, F_1)]$. Applying the inductive hypothesis now to the translation of $e_2$ leads to the type preservation theorem.

\textbf{case fn} By the inductive assumption, $\Gamma_m[x \mapsto \mu]; \Gamma_H \vdash \text{LET}(H', e') : \mu'$. This implies that $\Gamma_m; \Gamma_H \vdash \lambda x : \mu.\text{LET}(H', e') : \mu \rightarrow \mu'$. By Lemma 10, $x$ does not occur free in $H'$. Applying Lemma 13, we get $\Gamma_m; \Gamma_H \vdash \text{LET}(H', \lambda x : \mu.e') : \mu \rightarrow \mu'$.

\[\Box\]

24
A.2 Semantic Soundness

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

\[(\text{values}) \quad v := \; i \mid \text{Clos}(x^\mu, e, E) \mid \text{Clos}^\delta(t_i, e, E)\]

**Type of a Value**

- \(\Gamma \vdash i : \text{int}\)
  - if \(\Gamma \vdash \lambda x : \mu. e : \mu \rightarrow \mu', \) then \(\Gamma \vdash \text{Clos}(x^\mu, e, E) : \mu \rightarrow \mu'\)
  - if \(\Gamma \vdash \bigwedge_{i \in I} e_v : \forall_i \mu, \) then \(\Gamma \vdash \text{Clos}^\delta(t_i, e_v, E) : \forall_i \mu\)

**Notation** \(E\) respects \(\Gamma\)

If \(E\) respects \(\Gamma\) then \(E(x) = v\) and \(\Gamma(x) = \sigma\) implies that \(\Gamma \vdash v : \sigma\).

**Notation** \(E : \Gamma \vdash e \rightarrow v\)

Implies that in a value environment \(E\) respecting \(\Gamma\), \(e\) evaluates to \(v\).

Henceforth in this section, we will assume that the value environment \(E\) respects the type environment \(\Gamma\). To avoid clutter in the presentation, we will not state this explicitly. This also implies that if a variable \(x\) is bound to a value \(v_i\) and \(\Gamma \vdash x : \sigma\), then \(\Gamma \vdash v_i : \sigma\).

**Equivalence of Values** Two values, \(v\) and \(v'\), are equivalent \((v \approx v')\) under the following conditions

- \(i \approx i'\) if \(\Gamma \vdash i : \text{int}\) and \(\Gamma \vdash i : \text{int}\) and \(i = i'\).
- \(\text{Clos}(x^\mu, e, E) \approx \text{Clos}(x^\mu, e', E')\) iff
  - \(\Gamma \vdash \lambda x : \mu. e : \mu \rightarrow \mu'\) and \(\Gamma \vdash \lambda x : \mu. e' : \mu \rightarrow \mu',\) and
  - \(E[x \mapsto v] \vdash v \rightarrow v\) and \(v \approx v'\)
- \(\text{Clos}^\delta(t_i, e, E) \approx \text{Clos}^\delta(t_i, e', E')\) iff
  - \(\Gamma \vdash e : \forall_i \mu_i\) and \(\Gamma \vdash e' : \forall_i \mu_i,\) and
  - \(E[e_{\mu_i/t_i}] \vdash v \implies E'[e'_{\mu_i/t_i}] \rightarrow v', \) and \(v \approx v'\).

**Compatible Environments** Two environments \(E\) and \(E'\) are compatible iff \(\text{dom}(E) = \text{dom}(E')\) and \(\forall x \in E, E(x) \approx E'(x)\),

**Definition 4** (\(E_H\) Header Value Environment)

Suppose \(\Gamma_m; \Gamma_l; H \vdash e : \sigma \Rightarrow e' ; H' ; F\). If \(H := [h_1, \ldots, h_n]\), then \(E_H = E_{h_1} \cup \ldots \cup E_{h_n}\). If \(h_i := (y_i = e_i)\), and \(E_{m} ; E_{h_0 \ldots h_{i-1}} \vdash e_i \leadsto v_i\), then \(E_{h_i} := y_i \leadsto v_i\).

**Definition 5** (\(H\) is well founded)

Suppose \(\Gamma_m; \Gamma_l; H \vdash e : \sigma \Rightarrow e' ; H' ; F'\) and \(\Gamma_l(x) = (\forall i \sigma, F)\). Suppose further that \(E_m\) is compatible to \(E_m'\). We say that \(H\) is well founded iff, \(\forall x \in E_l, E_m ; E_l \vdash x [t_i] \leadsto v\) implies that \(E_m' ; E_H \vdash \circ x [t_i] F \leadsto v'\) and \(v \approx v'\).

In the above definition, note that the environments \(E_m, E_m'\) respect \(\Gamma_m\), the environment \(E_H\) respects \(\Gamma_H\) and the environment \(E_l\) respects \(\Gamma_l\). As we said above, we shall leave this implicit to help the presentation, but it should be kept in mind that the value environment respects the type environment.
\[
\begin{align*}
(\text{const/var}) & \quad E \vdash i \rightarrow i \quad E \vdash x \rightarrow E(x) \\
(fn) & \quad E \vdash \lambda x; \mu. e \rightarrow \text{Clos}(x^\mu, e, E) \\
(app) & \quad E \vdash x_1 \rightarrow \text{Clos}(x^\mu, e, E') \\
 & \quad E \vdash x_2 \rightarrow v' \\
 & \quad E' + x \mapsto v' + e \rightarrow v \\
 & \quad E \vdash \oplus x_1 x_2 \rightarrow v \\
(tfn) & \quad E \vdash \Lambda t_i e_v \rightarrow \text{Clos}(t_i, e_v, E) \\
(let) & \quad E \vdash e_1 \rightarrow v_1 \\
 & \quad E + x \mapsto v_1 + e_2 \rightarrow v \\
 & \quad E \vdash \text{let} x = e_1 \text{ in } e_2 \rightarrow v \\
(tapp) & \quad E \vdash z \rightarrow \text{Clos}(t_i, e_v, E') \\
 & \quad E' + e_v[\mu_i/t_i] \rightarrow v \\
 & \quad E \vdash z[\mu_i] \rightarrow v 
\end{align*}
\]

Figure 11: Operational Semantics

The operational semantics is shown in Figure 11. The environment \( E \) is a generic environment which means that for the source terms, \( E \) is substituted with \( E_m; E_i \) and for the target terms, \( E \) is substituted with \( E_m'; E_H \).

**Lemma 15**

Suppose \( E_m; E_H \vdash \text{let } x = \text{LET}(H_1, e') \text{ in } \text{LET}(H_2, e'') \leadsto v \) and \( x \) does not occur free in either \( H_1 \) or \( H_2 \). Then \( E_m; E_H \vdash \text{LET}(H_1 \parallel H_2, \text{let } x = e' \text{ in } e'') \leadsto v \)

**Proof** The proof is exactly similar to the proof for Lemma 12. We only need to replace the type environments \((\Gamma_m; \Gamma_H)\) with the corresponding value environments \((E_m; E_H)\).

\(\square\)

**Theorem 16 (Semantic Soundness)**

Suppose \( \Gamma_m; \Gamma_H; H \vdash e : \sigma \Rightarrow e'; H'; F' \). Suppose further that \( E_m \) and \( E_m' \) are compatible. If \( H \) is well founded and \( E_m; E_i \vdash e \leadsto v \), then \( E_m'; E_H \vdash \text{LET}(H', e') \leadsto v' \) with \( v \approx v' \).

**Proof** The proof is by induction on the structure of \( e \).

**case int** Both the source and the target terms evaluate to \( i \).

**case var** The source term evaluates to \( E_m(x) \); the target term to \( E_m'(x) \). The equivalence follows since \( E_m \) and \( E_m' \) are compatible.

**case app** In the source term, \( x_1 \) evaluates to a closure, \( E_m(x_1) \), say \( v_1 \). \( x_2 \) evaluates to \( E_m(x_2) \), say \( v_2 \). In the target term, \( x_1 \) evaluates to a closure, \( E_m'(x_1) \), say \( v_1' \). \( x_2 \) evaluates to \( E_m'(x_2) \), say \( v_2' \). Since \( E_m \) and \( E_m' \) are compatible, \( v_1 \approx v_1' \) and \( v_2 \approx v_2' \). The required equivalence follows now from the definition of the equivalence of closures.

26
**case let** By the inductive assumption, if \( E_m; E_1 \vdash e_1 \leadsto v_1 \), then \( E'_m; E_H \vdash \text{LET}(H_1, e'_1) \leadsto v'_1 \) and \( v_1 \approx v'_1 \). Similarly, by induction on \( e_2 \), if \( E_m[x \mapsto v_1]; E_1 \vdash e_2 \leadsto v_2 \), then \( E'_m[x \mapsto v'_1]; E_H \vdash \text{LET}(H_2, e'_2) \leadsto v'_2 \) and \( v_2 \approx v'_2 \). This implies that \( E'_m; E_H \vdash \text{let} \; x = \text{LET}(H_1, e'_1) \) in \( \text{LET}(H_2, e'_2) \leadsto v'_2 \). By Lemma 10, \( x \) does not occur free in either \( H_1 \) or \( H_2 \). But by Lemma 15, this implies that \( E_m; E_H \vdash \text{LET}(H_1 || H_2, \text{let} \; x = e'_1 \text{ in } e'_2) \leadsto v'_2 \). The required equivalence follows from this.

**case tapp** We need to prove that if \( E_m; E_1 \vdash x[i] \leadsto v \), then \( E'_m; E_H \vdash @x[i] F \leadsto v' \) and that \( v \approx v' \); given that \( E_m \) and \( E'_m \) are compatible. But this follows from the definition of well foundedness of the header \( H \).

**case tfn** We need to prove that if \( E_m; E_1 \vdash \text{let} \; x = \lambda \eta.e \; \text{in} \; e_2 \leadsto v_2 \), then \( E'_m; E_H \vdash \text{LET}(H' || H_2, e'_2) \leadsto v'_2 \) and \( v_2 \approx v'_2 \). We will first prove that the augmented environment \( H || H' \) is well founded during the translation of \( e_2 \). By definition, \( E_{H'} := x \mapsto \text{Clo}_{F_i}(\Pi_i, \text{LET}(H_i, \lambda^*F_i.e'_i), E'_m, E_H) \).

Now for any compatible pairs of environments; \( E_m, E'_m \), and environments \( E_1, E'_{H_1} \), we need to prove that if \( E_m; E_1 \vdash x[i] \leadsto v \) then \( E'_m; E'_{H_1} \vdash @x[i] F \leadsto v' \) and \( v \approx v' \). This implies we need to prove that if \( E_m; E_1 \vdash e_1[\mu_i/t_i] \leadsto v \) then \( E'_m; E'_{H_1} \vdash @x[\mu_i] F \leadsto v' \) and \( v \approx v' \). We will simplify the evaluation of the target term. \( E_m; E_1 \vdash x[\mu_i] \rightarrow F \rightarrow \text{LET}(H_1, \lambda^*F_i.e')(\mu_i/t_i) \). This in turn reduces to \( \text{Clo}_{F_i}(\Pi_i, \lambda^*\text{LET}(H_i, \lambda^*F_i.e'), \Pi_i, E_m \circ \text{LET}(H_1, \lambda^*F_i.e')(\mu_i/t_i)) \). Therefore, the evaluation of the target term reduces to \( E'_m; E'_{H_1} \vdash @x[F_i] \rightarrow \text{LET}(H_1, \lambda^*F_i.e')(\mu_i/t_i) \rightarrow @x[\mu_i] F \rightarrow \text{LET}(H_1, \lambda^*F_i.e')(\mu_i/t_i)) \). This in turn reduces to \( E'_m \circ \text{LET}(H_1, \lambda^*F_i.e')(\mu_i/t_i) \rightarrow e_1[\mu_i/t_i] \). Since variables are bound only once, \( E'_m(F_1) = E_m(F_1) \). Therefore \( E'_m(F_1) \equiv E_m = E'_m \). And so the evaluation of the target term can be reduced finally to \( E'_m; E_H \circ \text{LET}(H_1, \lambda^*F_i.e)(\mu_i/t_i) \rightarrow e_1[i] \rightarrow v' \). But by induction on the translation of \( e_1 \), we get that \( v \approx v' \). Hence \( H || H' \) is well founded with respect to \( E_i[x \mapsto v] \). And so we can apply the inductive hypothesis to the translation of \( e_2 \) which leads directly to the semantic soundness result.

**case fn** We need to prove that if \( E_m; E_1 \vdash \lambda x : \mu.e \leadsto v \), then \( E'_m; E_H \vdash \text{LET}(H', \lambda x : \mu.e') \leadsto v' \) and \( v \approx v' \). By the inductive hypothesis, we know that if \( E_m[x \mapsto v_1]; E_1 \vdash e \leadsto v_2 \), then \( E'_m[x \mapsto v'_1]; E_H \vdash \text{LET}(H', e') \leadsto v'_2 \) and \( v_2 \approx v'_2 \). By Lemma 10, \( x \) does not occur free in \( H' \). The header \( H' \) actually consists of \( [h_1, \ldots, h_n] \) and each \( h_i := y_i = e_i \). Consider the expression \( \text{LET}(H', e') \) as \( \text{let} \; y_i = e_1 \text{ in } e'' \) where \( e'' = \text{LET}(h_2, \ldots, h_n, e') \). From the evaluation rules we can deduce that \( E'_m[x \mapsto v'_1]; E_H \vdash e_1 \leadsto u_1 \) (for some \( u_1 \)) and \( E'_m[x \mapsto v'_1]; E_H[y_1 \mapsto u_1] \rightarrow e'' \leadsto v'_2 \). But since \( x \) does not occur in \( e_1 \), we can remove the binding of \( x \) from the environment while evaluating \( e_1 \). Therefore we get that \( E'_m; E_H \vdash e_1 \leadsto u_1 \) and \( E'_m[x \mapsto v'_1]; E_H[y_1 \mapsto u_1] \rightarrow e'' \leadsto v'_2 \). By continuing this process of splitting \( H' \), we can prove the required equivalence.

\[\square\]

## B Proofs For The FLINT Algorithm

In this section, we prove the semantic soundness and the type preservation property of the algorithm.

**Notation** \( \lambda^*F.e \) and \( @^*zF \)

We use \( \lambda^*F.e \) and \( @^*zF \) to denote multiple abstractions and applications respectively. If \( F = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \), then \( \lambda^*F.e \) denotes \( \lambda x_1 : \sigma_1.(\ldots(\lambda x_n : \sigma_n.e)\ldots) \). Similarly \( @^*zF \) denotes \( @.(\ldots(@z_2x_1)\ldots)x_n \).

**Notation** \( Ty(L) \)

If \( L = \{x_1, x_2, \ldots, x_n\} \) and the types of the variables are respectively \( \sigma_1, \ldots, \sigma_n \), then \( Ty(L) \rightarrow \sigma \) is shorthand for \( \sigma_1 \rightarrow (\ldots(\sigma_n \rightarrow \sigma)\ldots) \).

Throughout this section, we assume unique variable bindings – variables are never redefined in the program.
\[(\text{int})\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash i : \text{Int} \Rightarrow i ; \nil ; \emptyset
\]

\[(\text{var})\]
\[
\Gamma_m(x) = T(\mu)
\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash x : T(\mu) \Rightarrow x ; \nil ; \{x : T(\mu)\}
\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash x : \sigma \Rightarrow x ; \nil ; \emptyset
\]

\[(\text{app})\]
\[
\Gamma_m(x_1) = T(\mu_1 \rightarrow \mu_2)
\]
\[
\Gamma_m(x_2) = T(\mu_1)
\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash \circ x_1 x_2 : T(\mu_2) \Rightarrow \circ x_1 x_2 ; \nil ; \{x_1 : T(\mu_1 \rightarrow \mu_2), x_2 : T(\mu_1)\}
\]

\[(\text{app2})\]
\[
\Gamma_p(x_1) = \sigma_1 \rightarrow \sigma_2
\]
\[
\Gamma_p(x_2) = \sigma_1
\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash \circ x_1 x_2 : \sigma_2 \Rightarrow \circ x_1 x_2 ; \emptyset ; \emptyset
\]

\[(\text{let})\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash e_1 : T(\mu_1) \Rightarrow e'_1 ; H_1 ; F_1
\]
\[
\Gamma_m[x \mapsto T(\mu_1)]; \Gamma_p; \Gamma; H; d \vdash e_2 : \sigma_2 \Rightarrow e'_2 ; H_2 ; F_2
\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash \text{let } x = e_1 \in e_2 : \sigma_2 \Rightarrow \text{let } x = e'_1 \in e'_2 ; H_1 \parallel H_2 ; F_1 \cup F_2 \{x : T(\mu_1)\}
\]

\[(\text{let2})\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash e_1 : \sigma_1 \Rightarrow e'_1 ; H_1 ; F_1
\]
\[
\Gamma_m; \Gamma_p[x \mapsto \sigma_1]; \Gamma; H; d \vdash e_2 : \sigma_2 \Rightarrow e'_2 ; H_2 ; F_2
\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash \text{let } x = e_1 \in e_2 : \sigma_2 \Rightarrow \text{let } x = e'_1 \in \text{LET}(H_2, e'_2) ; H_1 ; F_1 \cup F_2
\]

\[(\text{app})\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash e_1 : \sigma_1 \Rightarrow e'_1 ; H_1 ; F_1
\]
\[
\Gamma_m[x \mapsto T(\mu_1)]; \Gamma_p; \Gamma; H; d \vdash e_2 : \sigma_2 \Rightarrow e'_2 ; H_2 ; F_2
\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash \text{let } x = e_1 \in e_2 : \sigma_2 \Rightarrow \text{let } x = e'_1 \in \text{LET}(H_2, e'_2) ; H_1 ; F_1 \cup F_2
\]

\[(\text{tfn})\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d > 0 \vdash x = \Lambda_{i.e_1} \text{ in } e_2 : \sigma_2 \Rightarrow e'_2 ; H' \parallel H_2 ; F_2
\]

\[(\text{tfn2})\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d > 0 \vdash x = \Lambda_{i.e_1} \text{ in } e_2 : \sigma_2 \Rightarrow e'_2 ; H' \parallel H_2 ; F_2
\]

\[(\text{tfn3})\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d > 0 \vdash x = \Lambda_{i.e_1} \text{ in } e_2 : \sigma_2 \Rightarrow e'_2 ; H' \parallel H_2 ; F_2
\]

\[(\text{fct})\]
\[
\Gamma_m; \Gamma_p[x \mapsto \sigma]; \Gamma; H; d = 0 \vdash e : \sigma' \Rightarrow e' ; H' ; F
\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d \vdash \lambda^m x : \sigma.e : \sigma' \Rightarrow \lambda^m x : \sigma.e' ; H' ; F
\]
\[
\Gamma_m[x \mapsto T(\mu)]; \Gamma_p; \Gamma; H; d + 1 \vdash e : T(\mu) \Rightarrow e' ; H' ; F
\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d = 0 \vdash \lambda^c x : T(\mu).e : T(\mu) \Rightarrow \text{LET}(H', \lambda^c x : T(\mu).e') ; \nil ; F \{x : T(\mu)\}
\]
\[
\Gamma_m; \Gamma_p; \Gamma; H; d > 0 \vdash \lambda^c x : T(\mu).e : T(\mu) \rightarrow \lambda^c x : T(\mu).e' ; H' ; F \{x : T(\mu)\}
\]

Figure 12: The Lifting Algorithm For FLINT
The variable environment is split up into 3 environments – $\Gamma_m$ which binds the monomorphic variables, $\Gamma_p$ which binds the variables introduced by functor abstractions, and $\Gamma_i$ which binds the variables introduced by polymorphic let expressions.

**Lemma 17**

Suppose $\Gamma_m; \Gamma_p; \Gamma_i; H; d \vdash e : \sigma \Rightarrow e' ; H' ; F$. If $x$ occurs free in $e'$, and $x \in \Gamma_m$, then $x \in F$.

**Proof** The proof is by induction on the structure of $e$. Most of the cases follow directly from the inductive hypothesis; the only cases of interest being $\text{tapp}$ and $tfn$.

- **case tapp** For ($d > 0$), the only variables occurring free in $e'$ are $z$ and the variables in $L$. The variable $z$ is newly introduced and hence cannot occur in $\Gamma_m$, while $L$ is returned. For ($d = 0$), the only variables occurring free in $e'$ are $x$ and the variables in $L$. The variable $x$ occurs in $\Gamma_i$, while $L$ is returned.

- **case tapp2** For ($d > 0$), the only variable occurring free in $e'$ is $z$ which is a newly introduced variable; hence does not occur in $\Gamma_m$. For ($d = 0$), $x$ is the only free variable but $x$ occurs in $\Gamma_p$.

- **case tfn** By the inductive hypothesis, if $x \in \Gamma_m$ and $x$ occurs free in $e'_1$, then $x \in F_1$. And similarly, if $y \in \Gamma_m$, and $y$ occurs free in $e'_2$, then $y \in F_2$. The free variables of the translated term are the free variables of $e'_2$ and therefore the lemma is satisfied.

- **case tfn2** By the inductive hypothesis, if $x \in \Gamma_m$ and $x$ occurs free in $e'_1$, then $x \in F_1$. And similarly, if $y \in \Gamma_m$, and $y$ occurs free in $e'_2$, then $y \in F_2$. The free variables of the translated term are the free variables of $e'_2$ plus the free variables of $e'_1$ which is $F_2 \cup F_1$.

**Lemma 18**

Suppose $\Gamma_m; \Gamma_p; \Gamma_i; H; d \vdash e : \sigma \Rightarrow e' ; H' ; F$. Then if $d = 0$, the header $H' = \text{nil}$.

**Proof** The proof is a straightforward induction over the structure of $e$.

**Lemma 19**

Suppose $\Gamma_m; \Gamma_p; \Gamma_i; H; d \vdash e : \sigma \Rightarrow e' ; H' ; F$. Then if $x \in \Gamma_m$, then $x$ does not occur free in $H'$.

**Proof** The proof is by induction on the structure of $e$. We will only consider the cases that do not follow directly from the inductive assumption.

- **case tapp** The header is non-null only for $d > 0$. The only free variable in the header is $x$ which occurs in $\Gamma_i$.

- **case tapp2** The header is non-null only for $d > 0$. The only free variable in the header is $x$ which occurs in $\Gamma_p$.

- **case tfn** By Lemma 17, we know that $\lambda^*F_1.e'_1$ is closed with respect to variables in $\Gamma_m$. By induction on $e_1$, we know that $H_1$ is closed with respect to variables in $\Gamma_m$. This implies that $H'$ is closed with respect to variables in $\Gamma_m$. By induction on $e_2$, we know that $H_2$ is closed with respect to variables in $\Gamma_m$, which proves the lemma.

- **case tfn2** By Lemma 18, the header $H_1 \parallel H_2$ is nil and so the lemma holds trivially.

\[\Box\]
Lemma 20

The variables bound in $\Gamma_m$, $\Gamma_p$ and $\Gamma_l$ are mutually disjoint; and the variables bound in $\Gamma_m$, $\Gamma_p$ and $\Gamma_H$ are mutually disjoint.

**Proof** The proof is again a straightforward induction over the translation rules. In each rule, only one of the environments is augmented with a variable binding.

\[
\square
\]

B.1 Type Preservation

In this section, we prove that the transformation preserves the type of the original expression.

\[
\begin{array}{ll}
(\text{const/var}) & \Gamma \vdash i : \text{Int} \quad \Gamma \vdash x : \Gamma(x) \\
(fn) & \Gamma \vdash \{x : T(\mu_1)\} \vdash e : T(\mu_2) \\
& \Gamma \vdash \lambda x : T(\mu_1)e : T(\mu_1 \rightarrow \mu_2) \\
(fct) & \Gamma \vdash \{x : \sigma_1\} \vdash e : \sigma_2 \\
& \Gamma \vdash \lambda^m x : \sigma_1 e : \sigma_1 \rightarrow \sigma_2 \\
(app) & \Gamma \vdash x_1 : \sigma' \rightarrow \sigma \quad \Gamma \vdash x_2 : \sigma' \\
& \Gamma \vdash \mathbf{@}x_1 x_2 : \sigma \\
(app2) & \Gamma \vdash x_1 : T(\mu_1 \rightarrow \mu_2) \quad \Gamma \vdash x_2 : T(\mu_1) \\
& \Gamma \vdash \mathbf{@}x_1 x_2 : T(\mu_2) \\
(tfn) & \Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash \{x : \forall \mu_i \sigma_1\} \vdash e_2 : \sigma_2 \\
& \text{t}_i \text{ not free in } \Gamma \\
& \Gamma \vdash \text{let } x = \Lambda \mu_i e_1 \text{ in } e : \sigma_2 \\
(tapp) & \Gamma \vdash x : \forall \mu_i \sigma \\
& \Gamma \vdash \mu_j = i - j \\
& \Gamma \vdash x[\mu_j] : \forall \mu_i [\mu_j / t_i] \sigma \\
(let) & \Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash \{x : \sigma_1\} \vdash e_2 : \sigma_2 \\
& \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \sigma_2
\end{array}
\]

Figure 13: Static Semantics

Definition 6 (LET($H, e$))

If $H ::= [h_1, \ldots, h_n]$, and each $h_i ::= (y_i = e_i)$, then LET($H, e$) is shorthand for let $y_1 = e_1 \text{ in } \ldots \text{ in } \text{let } y_n = e_n \text{ in } e$. 

30
Definition 7 (Γ_H Header Type Environment)

Suppose Γ_m; Γ_p; Γ_H; H_d ⊢ e : σ → e' ; H' ; F. If H = [h_1, ..., h_n], then Γ_H = Γ_{h_1} ⊔ ... ⊔ Γ_{h_n}. If h_i := (y_i = e_i), and Γ_m; Γ_p; Γ_{h_1...h_{i-1}} ⊢ e_i : σ_i; then Γ_{h_i} := y_i → σ_i.

Definition 8 (H is well formed with respect to Γ_i)

H is well formed iff Γ_i(x) = (∨i,σ,F) implies that Γ_H(x) = (∨i.Ty(F) → σ).

The static semantics is shown in Figure 13. The environment Γ is a generic environment which means that for the source terms, Γ is substituted with Γ_m; Γ_p; Γ_i, and for the target terms, Γ is substituted with Γ_m; Γ_p; Γ_H.

Lemma 21

Suppose Γ_m; Γ_p; Γ_H ⊢ let x = \text{LET}(H_1,e') in LET(H_2,e'') : σ and x does not occur free in either H_1 or H_2. Then Γ_m; Γ_p; Γ_H ⊢ \text{LET}(H_1 || H_2, let x = e' in e'') : σ.

Proof From the typing rules for this expression, we get that Γ_m[x → T(μ)]; Γ_p; Γ_H ⊢ \text{LET}(H_2,e'') : σ. This implies that Γ_m; Γ_p; Γ_H ⊢ e : T(μ). Since variables bound in H_1 do not occur in \text{LET}(H_2,e''), we get that Γ_m[x → T(μ)]; Γ_p; Γ_H ⊢ \text{LET}(H_2,e'') : σ. Therefore Γ_m; Γ_p; Γ_H ⊢ let x = e' in \text{LET}(H_2,e'') : σ.

The header H_2 := [h_1, ..., h_n] where each h_i := y_i = e_i. Therefore write \text{LET}(H_2,e'') as let y_1 = e_1 in e where e = \text{LET}([h_2, ..., h_n],e''). When we consider the typing rules for this expression, we get that Γ_m[x → T(μ)]; Γ_p; Γ_H ⊢ e : σ_1. Further y_1 does not occur in e' and can therefore be introduced into the environment while typing e'. So we get the following typing equations Γ_m; Γ_p; Γ_H ⊢ e : σ_1 and Γ_m[x → T(μ)]; Γ_p; Γ_H ⊢ y_1 → σ_1 | e : σ for some type σ_1. Therefore we can remove it from the environment while typing e'. Continuing this process of breaking H_2 into its components, in the end, we get the following typing equations Γ_m; Γ_p; Γ_H ⊢ e : σ which proves the lemma.

Lemma 22

Suppose that Γ_m; Γ_p; Γ_H ⊢ λx : T(μ).LET(H',e) : T(μ) → σ'. Suppose further that x does not occur free in H'. Then Γ_m; Γ_p; Γ_H ⊢ \text{LET}(H',λx : T(μ).e) : T(μ) → σ'.

Proof The header H' := [h_1, ..., h_n] where each h_i := y_i = e_i. Therefore write \text{LET}(H',e) as let y_1 = e_1 in e' where e' := \text{LET}([h_2, ..., h_n],e). The typing rule for the expression λx : T(μ).LET(H',e) can be written as Γ_m[x → T(μ)]; Γ_p; Γ_H ⊢ e : σ_1 and Γ_m[x → T(μ)]; Γ_p; Γ_H[y_1 → σ_1] | e' : σ'. Since x does not occur free in e_1, we may remove it from the environment while typing e_1. Therefore, the typing equations now become Γ_m; Γ_p; Γ_H ⊢ e : σ_1 and Γ_m[x → T(μ)]; Γ_p; Γ_H[y_1 → σ_1] | e' : σ'. This implies that Γ_m; Γ_p; Γ_H ⊢ let y_1 = e_1 in λx : T(μ).e' : T(μ) → σ'. Continuing to break H' like this, we finally reach Γ_m; Γ_p; Γ_H ⊢ \text{LET}(H',λx : T(μ).e) : T(μ) → σ'.
Theorem 23 (Type Preservation)

Suppose \( \Gamma_m ; \Gamma_p ; \Gamma_l ; \Gamma_H ; d \vdash e : \sigma \rightarrow e' ; H' ; F \) and \( H \) is well formed with respect to \( \Gamma_l \). If \( \Gamma_m ; \Gamma_p ; \Gamma_l \vdash e : \sigma \), then \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash \text{LET}(H', e') : \sigma \).

Proof The proof is by induction on the structure of \( e \).

**case int** Both the source and target terms have type \( \text{Int} \).

**case var** \( x \) occurs in \( \Gamma_m \) and a variable does not occur in more than one environment (Lemma 20). Therefore both the source and target terms have type \( \Gamma_m(x) \).

**case var** \( x \) occurs in \( \Gamma_p \) and a variable does not occur in more than one environment. Therefore both the source and target terms have type \( \Gamma_p(x) \).

**case app** By an argument similar to the (var1) case, we can deduce that \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash x_1 : T(\mu_1 \rightarrow \mu_2) \) and \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash x_2 : T(\mu_2) \). This implies that \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash \xi x_1 x_2 : T(\mu_2) \).

**case app2** By an argument similar to the (var2) case, we can deduce that \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash x_1 : \sigma_1 \rightarrow \sigma_2 \) and \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash x_2 : \sigma_1 \). This implies that \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash \xi x_1 x_2 : \sigma_2 \).

**case let** By the inductive hypothesis, \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash \text{LET}(H_1, e'_1) : T(\mu_1) \) and \( \Gamma_m[x \mapsto T(\mu_1)] ; \Gamma_p ; \Gamma_H \vdash \text{LET}(H_2, e'_2) : \sigma_2 \). Since \( H_2 \) and \( H_1 \) are closed with respect to variables in \( \Gamma_m \) (Lemma 19), \( x \) does not occur free in \( H_2 \) or \( H_1 \). Applying Lemma 21 leads to the preservation theorem.

**case let2** By the inductive hypothesis, \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash \text{LET}(H_1, e'_1) : \sigma_1 \) and \( \Gamma_m[x \mapsto \sigma_1] ; \Gamma_H \vdash \text{LET}(H_2, e'_2) : \sigma_2 \). Since the variables bound in \( H_1 \) do not occur in \( \text{LET}(H_2, e'_2) \), we can introduce the variables in \( H_1 \) in the environment while typing \( \text{LET}(H_2, e'_2) \). Therefore we have that \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash e'_1 : \sigma_1 \) and \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash e'_2 : \sigma_2 \). The type preservation theorem follows from this.

**case tapp** From the antecedent we know that \( \Gamma_l(x) = (\forall \ell, \sigma, L) \). By the well formedness of \( H \), we know that \( \Gamma_H(x) = \forall \ell, T y(L) \rightarrow \sigma \). Therefore \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash x \forall \ell k . T y(L) \rightarrow \sigma[\mu_j / t_j] \). Since \( L \) consists of the free variables of \( x \), the type \( T y(L) \) cannot contain any of the \( t_i \) as a free type variable.

For the \( (d = 0) \) case, therefore \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash x \forall \ell k . T y(L) \rightarrow \sigma[\mu_j / t_j][\ell_k / t_k] \). Therefore the translated term has type \( \forall \ell [k . \sigma][\mu_j / t_j][\ell_k / t_k] \). Therefore the translated term has type \( \forall \ell [k . \sigma][\mu_j / t_j][\ell_k / t_k] \) from where the preservation theorem follows directly.

**case tapp2** We know that \( \Gamma_p(x) = \sigma \) and a variable does not occur in more than one environment (Lemma 20). Therefore both the source and target terms, \( x \) has type \( \sigma \). For the \( (d = 0) \) case, the type preservation follows immediately. For the \( (d > 0) \) case, we get that \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash z ; \sigma[\mu_i / t_i] \) which means that the translated term has the same type as the source term.

**case tfn** We need to prove that \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash \text{LET}(H' \ || \ H_2, e'_2) : \sigma_2 \) which implies that \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash \text{LET}(H, e'_2) : \sigma' \) and \( \Gamma_m \vdash \text{LET}(H', e'_2) : \sigma'' \). By definition, \( \Gamma_H := x \mapsto \sigma' \) where \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash \Lambda \ell_r . \text{LET}(H_1, \lambda^* F_i, e_i') : \sigma'' \). By the inductive hypothesis, we know that \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash \text{LET}(H_1, e'_1) : \sigma_1 \). This implies that \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash \lambda^* F_i . \text{LET}(H_1, e'_1) : T y(F_i) \rightarrow \sigma_1 \) where \( F_i \) is the set of free variables of \( e'_1 \) bound in \( \Gamma_m \). By generalising Lemma 22, we get that \( \Gamma_m ; \Gamma_p ; \Gamma_H \vdash \text{LET}(H_1, \lambda^* F_i, e_i') : T y(F_i) \rightarrow \sigma_1 \) since none of the \( F_i \) occur free in \( H_1 \) (Lemma 19). By the inductive assumption, we know that \( H \) was well formed with respect to \( \Gamma_l \); we now showed that \( H' \ || \ H_2 \) is well formed with respect to \( \Gamma_l[x \mapsto (\forall \ell, \sigma_1, F_i)] \). Applying the inductive hypothesis now to the translation of \( e_2 \) leads to the type preservation theorem.
case tfn2 We know that $H_1 = H_2 = \text{nil}(\text{Lemma 18})$. Therefore we need to prove that $\Gamma_m; \Gamma_p; \Gamma_H \vdash \text{let } x = \lambda \overline{t}_i.e_1 \text{ in } e_2; \sigma_2$. By induction on the translation of $e_1$, ($H_1 = \text{nil}$), we know that $\Gamma_m; \Gamma_p; \Gamma_H \vdash e_1 : \sigma_1$. Therefore $\Gamma_m; \Gamma_p; \Gamma_H \vdash \lambda \overline{t}_i.e_1 : \forall \overline{t}_i \sigma_1$. This implies that $H \parallel H'$ is well formed with respect to $\Gamma_l[x \mapsto (\forall \overline{t}_i, \sigma_1, \text{nil})]$. Applying the inductive hypothesis now to the translation of $e_2$ leads to the type preservation theorem.

case fct By the inductive assumption, $\Gamma_m; \Gamma_p[x \mapsto \sigma]; \Gamma_H \vdash \text{LET} (H', e') : \sigma'$. By Lemma 18, $H'$ is nil. Therefore we get that $\Gamma_m; \Gamma_p[x \mapsto \sigma]; \Gamma_H \vdash e' : \sigma'$. The type preservation theorem follows from here.

case fn By the inductive assumption, $\Gamma_m[x \mapsto T(\mu)]; \Gamma_p; \Gamma_H \vdash \text{LET} (H', e') : T(\mu')$. This implies that $\Gamma_m; \Gamma_p; \Gamma_H \vdash \lambda x : \sigma.e : \sigma'$. By Lemma 19, $x$ does not occur free in $H'$. Applying Lemma 22, we get $\Gamma_m; \Gamma_p; \Gamma_H \vdash \text{LET} (H', \lambda x : T(\mu).e') : T(\mu \rightarrow \mu')$. In both the ($d = 0$) and ($d > 0$) cases, we need to prove the above typing equation.

\[\square\]

B.2 Semantic Soundness

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

\[(\text{values}) \quad v := i \mid Clos(x^\sigma, e, E) \mid Clos^t(t_i, e, E)\]

Type of a Value

- $\Gamma \vdash i : \text{int}$
  - if $\Gamma \vdash \lambda x : \sigma.e : \sigma' \text{, then } \Gamma \vdash Clos(x^\sigma, e, a) : \sigma \rightarrow \sigma'$
  - if $\Gamma \vdash \lambda \overline{t}_i.e_v : \forall \overline{t}_i \sigma$, then $\Gamma \vdash Clos^t(\overline{t}_i, e_v, a) : \forall \overline{t}_i \sigma$

Notation $E$ respects $\Gamma$

If $E$ respects $\Gamma$ then $E(x) = v$ and $\Gamma(x) = \sigma$ implies that $\Gamma \vdash v : \sigma$.

Notation $E : \Gamma \vdash e \rightarrow v$

Implies that in a value environment $E$ respecting $\Gamma$, $e$ evaluates to $v$.

Henceforth in this section, we will assume that the value environment $E$ respects the type environment $\Gamma$. To avoid clutter in the presentation, we will not state this explicitly. This also implies that if a variable $x$ is bound to a value $v$, and $\Gamma \vdash x : \sigma$, then $\Gamma \vdash v : \sigma$.

Equivalence of Values

Two values, $v$ and $v'$, are equivalent ($v \approx v'$) under the following conditions

- $i \approx i'$ iff $\Gamma \vdash i : \text{Int}$ and $\Gamma \vdash i : \text{Int}$ and $i = i'$.
- $\text{Clos}(x^\sigma, e, E) \approx \text{Clos}(x'^\sigma, e', E')$ iff
  - $\Gamma \vdash \lambda x : \sigma.e : \sigma \rightarrow \sigma' \text{ and } \Gamma \vdash \lambda x : \sigma.e' : \sigma \rightarrow \sigma'$, and
  - $E[x \mapsto v_i] \vdash e \rightarrow v$ and $v_1 \approx v_1'$ implies that $E'[x \mapsto v'_i] \vdash e' \rightarrow v'$ and $v \approx v'$
- $\text{Clos}^t(t_i, e, E) \approx \text{Clos}^t(t_i', e', E')$ iff
  - $\Gamma \vdash e : \forall t_i \sigma$ and $\Gamma \vdash e' : \forall t_i \sigma$, and
  - $E \vdash e[\mu_i/t_i] \rightarrow v$ implies that $E' \vdash e'[\mu_i/t_i] \rightarrow v'$, and $v \approx v'$.

Compatible Environments

Two environments $E$ and $E'$ are compatible iff $\text{dom}(E) = \text{dom}(E')$ and $\forall x \in E, E(x) \approx E'(x)$,
(const/var) \[ E \vdash i \rightarrow i \quad E \vdash x \rightarrow E(x) \]

(fn) \[ E \vdash \lambda x : T(\mu). e \rightarrow Clos(x^T[\mu], e, E) \]

(fn) \[ E \vdash \lambda^m x : \sigma. e \rightarrow Clos(x^T[\mu], e, E) \]

(app) \[ E \vdash x_1 \rightarrow Clos(x^\sigma, e, E') \quad E \vdash x_2 \rightarrow v' \quad E' \vdash x \rightarrow v' \vdash e \rightarrow v \]
\[ E \vdash \@x_1 x_2 \rightarrow v \]

(tfn) \[ E \vdash \Lambda^p_i. e_v \mapsto Clos^p (\tau_i, e_v, E) \]

(let) \[ E \vdash e_1 \rightarrow v_1 \quad E + x \mapsto v_1 \vdash e_2 \rightarrow v \]
\[ E \vdash \text{let} \ x = e_1 \text{ in } e_2 \rightarrow v \]

(tapp) \[ E \vdash x \rightarrow Clos^p (\tau_i, e_v, E') \quad E' \vdash e_v[\mu_j/t_j] \rightarrow v \]
\[ E \vdash x[\tau_j] \rightarrow v \]

Figure 14: Operational Semantics

**Definition 9 (E_H Header Value Environment)**

Suppose \( \Gamma_m; \Gamma_p; \Gamma_i; H; d \vdash e : \sigma \Rightarrow e' \ ; H' \ ; F \). If \( H := [h_1, \ldots, h_n] \), then \( E_H = E_{h_1} \sqcup \ldots \sqcup E_{h_n} \). If \( h_i := (y_i = e_i) \), and \( E_{m}; E_{p}; E_{h_0\ldots h_{i-1}} \vdash e_i \rightarrow v_i \), then \( E_{h_i} := y_i \rightarrow v_i \).

**Definition 10 (H is well founded)**

Suppose \( \Gamma_m; \Gamma_p; \Gamma_i; H; d \vdash e : \sigma \Rightarrow e' \ ; H' \ ; F' \) and \( \Gamma_i(x) = (\forall \tau_i. \sigma, F) \). Suppose further that \( E_m \) is compatible to \( E_{m}' \), and \( E_p \) is compatible to \( E_{p}' \). We say that \( H \) is well founded if, \( \forall x \in \Gamma_i, E_m; E_p; E_i \vdash x[\tau_i] \rightarrow v \) implies that \( E_m'; E_p'; E_H \vdash \!@x[\tau_i]F \rightarrow v' \) and \( v \approx v' \).

In the above definition, note that the environments \( E_m, E_m' \) respect \( \Gamma_m \); the environments \( E_p, E_p' \) respect \( \Gamma_p \); the environment \( E_H \) respects \( \Gamma_H \) and the environment \( E_i \) respects \( \Gamma_i \). As we said above, we shall leave this implicit to help the presentation, but it should be kept in mind that the value environment respects the type environment.

The operational semantics is shown in Figure 14. The environment \( E \) is a generic environment which means that for the source terms, \( E \) is substituted with \( E_m; E_p; E_i \) and for the target terms, \( E \) is substituted with \( E_m'; E_p'; E_H \).

**Lemma 24**

Suppose \( E_m; E_p; E_H \vdash \text{let} \ x = \text{LET}(H_1, e') \text{ in } \text{LET}(H_2, e'') \rightarrow v \) and \( x \) does not occur free in either \( H_1 \) or \( H_2 \). Then \( E_m; E_p; E_H \vdash \text{LET}(H_1 \parallel H_2, \text{let} \ x = e' \text{ in } e'') \rightarrow v \)

34
Proof The proof is exactly similar to the proof for Lemma 21. We only need to replace the type environments $(\Gamma_m;\Gamma_p;\Gamma_H)$ with the corresponding value environments $(E_m;E_p;E_H)$.

\[\square\]

Theorem 25 (Semantic Soundness)

Suppose $\Gamma_m;\Gamma_p;\Gamma_H;H\vdash e : \sigma \Rightarrow e' ; H' ; F'$. Suppose further that $E_m,E'_m$ are compatible and $E_p,E'_p$ are compatible. If $H$ is well formed and $E_m;E_p;E_l \vdash e \leadsto v$, then $E'_m;E'_p;E_H \vdash \text{LET}(H',e') \leadsto v'$ with $v \approx v'$.

Proof The proof is by induction on the structure of $e$.

case int Both the source and the target terms evaluate to $i$.

case var The source term evaluates to $E_m(x)$; the target term to $E'_m(x)$. The equivalence follows since $E_m$ and $E'_m$ are compatible.

case var The source term evaluates to $E_p(x)$; the target term to $E'_p(x)$. The equivalence follows since $E_p$ and $E'_p$ are compatible.

case app In the source term, $x_1$ evaluates to a closure, $E_m(x_1)$, say $v_1$. $x_2$ evaluates to $E_m(x_2)$, say $v_2$. In the target term, $x_1$ evaluates to a closure, $E'_m(x_1)$, say $v'_1$. $x_2$ evaluates to $E'_m(x_2)$, say $v'_2$. Since $E_m$ and $E'_m$ are compatible, $v_1 \approx v'_1$ and $v_2 \approx v'_2$. The required equivalence follows from the definition of the equivalence of closures.

case app2 The proof is exactly similar to the proof for (app). We only need to replace the environment $E_m$ with the environment $E_p$ and the environment $E'_m$ with the environment $E'_p$.

case let By the inductive assumption, if $E_m;E_p;E_l \vdash e_1 \leadsto v_1$, then $E'_m;E'_p;E_H \vdash \text{LET}(H_1,e_1) \leadsto v'_1$ and $v_1 \approx v'_1$. Similarly, by induction on $e_2$, if $E_m[x \mapsto v_1];E_p;E_l \vdash e_2 \leadsto v_2$, then $E'_m[x \mapsto v_1];E'_p;E_H \vdash \text{LET}(H_2,e_2) \leadsto v'_2$ and $v_2 \approx v'_2$. This implies that $E'_m;E'_p;E_H \vdash \text{let } x = \text{LET}(H_1,e'_1) \text{ in } \text{LET}(H_2,e'_2) \leadsto v'_2$. By Lemma 19, $x$ does not occur free in either $H_1$ or $H_2$. But by Lemma 24, this implies that $E'_m;E'_p;E_H \vdash \text{LET}(H_1 || H_2, \text{let } x = e'_1 \text{ in } e'_2) \leadsto v'_2$. The required equivalence follows from this.

case let2 By the inductive assumption, if $E_m;E_p;E_l \vdash e_1 \leadsto v_1$, then $E'_m;E'_p;E_H \vdash \text{LET}(H_1,e_1) \leadsto v'_1$ and $v_1 \approx v'_1$. Similarly, by induction on $e_2$, if $E_m;E_p[x \mapsto v_1];E_l \vdash e_2 \leadsto v_2$, then $E'_m;E'_p[x \mapsto v_1];E_H \vdash \text{LET}(H_2,e'_2) \leadsto v'_2$ and $v_2 \approx v'_2$. Since the variables bound in $H_1$ do not occur in $\text{LET}(H_2,e'_2)$, we can introduce the variables in $H_1$ in the environment while typing $\text{LET}(H_2,e'_2)$. Therefore we have that $E'_m;E'_p;E_H \not\equiv E_H', \vdash e'_1 \leadsto v'_1$ and $E'_m;E'_p[x \mapsto v_1];E_H \not\equiv E_H || \text{LET}(H_2,e'_2) \leadsto v'_2$. The required equivalence follows from here.

case tapp Consider first the case of $(d = 0)$. We need to prove that if $E_m;E_p;E_l \vdash x[\overline{p}] \leadsto v$, then $E'_m;E'_p;E_H \vdash \Lambda\overline{\overline{\overline{p}}}_k.\odot^*x[\overline{p}][\overline{\overline{p}}]L \leadsto v'$ and $v \approx v'$. From the type of $x[\overline{p}]$, we know that $v$ evaluates to a type closure. The target value $v'$ is obviously a type closure. In order to prove the equivalence of the two type closures, we need to prove that if $E_m;E_p;E_l \vdash v[\overline{p}] \leadsto v_1$, then $E'_m;E'_p;E_H \vdash v'[\overline{p}] \leadsto v'_1$, and $v_1 \approx v'_1$. This implies we must prove that if $E_m;E_p;E_l \vdash x[\overline{p}][\overline{\overline{p}}] \leadsto v_1$, then $E'_m;E'_p;E_H \vdash \odot^*x[\overline{p}][\overline{\overline{p}}]L \leadsto v'_1$, and $v_1 \approx v'_1$. But this follows from the definition of well-foundedness of the header environment $E_H$.

Consider now the case of $(d > 0)$. We need to prove that if $E_m;E_p;E_l \vdash x[\overline{p}] \leadsto v$, then $E'_m;E'_p;E_H \vdash \text{LET}(x = x[\overline{p}], \Lambda\overline{\overline{\overline{p}}}_k.\odot^*x[\overline{p}]L) \leadsto v'$ and $v \approx v'$. Since our calculus obeys the value restriction for polymorphic definitions, we can substitute for $z$ in the body of the LET. Therefore we need to prove that $E'_m;E'_p;E_H \vdash \Lambda\overline{\overline{\overline{p}}}_k.\odot^*x[\overline{p}][\overline{\overline{p}}]L \leadsto v'$ and $v \approx v'$. This can be proved in an exactly similar way to the proof of soundness for $(d = 0)$.  

35
**Case tap2** In the source term, $x$ evaluates to $E_p(x) = v\text{(say)}$. In the target term, $x$ evaluates to $E'_p(x) = v'\text{(say)}$. Since the environments are compatible, $v \approx v'$. The required equivalence of the source and target terms follows from this.

**Case tfn** We need to prove that if $E_m; E_p; E_l \vdash \text{let } x = \Lambda t_i.e_1 \text{ in } e_2 \leadsto v_2$, then $E'_m; E'_p; E_H \vdash \text{LET}(H' \ | \ H_2, e'_2) \leadsto v'_2 \text{ and } v_2 \approx v'_2$. We will first prove that the augmented environment $H' \ | \ H$ is well founded during the translation of $e_2$. By definition, $E_H := \vdash x \rightarrow \text{Clos}^t(\overline{t}_i, \text{LET}(H_1, \lambda^x F_1, e_1), E'_m, E'_p, E_H)$

Now for any compatible pairs of environments; $E_m$, $E'_m$, $E_p$, $E'_p$ and environments $E_l$, $E'_l$, we need to prove that if $E_m; E_p; E_l \vdash x[\overline{t}_i] \leadsto v$ then $E'_m; E'_p; E'_l \vdash \star^* x[\overline{t}_i] \leadsto v' \text{ and } v \approx v'$. This implies we need to prove that if $E_m; E_p; E_l \vdash e_1[\mu_i/t_i] \leadsto v$ then $E'_m; E'_p; E'_l \vdash \star^* x[\overline{t}_i] \leadsto v' \text{ and } v \approx v'$. We will simplify the evaluation of the target term. $E'_m; E'_p; E'_l \vdash x[\overline{t}_i] \rightarrow E'_m; E'_p; E_H \vdash \text{LET}(H_1, \lambda^x F_1, e_1)[\mu_i/t_i]$. This in turn reduces to $\text{Clos}(F_1, \overline{t}_i, e'_1[\mu_i/t_i], E'_m, E'_p, E_H \equiv E_H[\mu_i/t_i])$. Therefore, the evaluation of the target term reduces to $E'_m; E'_p; E_H \vdash \star^* \text{Clos}(F_1, \overline{t}_i, e'_1[\mu_i/t_i], E'_m, E'_p, E_H \equiv E_H[\mu_i/t_i])$. $F_1$. This in turn reduces to $E'_m \equiv E'_m(F_1); E'_p; E_H \equiv E_H[\mu_i/t_i] \vdash e'_1[\mu_i/t_i] \leadsto v'$. But by induction on the translation of $e_1$, we get that $v \approx v'$. Hence $H' \ | \ H$ is well founded with respect to $e_l[x \rightarrow v]$. And so we can apply the inductive hypothesis to the translation of $e_2$ which leads directly to the semantic soundness result.

**Case tfn2** We know by Lemma 18 that $H_1 = H_2 = \text{nil}$. Therefore we need to prove that if $E_m; E_p; E_l \vdash \text{let } x = \Lambda t_i.e_1 \text{ in } e_2 \leadsto v$ then $E'_m; E'_p; E_H \vdash \text{let } x = \Lambda t_i.e'_1 \text{ in } e'_2 \leadsto v'$ and $v \approx v'$. We will first prove that the header $H' \ | \ H$ is well founded. By definition, $E_H := \vdash x \rightarrow \text{Clos}^t(\overline{t}_i, e'_1, E'_m, E'_p, E_H)$. Now for any compatible pairs of environments; $E_m$, $E'_m$, $E_p$, $E'_p$ and $E_l$, $E'_l$ and environments $E_l$, $E'_l$, we need to prove that if $E_m; E_p; E_l \vdash x[\overline{t}_i] \leadsto v$ then $E'_m; E'_p; E'_l \vdash x[\overline{t}_i] \leadsto v'$ and $v \approx v'$. This in turn implies that we need to prove that if $E_m; E_p; E_l \vdash e_1[\mu_i/t_i] \leadsto v$ then $E'_m; E'_p; E_H \equiv E_H[\mu_i/t_i] \vdash e'_1[\mu_i/t_i] \leadsto v'$. But this follows from the inductive assumption on the translation of $e_1$ (since $H_1$ is nil). We can now apply the inductive hypothesis to the translation of $e_2$ which leads directly to the semantic soundness result.

**Case fct** By induction, if $E_m; E_p[x \rightarrow v_2]; E_l \vdash e \leadsto v$, then $E'_m; E'_p[x \rightarrow v'_2]; E_H \vdash \text{LET}(H', e') \leadsto v'$ and $v \approx v'$. By Lemma 18, $H'$ is nil. Therefore we need to prove that if $E_m; E_p; E_l \vdash \lambda^{m.x} : \sigma.e \leadsto v_1$, then $E'_m; E'_p; E_H \rightarrow \lambda^m.x : \sigma.e' \leadsto v'_1 \text{ and } v_1 \approx v'_1$. By the definition of equivalence of closures, this follows directly from the inductive hypothesis.

**Case fn** For both ($d = 0$) and ($d > 0$) cases, we need to prove that if $E_m; E_p; E_l \vdash \lambda^c.x : T(\mu).e \leadsto v$, then $E'_m; E'_p; E_H \rightarrow \text{LET}(H', \lambda^c.x : T(\mu).e') \leadsto v'$ and $v \approx v'$. By the inductive hypothesis, we know that if $E_m[x \rightarrow v_1]; E_p; E_l \vdash e \leadsto v_2$, then $E'_m[x \rightarrow v'_1]; E'_p; E_H \vdash \text{LET}(H', e') \leadsto v'_2 \text{ and } v_2 \approx v'_2$. By Lemma 19, $x$ does not occur free in $H'$. The header $H'$ actually consists of $[h_1, \ldots, h_n]$ and each $h_i := y_i = e_\sigma$. Consider the expression $\text{LET}(H', e')$ as $\text{let } y_1 = e_1 \text{ in } e''$ where $e'' = \text{LET}(h_2, \ldots, h_n, e')$. From the evaluation rules we can deduce that $E'_m[x \rightarrow v'_1]; E'_p; E_H \vdash e_1 \leadsto u_1$ (for some $u_1$) and $E'_m[x \rightarrow v'_1]; E'_p; E_H[y_1 \rightarrow u_1] \vdash e'' \leadsto v'_2$. But since $x$ does not occur in $e_1$, we can remove the binding of $x$ from the environment while evaluating $e_1$. Therefore we get that $E'_m; E'_p; E_H \vdash e_1 \leadsto u_1 \text{ and } E'_m[x \rightarrow v'_1]; E'_p; E_H[y_1 \rightarrow u_1] \vdash e'' \leadsto v'_2$. By continuing this process of splitting $H'$, we can prove the required equivalence.

\[\Box\]