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, $x[\alpha]$ denotes type application and $x(e)$ denotes term application.

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

......

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

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

Here, `f` is a polymorphic function defined inside function `pair`; it refers to the parameter `x` of `pair`, so `f` cannot be easily lifted outside `pair`. Function `main` executes a loop: in each iteration, it selects an element `elem` of the array `a` and then performs some computation (i.e, `pair`) on it. Executing the function `doit` results in three type applications arising from the `Array.sub` function, `pair`, and `f`. In each iteration, `sub` and `pair` are applied to types $\alpha * \alpha$ and α respectively. A clever compiler may do a *loop-invariant removal* [1] to avoid the repeated type construction (e.g., $\alpha * \alpha$) and application (e.g., `pair`$[\alpha]$). But optimizing type applications such as `f[s*s]` is less obvious; `f` is nested inside `pair`, and its repeated type applications are not apparent in the `doit` function (specially, if one considers that `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 `doit` is called, `pair`$[\alpha]$ gets executed and then every time `pair` is called, `f[s*s]` will be executed. Since `loop` calls `doit` repeatedly and each such call generates type applications of `pair` and `f`, we are forced to incur the overhead of repeated type construction and application. If the type representation is complicated, this is clearly expensive.

In this paper, we present an algorithm that minimizes the cost of run-time type passing. More specifically, the optimization eliminates all type application inside any core-language function — it guarantees that the amount of type information constructed at runtime is a static constant. This guarantee 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 (λ); they are nested only inside type abstractions (Λ). All type applications are now performed once and for all at the beginning of execution of each compilation unit. In essence, the code 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.

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

Figure 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 $\overline{t_i}$ (and $\overline{\mu_i}$) to denote a sequence of type variables $t_1, ..., t_n$ (and type constructors) so $\forall\overline{t_i}.\,\mu$ is equivalent to $\forall t_1 \ldots \forall t_n.\mu$. The *vterms* ($e_v$) denote values – terms that are free of side-effects.

There are several aspects of this calculus that are worth noting. First, we restrict polymorphic definitions to value expressions (e_v) only, so that moving type applications and polymorphic definitions is semantically sound [30]. Variables introduced by normal λ-abstraction are always monomorphic, and polymorphic functions are introduced only by the **let** construct. In our calculus, type applications of polymorphic functions are never curried which implies that polymorphic functions do not escape. Therefore in the algorithm in Figure 2, the *exp* rule assumes that the variable is monomorphic. The *tapp* rule also assumes that the type application is not curried and therefore the newly introduced variable `v` (bound to the lifted type application) is monomorphic and is not instantiated by further type application. Finally, following SML [14], polymorphic functions are not

initialization and module linkage (e.g., functor application) in a ML-style module language.

recursive. [2] *This restriction is crucial to proving that all type applications can be lifted to the top level.*

Throughout the paper we take a few liberties with the syntax: we allow ourselves infix operators, multiple definitions in a single `let` expression to abbreviate a sequence of nested `let` expressions, and term applications that are at times not in A-Normal form [4]. We also use indentation to indicate the nesting.

2.2 Informal description

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

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

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

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

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

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

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

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 occuring 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.

[2] Our 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 λx (of the `pair` function), and the type applications `Array.sub[α*α]` and `pair[α]` are at depth 2 since they occur inside the λa and λi (of the `main` function). We want to lift all of these type applications to depth zero. Translating `main` first, the resulting code becomes —

```
pair = Λs.λx:s*s.
         let f = Λt.λy:t. ... (x , y)
         in  ... f[s*s](x) ...
......
main = Λα.
         let v1 = Array.sub[α*α]
             v2 = pair[α]
         in  λa:α.
                let
                  doit = λi:Int.
                           let elem = v1(a,i)
                           in ... v2(elem) ...
                  loop = λn1:Int.λn2:Int.λg:Int→Unit.
                           if n1 <= n2
                             (g(n1);
                              loop(n1+1,n2,g))
                           else ()
                in loop(1,n,doit)
```

We then lift the type application of `f` (inside `pair`). This requires us to lift `f`'s definition by abstracting over its free variables. In the resulting code, all type applications occur at depth zero. Therefore v_1, v_2 and v_3 get evaluated now at the beginning of execution. When the function `loop` runs through the array and calls function `doit`, none of the type applications are repeated – the type specialised functions v_1, v_2 and v_3 are used instead.

```
pair = Λs.
         let f = Λt.λz:s*s.λy:t. ... (z , y)
             v3 = f[s*s]
         in  λx:s*s. ... (v3(x))(x) ...
......
main = Λα.
         let v1 = Array.sub[α*α]
             v2 = pair[α]
         in  λa:α.
                let doit =
                      λi:Int.
                          let elem = v1(a,i)
                          in  ... v2(elem) ...
                    loop =
                      λn1:Int.λn2:Int.λg:Int→Unit.
                          if n1 <= n2
                            (g(n1);
                             loop(n1+1,n2,g))
                          else ()
                in loop(1,n,doit)
```

2.3 Formalization

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

The environment Γ maps a variable to its type and to a list of the free variables in its definition. In the algorithm, we use standard notation for lists and operations on lists. The functions λ^* and $@^*$ are defined so that $\lambda^* L.\, e$ and $@^* v L$ reduce to $\lambda x_1 : \mu_1 . \ldots \lambda x_n : \mu_n . e$ and $@(\ldots(@ v x_1)\ldots) x_n$, respectively, where $L = \{x_1 : \mu_1, \ldots, x_n : \mu_n\}$. We use $\|$ to denote the append operation and $::$ to denote the consing operation.

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

The translation of monomorphic `let` expressions is similar. We translate each of the subexpressions replacing the old terms with the translated terms and return this as the result of the translation. The result header, H, 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 (*tfn*), 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 `let` expression(e_2). The result of the translation is only e_2'; the polymorphic function introduced by the **let** is added to the result header H_r, so that, it is lifted to the top level. The polymorphic function body (in H_r) is closed by abstracting over its free variables F_1 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 (*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 LET(H,e'), the term e' does not contain any type application and H does not contain any type application nested inside a value(λ) abstraction.

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 (*tfn*) and (*tapp*) cases here.

case tfn 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.

□

Theorem 2 (Full Lifting)

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

Proof The theorem follows from Proposition 1.

□

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.

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

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

1. After translating **f's** body

```
e'1 = λu:t3.λv:t4. (v,u,x)
H1 = []
F1 = {x:t1}
```

2. Now **g's** body is translated

```
e'1 = λz:t5. @(@(@v1x)z)x
H1 = v1 = f[t5][t1]
F1 = {x:t1}
```

3. Now the body of the inner let

```
e'2 = @(@v2x)y
H2 = v2 = g[t2]
F2 = {x:t1,y:t2}
```

4. The inner let as a whole returns

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

5. For the outer let

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

6. The outer let as a whole returns

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

7. After translating the lambda abstraction

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

The final translated code with all type applications lifted:

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

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 langauge, 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 λ-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 μ. The terms are an explicitly typed λ-calculus (but in A-normal form) with explicit constructor abstraction and application.

(*cons*)	μ	$::=$	$t \mid \texttt{Int} \mid \mu_1 \to \mu_2$
(*types*)	σ	$::=$	$T(\mu) \mid \sigma_1 \to \sigma_2 \mid \forall t.\sigma$
(*terms*)	e	$::=$	$i \mid x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid @x_1 x_2 \mid \lambda^c x\colon T(\mu).e \mid \lambda^m x\colon\sigma.e \mid \texttt{let}\ x = \Lambda\overline{t_i}.e_v\ \texttt{in}\ e_2 \mid x[\overline{\mu}_i]$
(*values*)	e_v	$::=$	$i \mid x \mid \texttt{let}\ x = e_v\ \texttt{in}\ e'_v \mid \lambda^c x\colon T(\mu).e \mid \lambda^m x\colon\sigma.e \mid \texttt{let}\ x = \Lambda\overline{t_i}.e_v\ \texttt{in}\ e'_v \mid x[\overline{\mu}_i]$

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` = $\Lambda\overline{t_i} :: \overline{k_i}.\lambda^m x : \sigma.e$). The variable `x` in the functor definition is polymorphic since the parameterised structure may contain polymorphic components. In the functor body `e`, the polymorphic components of `x` 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, $\lambda^c x\colon T(\mu).e$ denotes functions, whereas $\lambda^m x\colon\sigma.e$ 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 (λ^m) occur outside function abstractions (λ^c).
- No partial type application occurs inside a function abstraction (λ^c).
- No functor application occurs inside a function abstraction (λ^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 occuring outside all function abstractions only.* Note that depth 0 terms may not occur outside all abstractions since they may be nested inside functor abstractions. 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(λX). 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 (λv), it is not lifted outside the functor abstraction (λX). The resulting code is shown in Figure 6.

$$(int) \qquad \Gamma; d \vdash^w i : Int$$

$$(var) \qquad \Gamma; d \vdash^w x : T(\mu)$$

$$(app1) \qquad \frac{\Gamma(x_1) = \sigma_1 \rightarrow \sigma_2 \qquad \Gamma(x_2) = \sigma_1}{\Gamma; d = 0 \vdash^w @x_1 x_2 : \sigma_2}$$

$$(app2) \qquad \frac{\Gamma(x_1) = T(\mu_1 \rightarrow \mu_2) \qquad \Gamma(x_2) = T(\mu_1)}{\Gamma; d \vdash^w @x_1 x_2 : T(\mu_2)}$$

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

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

$$(tfn) \qquad \frac{\Gamma; d \vdash^w e_v : \sigma \qquad \Gamma[x \mapsto \forall \overline{t}_i.\sigma]; d \vdash^w e' : \sigma'}{\Gamma; d \vdash^w \texttt{let } x = \Lambda \overline{t}_i.e_v \texttt{ in } e' : \sigma'}$$

$$(tapp) \qquad \frac{\Gamma(x) = \forall \overline{t}_i.\sigma \qquad j < i}{\begin{array}{c}\Gamma; d \vdash^w x[\overline{\mu}_i] : \sigma[\mu_i/t_i] \\ \Gamma; d = 0 \vdash^w x[\overline{\mu}_j] : \sigma[\mu_j/t_j]\end{array}}$$

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

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

Figure 4: The Well Formedness Relation $\vdash^w$

```
F = Λt₀.λᵐX:S.
     f = λᶜv.
           let id = Λt₁.λᶜx₂.x₂
               .....
               v₁ = ... id[Int](3) ....
           in  v₁
     v₂ = (#1(X))[t₀]
     ....
```

Figure 5: Example FLINT code

```
F = Λt₀.λᵐX:S.
     f = let
           id = Λt₁.λᶜx₂.x₂
           z₁ = id[Int]
           .. (Other type expressions in f's body)..
         in  λᶜv.
               let  ..... (type lifted body of f)
                      v₁ = ... z₁(3) ....
               in   v₁
     v₂ = (#1(X))[t₀]
     ....
```

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'; Γ 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 σ for which the algorithm infers $\emptyset; 0 \vdash e : \sigma \Rightarrow e'; H; \emptyset$ is $\mathtt{LET}(H, e')$, where the function $\mathtt{LET}(H, e)$ expands a list of bindings $H = [\langle x_1, e_1\rangle, \ldots, \langle x_n, e_n\rangle]$ and a term e into the term $\mathtt{let}\ x_1 = e_1\ \mathtt{in}\ \ldots \mathtt{let}\ x_n = e_n\ \mathtt{in}\ e$.

The (*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 (*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 (*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 (*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 (*fn*) rule deals with functions. The algorithm first collects the type applications and polymorphic functions occuring 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 presrvation and semantic soundness.

In the algorithm, we donot abstract over polymorphic variables. (Hence we donot add them to the free variable list). *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

$$(int)\qquad \frac{}{\Gamma; d \vdash i : \texttt{Int} \Rightarrow i \;;\; nil \;;\; \emptyset}$$

$$(var)\qquad \frac{\Gamma(x) = \langle T(\mu), \_\rangle}{\Gamma; d \vdash x : T(\mu) \Rightarrow x \;;\; nil \;;\; \{x : T(\mu)\}} \qquad \frac{\Gamma(x) = \langle \sigma, \emptyset^f\rangle}{\Gamma; d \vdash x : \sigma \Rightarrow x \;;\; nil \;;\; \emptyset}$$

$$(app)\qquad \frac{\Gamma; d \vdash x_1 : T(\mu \to \mu_1) \Rightarrow x_1 \;;\; nil \;;\; \{x_1 : T(\mu \to \mu_1)\} \quad \Gamma; d \vdash x_2 : T(\mu) \Rightarrow x_2 \;;\; nil \;;\; \{x_2 : T(\mu)\}}{\Gamma; d \vdash @x_1 x_2 : T(\mu_1) \Rightarrow @x_1 x_2 \;;\; nil \;;\; \{x_1 : T(\mu \to \mu_1), x_2 : T(\mu)\}}$$

$$(app2)\qquad \frac{\Gamma; d \vdash x_1 : \sigma_2 \to \sigma_1 \Rightarrow x_1 \;;\; nil \;;\; \emptyset \quad \Gamma; d \vdash x_2 : \sigma_2 \Rightarrow x_2 \;;\; nil \;;\; \emptyset}{\Gamma; d \vdash @x_1 x_2 : \sigma_1 \Rightarrow @x_1 x_2 \;;\; nil \;;\; \emptyset}$$

$$(let)\qquad \frac{\Gamma; d \vdash e_1 : T(\mu_1) \Rightarrow e_1' \;;\; H_1 \;;\; F_1 \quad \Gamma[x \mapsto \langle T(\mu_1), nil\rangle]; d \vdash e_2 : \sigma_2 \Rightarrow e_2' \;;\; H_2 \;;\; F_2}{\Gamma; d \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma_2 \Rightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' \;;\; H_1 \parallel H_2 \;;\; F_1 \;\cup\; F_2 \backslash \{x : T(\mu_1)\}}$$

$$(let2)\qquad \frac{\Gamma; d \vdash e_1 : \sigma_1 \Rightarrow e_1' \;;\; H_1 \;;\; F_1 \quad \Gamma[x \mapsto \langle \sigma_1, nil\rangle]; d \vdash e_2 : \sigma_2 \Rightarrow e_2' \;;\; H_2 \;;\; F_2}{\Gamma; d \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma_2 \Rightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ \texttt{LET}(H_2, e_2') \;;\; H_1 \;;\; F_1 \;\cup\; F_2}$$

$$(tfn)\qquad \frac{\begin{array}{c}\Gamma; d \vdash e_1 : \sigma_1 \Rightarrow e_1' \;;\; H_1 \;;\; F_1 \qquad H' = \langle x = \Lambda \bar{t}_i.\texttt{LET}(H_1, \lambda^* F_1.e_1')\rangle \\ \Gamma[x \mapsto \langle \forall \bar{t}_i.\sigma_1, F_1\rangle]; d \vdash e_2 : \sigma_2 \Rightarrow e_2' \;;\; H_2 \;;\; F_2\end{array}}{\Gamma; d > 0 \vdash \texttt{let}\ x = \Lambda \bar{t}_i.e_1\ \texttt{in}\ e_2 : \sigma_2 \Rightarrow e_2' \;;\; H' \parallel H_2 \;;\; F_2}$$

$$(tfn2)\qquad \frac{\begin{array}{c}\Gamma; d = 0 \vdash e_1 : \sigma_1 \Rightarrow e_1' \;;\; H_1 \;;\; F_1 \qquad H_1 = nil \\ \Gamma[x \mapsto \langle \forall \bar{t}_i.\sigma_1, nil\rangle]; d = 0 \vdash e_2 : \sigma_2 \Rightarrow e_2' \;;\; H_2 \;;\; F_2 \qquad H_2 = nil\end{array}}{\Gamma; d = 0 \vdash \texttt{let}\ x = \Lambda \bar{t}_i.e_1\ \texttt{in}\ e_2 : \sigma_2 \Rightarrow \texttt{let}\ x = \Lambda \bar{t}_i.e_1'\ \texttt{in}\ e_2' \;;\; nil \;;\; F_1 \;\cup\; F_2}$$

$$(tapp)\qquad \frac{\Gamma(x) = \langle \forall \bar{t}_i.\sigma, F\rangle \qquad k = i - j}{\begin{array}{c}\Gamma; d > 0 \vdash x[\overline{\mu}_j] : [\mu_j/t_j]\forall \bar{t}_k.\sigma \Rightarrow \texttt{let}\ x = \Lambda \bar{t}_k.@^*(v[\bar{t}_k])F\ \texttt{in}\ x \;;\; v = x[\overline{\mu}_j] \;;\; F \\ \Gamma; d = 0 \vdash x[\overline{\mu}_j] : [\mu_j/t_j]\forall \bar{t}_k.\sigma \Rightarrow \texttt{let}\ x = \Lambda \bar{t}_k.@^*(x[\overline{\mu}_j][\bar{t}_k])F\ \texttt{in}\ x \;;\; nil \;;\; F\end{array}}$$

$$(fct)\qquad \frac{\Gamma[x \mapsto \langle \sigma, \emptyset^f\rangle]; d = 0 \vdash e : \sigma' \Rightarrow e' \;;\; H \;;\; F \qquad H = nil}{\Gamma; d \vdash \lambda^m x : \sigma.e : \sigma \to \sigma' \Rightarrow \lambda^m x : \sigma.e' \;;\; nil \;;\; F}$$

$$(fn)\qquad \frac{\Gamma[x \mapsto \langle T(\mu), nil\rangle]; d + 1 \vdash e : T(\mu') \Rightarrow e' \;;\; H \;;\; F}{\begin{array}{c}\Gamma; d > 0 \vdash \lambda^c x : T(\mu).e : T(\mu \to \mu') \Rightarrow \lambda^c x : T(\mu).e' \;;\; H \;;\; F \backslash \{x : T(\mu)\} \\ \Gamma; d = 0 \vdash \lambda^c x : T(\mu).e : T(\mu \to \mu') \Rightarrow \texttt{LET}(H, \lambda^c x : T(\mu).e') \;;\; nil \;;\; F \backslash \{x : T(\mu)\}\end{array}}$$

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 (λ^c).

Proof In Figure 4, the depth is zero either initially, before any (λ^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 (λ^c), but the depth of e' is zero. This is possible only if an occurence of rule (*fct*) inside the (λ^c) resets the depth to zero. But this implies an occurence of the (*fct*) rule at ($d > 0$), and since e is well formed, this cannot happen. Therefore the above proposition is true.

$\square$

Proposition 4

Suppose $\Gamma; d \vdash e : \sigma \Rightarrow e' \ ; \ H \ ; \ F$ and e is a well formed expression. If x has type $\forall \overline{t}_i.\sigma$ and $x[\overline{\mu}_j]$ 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.

$\square$

Proposition 5

Suppose $\Gamma; d \vdash e : \sigma \Rightarrow e' \ ; \ H \ ; \ F$. Suppose further that e is well formed and contains the subterm $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ 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 σ, the term $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ must satisfy rule (*let1*) which restricts the `Let` definition to $d = 0$.

$\square$

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 $@^* vF$.

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

$\square$

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 (λ^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 (λ^c). Therefore H' does not contain any type application nested inside a (λ^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 (λ^c).

case fct 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 (λ^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 $\texttt{LET}(H, e')$ does not have a type application nested inside a function abstraction (λ^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.

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

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 occuring 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 $t_i's$ as free type variables. The advantage of dumping all the expressions is that the *de Bruijn* depth of the terms in H_1 remains the same even after lifting. 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)

$$
\begin{array}{l}
\Lambda\alpha\beta\gamma.\lambda x_1. \\
\ \ \texttt{let} \\
\ \ \ \ y_1 = \Lambda t_1.\lambda x_2.\ \texttt{@(f}[t_1*\alpha]\texttt{)}x_2\ldots \\
\ \ \ \ y_2 = \Lambda t_3.\lambda x_3.\ \texttt{@(}y_1[\beta*t_3]\texttt{)}x_3\ldots \\
\ \ \ \ y_3 = \Lambda t_5.\lambda x_4.\ \texttt{@(}y_2[\gamma*t_5]\texttt{)}x_4\ldots \\
\ \ \texttt{in @(}y_3[\beta*\gamma]\texttt{)}x_1
\end{array}
$$

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

$$
\begin{array}{l}
\ \ \ \ y_1 = \Lambda\alpha t_1.\lambda x_2.\ \texttt{@(f}[t_1*\alpha]\texttt{)}x_2\ldots \\
\ \ \ \ y_2 = \Lambda\alpha\beta t_3.\lambda x_3.\ \texttt{@(}y_1[\alpha][\beta*t_3]\texttt{)}x_3\ldots \\
\ \ \ \ y_3 = \Lambda\alpha\beta\gamma t_5.\lambda x_4.\ \texttt{@(}y_2[\alpha][\beta][\gamma*t_3]\texttt{)}x_4\ldots \\
\ \ \texttt{expr} = \ \Lambda\alpha\beta\gamma.\lambda x_1.\ \texttt{@(}y_3[\alpha][\beta][\gamma][\beta*\gamma]\texttt{)}x_1
\end{array}
$$

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

$$
\begin{array}{l}
\Lambda\alpha\beta\gamma. \\
\ \ \texttt{let} \\
\ \ \ \ y_1 = \Lambda t_1. \\
\ \ \ \ \ \ \ \ \texttt{let}\ v_1 = \texttt{f}[t_1*\alpha] \\
\ \ \ \ \ \ \ \ \texttt{in}\ \ \lambda x_2.\ \texttt{@}v_1x_2\ldots \\
\ \ \ \ y_2 = \Lambda t_3. \\
\ \ \ \ \ \ \ \ \texttt{let}\ v_2 = y_1[\beta*t_3] \\
\ \ \ \ \ \ \ \ \texttt{in}\ \ \lambda x_3.\ \texttt{@}v_2x_3\ldots \\
\ \ \ \ y_3 = \Lambda t_5. \\
\ \ \ \ \ \ \ \ \texttt{let}\ v_3 = y_2[\gamma*t_5] \\
\ \ \ \ \ \ \ \ \texttt{in}\ \ \lambda x_4.\ \texttt{@}v_3x_4\ldots \\
\ \ \ \ v_4 = y_3[\beta*\gamma] \\
\ \ \texttt{in}\ \lambda x_1.\ \texttt{@}v_4x_1
\end{array}
$$

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.

```
Λαβγ.
  let u = β*γ
  λx₁.
     let
       y₁ = Λt₁.
              let u₁ = f[t₁*α]
              in  λx₂. @u₁x₂ ...
       y₂ = Λt₃.
              let u₂ = y₁[β*t₃]
              in  λx₃. @u₂x₃ ...
       y₃ = Λt₅.
              let u₃ = y₂[γ*t₅]
              in  λx₄. @u₃x₄ ...
     in @(y₃[u])x₁
```

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.

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

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

The advantage of his method is that he eliminates the runtime construction of types. 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

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

The actual type of `f` is $Eq[a] => a \rightarrow a \rightarrow Bool$, but after context reduction, we get the type shown in the example. Here $[a]$ means a list of elements of type a. $Eq\ a$ means that the type a must be an instance of the Equality Class. $Eq\ [a]$ means that the type $List\ of\ a's$ must be an instance of the Equality Class. Function `f` in the example above, has type $a \rightarrow a \rightarrow Bool$, with a being an instance of the Equality class. Jones optimises this by constructing a dictionary for *Eq [a]* at the call site of `f`; rather than pass a dictionary for $Eq\ a$ and construct the dictionary for *Eq [a]* 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 $Eq[a] => a \rightarrow a \rightarrow Bool$. 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(σ_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 Morisett [7] proposed an alternative approach for compiling polymorphism where types are passed as arguments to polymorphic routines in order to determine the representation of an object. 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

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

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

[3] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, 1993.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[22] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pages 408–425. Springer-Verlag, Berlin, 1974.

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

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

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

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

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

[28] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 181–192. ACM Press, 1996.

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

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

A Proofs For The Core-ML Algorithm

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

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

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

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

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

$$(tfn) \qquad \frac{\begin{array}{c}\Gamma_m; \Gamma_l; H \vdash e_1 : \mu_1 \Rightarrow e_1'; H_1; F_1 \qquad H' = [\langle x = \Lambda\overline{t_i}.\texttt{LET}(H_1, \lambda^* F_1.e_1')\rangle] \\ \Gamma_m; \Gamma_l[x \mapsto \langle \forall \overline{t_i}.\mu_1, F_1\rangle]; H \parallel H' \vdash e_2 : \mu_2 \Rightarrow e_2'; H_2; F_2\end{array}}{\Gamma_m; \Gamma_l; H \vdash \texttt{let}\ x = \Lambda\overline{t_i}.e_1\ \texttt{in}\ e_2 : \mu_2 \Rightarrow e_2'; H' \parallel H_2; F_2}$$

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

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 $F = \{x_1 : \mu_1, ..., x_n : \mu_n\}$, then $\lambda^* F.e$ denotes $\lambda x_1 : \mu_1.(...(\lambda x_n : \mu_n.e)..)$. Similarly $@^* zF$ denotes $@(..(@zx_1)..)x_n$.

Notation $Ty(L)$

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

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

The variable environment is split up into 2 environments – Γ_m which binds the monomorphic variables, and Γ_l which binds the variables introduced by polymorphic let expressions.

Lemma 9

Suppose $\Gamma_m; \Gamma_l; H \vdash e : \sigma \Rightarrow e'; H'; F$. 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 occuring free in e' are z and the variables in F. The variable z is newly introduced and hence cannot occur in Γ_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.

$\square$

Lemma 10

Suppose $\Gamma_m; \Gamma_l; H \vdash e : \sigma \Rightarrow e'; H'; F$. 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 donot follow directly from the inductive assumption.

case tapp The only free variable in the header is x which occurs in Γ_l.

case tfn By Lemma 9, we know that $\lambda^* F_1.e'_1$ is closed with respect to variables in Γ_m. By induction on e_1, we know that H_1 is closed with respect to variables in Γ_m. This implies that H' is closed with respect to variables in Γ_m. By induction on e_2, we know that H_2 is closed with respect to variables in Γ_m, which proves the lemma.

$\square$

Lemma 11

The variables bound in Γ_m and Γ_l are mutually disjoint; and the variables bound in Γ_m and Γ_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.

$\square$

A.1 Type Preservation

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

Definition 1 ($\texttt{LET}(H, e)$)

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

Definition 2 (Γ_H **Header Type Environment**)

Suppose $\Gamma_m; \Gamma_l; H \vdash e : \sigma \Rightarrow e'; H'; F$. 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 $\Gamma_{h_i} := y_i \mapsto \sigma_i$.

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

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

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

$$(tfn) \qquad \frac{\Gamma \vdash e_v : \mu_1 \quad \Gamma \uplus \{x : \forall \overline{t_i}.\mu_1\} \vdash e : \mu_2 \qquad t_i \textit{ not free in } \Gamma}{\Gamma \vdash \texttt{let } x = \Lambda \overline{t_i}.e_v \texttt{ in } e : \mu_2}$$

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

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

Figure 10: Static Semantics

Definition 3 (H is well formed with respect to Γ_l)

H is well formed iff $\Gamma_l(x) = (\forall \overline{t}_i.\mu, F)$ implies that $\Gamma_H(x) = \forall \overline{t}_i.Ty(F) \to \mu$.

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

Lemma 12

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

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

The header $H_2 ::= [h_1, \ldots, h_n]$ where each $h_i ::= y_i = e_i$. Therefore write $\texttt{LET}(H_2, e'')$ as $\texttt{let } y_1 = e_1 \texttt{ in } e$ where $e = \texttt{LET}([h_2, \ldots, h_n], e'')$. When we consider the typing rules for this expression, we get that $\Gamma_m[x \mapsto \mu];\Gamma_H \uplus \Gamma_{H_1} \vdash e_1 : \mu_1$ (for some μ_1) and $\Gamma_m[x \mapsto \mu];\Gamma_H \uplus \Gamma_{H_1} \uplus [y_1 \mapsto \mu_1] \vdash e : \mu_2$. 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 – $\Gamma_m;\Gamma_H \uplus \Gamma_{H_1} \vdash e_1 : \mu_1$ and $\Gamma_m;\Gamma_H \uplus \Gamma_{H_1} \uplus [y_1 \mapsto \mu_1] \vdash e' : \mu$ and $\Gamma_m[x \mapsto \mu];\Gamma_H \uplus \Gamma_{H_1} \uplus [y_1 \mapsto \mu_1] \vdash e : \mu_2$. This implies that $\Gamma_m;\Gamma_H \uplus \Gamma_{H_1} \vdash \texttt{let } y_1 = e_1 \texttt{ in let } x = e' \texttt{ in } e : \mu_2$. By continuing this process of breaking H_2 into its components, in the end, we get the following typing equations – $\Gamma_m;\Gamma_H \uplus \Gamma_{H_1} \uplus \Gamma_{H_2} \vdash e' : \mu$ and $\Gamma_m[x \mapsto \mu];\Gamma_H \uplus \Gamma_{H_1} \uplus \Gamma_{H_2} \vdash e'' : \mu_2$. which proves the lemma.

□

Lemma 13

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

Proof The header $H' ::= [h_1, \ldots, h_n]$ with each $h_i ::= y_i = e_i$. Therefore write $\mathtt{LET}(H',e)$ as $\mathtt{let}\ y_1 = e_1\ \mathtt{in}\ e''$ where $e'' ::= \mathtt{LET}([h_2,\ldots,h_n],e)$. The typing rule for the expression $\lambda x:\mu.\mathtt{LET}(H',e)$ can be written as $\Gamma_m[x \mapsto \mu];\Gamma_H \vdash e_1 : \mu_1$ (for some type μ_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 $\Gamma_m;\Gamma_H \vdash \mathtt{let}\ y_1 = e_1\ \mathtt{in}\ \lambda x\!:\!\mu.e'' : \mu \to \mu'$. Continuing to break H' like this, we finally reach $\Gamma_m;\Gamma_H \vdash \mathtt{LET}(H',\lambda x\!:\!\mu.e) : \mu \to \mu'$.

□

Theorem 14 (Type Preservation)

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

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

case int Both the source and target terms have type `Int`.

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

case app By an argument similar to the (*var1*) case, we can deduce that $\Gamma_m;\Gamma_H \vdash x_1 : \mu_1 \to \mu_2$ and $\Gamma_m;\Gamma_H \vdash x_2 : \mu_1$. This implies that $\Gamma_m;\Gamma_H \vdash @x_1x_2 : \mu_2$.

case let By the inductive hypothesis, $\Gamma_m;\Gamma_H \vdash \mathtt{LET}(H_1,e_1') : \mu_1$ and $\Gamma_m[x \mapsto \mu_1];\Gamma_H \vdash \mathtt{LET}(H_2,e_2') : \mu_2$. Since H_2 and H_1 are closed with respect to variables in Γ_m(Lemma 10), x does not occur free in H_2 and H_1. Applying Lemma 12 leads to the preservation theorem.

case tapp From the antecedent we know that $\Gamma_l(x) = (\forall \overline{t}_i.\mu, F)$. By the well formedness of H, we know that $\Gamma_H(x) = \forall \overline{t}_i.Ty(F) \to \mu$. Therefore $\Gamma_m;\Gamma_H \vdash x[\overline{\mu}_i] : Ty(F) \to \mu[\mu_i/t_i]$. Since F consists of the free variables of x, the type $Ty(F)$ 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.

case tfn We need to prove that $\Gamma_m;\Gamma_H \vdash \mathtt{LET}(H' \parallel H_2, e_2') : \mu_2$ which implies that $\Gamma_m;\Gamma_H \uplus \Gamma_{H'} \vdash \mathtt{LET}(H_2,e_2') : \mu_2$ must be true. By definition, $\Gamma_{H'} ::= x \mapsto \sigma'$ where $\Gamma_m;\Gamma_H \vdash \Lambda \overline{t}_i.\mathtt{LET}(H_1, \lambda^* F_1.e_1') : \sigma'$. By the inductive hypothesis, we know that $\Gamma_m;\Gamma_H \vdash \mathtt{LET}(H_1,e_1') : \mu_1$. This implies that $\Gamma_m;\Gamma_H \vdash \lambda^* F_1.\mathtt{LET}(H_1,e_1') : Ty(F_1) \to \mu_1$ where F_1 is the set of free variables of e_1' bound in Γ_m. By generalising Lemma 13, we get that $\Gamma_m;\Gamma_H \vdash \mathtt{LET}(H_1, \lambda^* F_1.e_1') : Ty(F_1) \to \mu_1$ 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 Γ_l; we now showed that $H \parallel H'$ is well formed with respect to $\Gamma_l[x \mapsto (\forall \overline{t}_i.\mu_1, F_1)]$. Applying the inductive hypothesis now to the translation of e_2 leads to the type preservation theorem.

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

□

A.2 Semantic Soundness

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

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

Type of a Value • $\Gamma \vdash i : int$

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

Notation E respects Γ

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

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

Implies that in a value environment E respecting Γ, e evaluates to v.

Henceforth in this section, we will assume that the value environment E respects the type environment Γ . 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 $\Gamma \vdash i : Int$ and $\Gamma \vdash i : Int$ and $i = i'$.
- $Clos\langle x^{\mu}, e, E\rangle \approx Clos\langle x^{\mu}, e', E'\rangle$ iff
  - $\Gamma \vdash \lambda x\!:\!\mu.e : \mu \to \mu'$ and $\Gamma \vdash \lambda x\!:\!\mu.e' : \mu \to \mu'$, and
  - $E[x \mapsto v_1] \vdash e \to v$ and $v_1 \approx v_1'$ implies that $E'[x \mapsto v_1'] \vdash e' \to v'$ and $v \approx v'$
- $Clos^{t}\langle \overline{t_i}, e, E\rangle \approx Clos^{t}\langle \overline{t_i}, e', E'\rangle$ iff
  - $\Gamma \vdash e : \forall\overline{t_i}.\mu$ and $\Gamma \vdash e' : \forall\overline{t_i}.\mu$, and
  - $E \vdash e[\mu_i/t_i] \to v$ implies that $E' \vdash e'[\mu_i/t_i] \to v'$, and $v \approx v'$.

Compatible Environments Two environments E and E' are compatible iff $dom(E) = dom(E')$ 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 $E_H = E_{h_1} \uplus \ldots \uplus E_{h_n}$. If $h_i ::= (y_i = e_i)$, and $E_m; E_{h_0 \ldots h_{i-1}} \vdash e_i \rightsquigarrow v_i$, then $E_{h_i} := y_i \mapsto v_i$.

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

Suppose $\Gamma_m; \Gamma_l; H \vdash e\!:\!\sigma \Rightarrow e'; H'; F'$ and $\Gamma_l(x) = (\forall\overline{t_i}.\sigma, F)$. Suppose further that E_m is compatible to E'_m. We say that H is well founded iff, $\forall x \in E_l, E_m; E_l \vdash x[\overline{\mu}_i] \rightsquigarrow v$ implies that $E'_m; E_H \vdash @^{*}x[\overline{\mu}_i]F \rightsquigarrow v'$ and $v \approx v'$.

In the above definition, note that the environments E_m, E'_m respect Γ_m; the environment E_H respects Γ_H and the environment E_l respects Γ_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.

(*const/var*)	$\overline{E \vdash i \to i} \qquad \overline{E \vdash x \to E(x)}$
(*fn*)	$\overline{E \vdash \lambda x:\mu.e \to Clos\langle x^{\mu}, e, E\rangle}$
(*app*)	$\dfrac{E \vdash x_1 \to Clos\langle x^{\mu}, e, E'\rangle \quad E \vdash x_2 \to v' \quad E' + x \mapsto v' \vdash e \to v}{E \vdash @x_1 x_2 \to v}$
(*tfn*)	$\overline{E \vdash \Lambda\overline{t_i}.e_v \mapsto Clos^t\langle\overline{t_i}, e_v, E\rangle}$
(*let*)	$\dfrac{E \vdash e_1 \to v_1 \qquad E + x \mapsto v_1 \vdash e_2 \to v}{E \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \to v}$
(*tapp*)	$\dfrac{E \vdash x \mapsto Clos^t\langle\overline{t_i}, e_v, E'\rangle \quad E' \vdash e_v[\mu_i/t_i] \to v}{E \vdash x[\overline{\mu_i}] \mapsto v}$

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 $E_m; E_l$ and for the target terms, E is substituted with $E'_m; E_H$.

Lemma 15

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

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 $(E_m; E_H)$.

□

Theorem 16 (Semantic Soundness)

Suppose $\Gamma_m; \Gamma_l; H \vdash e : \sigma \Rightarrow e'; H'; F'$. Suppose further that E_m and E'_m are compatible. If H is well founded and $E_m; E_l \vdash e \leadsto v$, then $E'_m; E_H \vdash \mathtt{LET}(H', e') \leadsto v'$ 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.

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

case tapp We need to prove that if $E_m; E_l \vdash x[\overline{\mu}_i] \leadsto v$, then $E'_m; E_H \vdash @^* x[\overline{\mu}_i]F \leadsto v'$ 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 $E_m; E_l \vdash \mathtt{let}\ x = \Lambda \bar{t}_i.e_1\ \mathtt{in}\ e_2 \leadsto v_2$, then $E'_m; E_H \vdash \mathtt{LET}(H' \parallel H_2, e'_2) \leadsto v'_2$ and $v_2 \approx v'_2$. We will first prove that the augmented environment $H \parallel H'$ is well founded during the translation of e_2. By definition, $E_{H'} ::= x \mapsto Clos^t\langle \bar{t}_i, \mathtt{LET}(H_1, \lambda^* F_1.e'_1), E'_m, E_H\rangle$.

Now for any compatible pairs of environments; E_{m_1} and E'_{m_1}, and environments E_{l_1}, E'_{H_1}, we need to prove that if $E_{m_1}; E_{l_1} \vdash x[\overline{\mu}_i] \leadsto v$ then $E'_{m_1}; E'_{H_1} \vdash @^* x[\overline{\mu}_i]F_1 \leadsto v'$ and $v \approx v'$. This implies we need to prove that if $E_m; E_l \vdash e_1[\mu_i/t_i] \leadsto v$ then $E'_{m_1}; E'_{H_1} \vdash @^* x[\overline{\mu}_i]F_1 \leadsto v'$ and $v \approx v'$. We will simplify the evaluation of the target term. $E'_{m_1}; E'_{H_1} \vdash x[\overline{\mu}_i]$ reduces to $E'_m; E_H \vdash \mathtt{LET}(H_1, \lambda^* F_1.e'_1)[\mu_i/t_i]$. This in turn reduces to $Clos\langle F_1^{Ty(F_1)}, e'_1[\mu_i/t_i], E'_m, E_H \uplus E_{H_1[\mu_i/t_i]}\rangle$. Therefore, the evaluation of the target term reduces to $E'_{m_1}; E'_{H_1} \vdash @^* Clos\langle F_1^{Ty(F_1)}, e'_1[\mu_i/t_i], E'_m, E_H \uplus E_{H_1[\mu_i/t_i]}\rangle F_1$. This in turn reduces to $E'_m \uplus E'_{m_1}(F_1); E_H \uplus E_{H_1[\mu_i/t_i]} \vdash e'_1[\mu_i/t_i]$. Since variables are bound only once, $E'_{m_1}(F_1) = E'_m(F_1)$. Therefore $E'_{m_1}(F_1) \uplus E'_m = E'_m$. And so the evaluation of the target term can be reduced finally to $E'_m; E_H \uplus E_{H_1[\mu_i/t_i]} \vdash e'_1[\mu_i/t_i] \leadsto v'$. But by induction on the translation of e_1, we get that $v \approx v'$. Hence $H \parallel H'$ is well founded with respect to $E_l[x \mapsto v]$. 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 $E_m; E_l \vdash \lambda x : \mu.e \leadsto v$, then $E'_m; E_H \vdash \mathtt{LET}(H', \lambda x : \mu.e') \leadsto v'$ and $v \approx v'$. By the inductive hypothesis, we know that if $E_m[x \mapsto v_1]; E_l \vdash e \leadsto v_2$, then $E'_m[x \mapsto v'_1]; E_H \vdash \mathtt{LET}(H', e') \leadsto v'_2$ 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 $\mathtt{LET}(H', e')$ as $\mathtt{let}\ y_1 = e_1\ \mathtt{in}\ e''$ where $e'' = \mathtt{LET}([h_2, \ldots, h_n], e')$. 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 $E'_m[x \mapsto v'_1]; E_H[y_1 \mapsto u_1] \vdash e'' \leadsto v'_2$. 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 $E'_m; E_H \vdash e_1 \leadsto u_1$ and $E'_m[x \mapsto v'_1]; E_H[y_1 \mapsto u_1] \vdash e'' \leadsto v'_2$. By continuing this process of splitting H', we can prove the required equivalence.

□

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 $F = \{x_1 : \sigma_1, ..., x_n : \sigma_n\}$, then $\lambda^* F.e$ denotes $\lambda x_1 : \sigma_1.(...(\lambda x_n : \sigma_n.e)..)$. Similarly $@^* zF$ denotes $@(..(@zx_1)..)x_n$.

Notation $Ty(L)$

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

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

(*int*)

$$\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash i : Int \Rightarrow i \ ; \ nil \ ; \ \emptyset$$

(*var*)

$$\frac{\Gamma_m(x) = T(\mu)}{\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash x : T(\mu) \Rightarrow x \ ; \ nil \ ; \ \{x : T(\mu)\}} \qquad \frac{\Gamma_p(x) = \sigma}{\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash x : \sigma \Rightarrow x \ ; \ nil \ ; \ \emptyset}$$

(*app*)

$$\frac{\Gamma_m(x_1) = T(\mu_1 \rightarrow \mu_2) \qquad \Gamma_m(x_2) = T(\mu_1)}{\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash @x_1 x_2 : T(\mu_2) \Rightarrow @x_1 x_2 \ ; \ nil \ ; \ \{x_1 : T(\mu_1 \rightarrow \mu_2), x_2 : T(\mu_1)\}}$$

(*app2*)

$$\frac{\Gamma_p(x_1) = \sigma_1 \rightarrow \sigma_2 \qquad \Gamma_p(x_2) = \sigma_1}{\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash @x_1 x_2 : \sigma_2 \Rightarrow @x_1 x_2 \ ; \ \emptyset \ ; \ \emptyset}$$

(*let*)

$$\frac{\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash e_1 : T(\mu_1) \Rightarrow e_1' \ ; \ H_1 \ ; \ F_1 \qquad \Gamma_m[x \mapsto T(\mu_1)]; \Gamma_p; \Gamma_l; H; d \vdash e_2 : \sigma_2 \Rightarrow e_2' \ ; \ H_2 \ ; \ F_2}{\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash \texttt{let}\ x = e_1 \ \texttt{in}\ e_2 : \sigma_2 \Rightarrow \texttt{let}\ x = e_1' \ \texttt{in}\ e_2' \ ; \ H_1 \parallel H_2 \ ; \ F_1 \cup F_2 \backslash \{x : T(\mu_1)\}}$$

(*let2*)

$$\frac{\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash e_1 : \sigma_1 \Rightarrow e_1' \ ; \ H_1 \ ; \ F_1 \qquad \Gamma_m; \Gamma_p[x \mapsto \sigma_1]; \Gamma_l; H; d \vdash e_2 : \sigma_2 \Rightarrow e_2' \ ; \ H_2 \ ; \ F_2}{\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash \texttt{let}\ x = e_1 \ \texttt{in}\ e_2 : \sigma_2 \Rightarrow \texttt{let}\ x = e_1' \ \texttt{in}\ \texttt{LET}(H_2, e_2') \ ; \ H_1 \ ; \ F_1 \cup F_2}$$

(*tapp*)

$$\frac{\Gamma_l(x) = (\forall \overline{t}_i.\sigma, L) \qquad k = i - j \qquad z \ \textit{new variable}}{\begin{array}{c} \Gamma_m; \Gamma_p; \Gamma_l; H; d = 0 \vdash x[\overline{\mu}_j] : (\forall \overline{t}_k.\sigma)[\mu_j / t_j] \Rightarrow \Lambda \overline{t'}_k.@^*(x[\overline{\mu}_j][\overline{t'}_k])L \ ; \ nil \ ; \ L \\ \Gamma_m; \Gamma_p; \Gamma_l; H; d > 0 \vdash x[\overline{\mu}_j] : (\forall \overline{t}_k.\sigma)[\mu_j / t_j] \Rightarrow \Lambda \overline{t'}_k.@^*(z[\overline{t'}_k])L \ ; \ [(z = x[\overline{\mu}_j])] \ ; \ L \end{array}}$$

(*tapp2*)

$$\frac{\Gamma_p(x) = \sigma \qquad z \ \textit{new variable}}{\begin{array}{c} \Gamma_m; \Gamma_p; \Gamma_l; H; d = 0 \vdash x[\overline{\mu}_i] : \sigma[\mu_i / t_i] \Rightarrow x[\overline{\mu}_i] \ ; \ nil \ ; \ \emptyset \\ \Gamma_m; \Gamma_p; \Gamma_l; H; d > 0 \vdash x[\overline{\mu}_i] : \sigma[\mu_i / t_i] \Rightarrow z \ ; \ [(z = x[\overline{\mu}_i])] \ ; \ \emptyset \end{array}}$$

(*tfn*)

$$\frac{\begin{array}{c} \Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash e_1 : \sigma_1 \Rightarrow e_1' \ ; \ H_1 \ ; \ F_1 \qquad H' = [(x = \Lambda \overline{t}_i.\texttt{LET}(H_1, \lambda^* F_1.e_1'))] \\ \Gamma_m; \Gamma_p; \Gamma_l[x \mapsto (\forall \overline{t}_i.\sigma_1, F_1)]; H \parallel H'; d \vdash e_2 : \sigma_2 \Rightarrow e_2' \ ; \ H_2 \ ; \ F_2 \end{array}}{\Gamma_m; \Gamma_p; \Gamma_l; H; d > 0 \vdash \texttt{let}\ x = \Lambda \overline{t}_i.e_1 \ \texttt{in}\ e_2 : \sigma_2 \Rightarrow e_2' \ ; \ H' \parallel H_2 \ ; \ F_2}$$

(*tfn2*)

$$\frac{\begin{array}{c} \Gamma_m; \Gamma_p; \Gamma_l; H; d = 0 \vdash e_1 : \sigma_1 \Rightarrow e_1' \ ; \ H_1 \ ; \ F_1 \qquad H' = [(x = \Lambda \overline{t}_i.e_1')] \\ \Gamma_m; \Gamma_p; \Gamma_l[x \mapsto (\forall \overline{t}_i.\sigma_1, nil)]; H \parallel H'; d = 0 \vdash e_2 : \sigma_2 \Rightarrow e_2' \ ; \ H_2 \ ; \ F_2 \end{array}}{\Gamma_m; \Gamma_p; \Gamma_l; H; d = 0 \vdash \texttt{let}\ x = \Lambda \overline{t}_i.e_1 \ \texttt{in}\ e_2 : \sigma_2 \Rightarrow \texttt{let}\ x = \Lambda \overline{t}_i.e_1' \ \texttt{in}\ e_2' \ ; \ H_1 \parallel H_2 \ ; \ F_1 \cup F_2}$$

(*fct*)

$$\frac{\Gamma_m; \Gamma_p[x \mapsto \sigma]; \Gamma_l; H; d = 0 \vdash e : \sigma' \Rightarrow e' \ ; \ H' \ ; \ F}{\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash \lambda^m x : \sigma.e : \sigma \rightarrow \sigma' \Rightarrow \lambda^m x : \sigma.e' \ ; \ H' \ ; \ F}$$

(*fn*)

$$\frac{\Gamma_m[x \mapsto T(\mu)]; \Gamma_p; \Gamma_l; H; d + 1 \vdash e : T(\mu') \Rightarrow e' \ ; \ H' \ ; \ F}{\begin{array}{c} \Gamma_m; \Gamma_p; \Gamma_l; H; d = 0 \vdash \lambda^c x : T(\mu).e : T(\mu \rightarrow \mu') \Rightarrow \texttt{LET}(H', \lambda^c x : T(\mu).e') \ ; \ nil \ ; \ F \backslash \{x : T(\mu)\} \\ \Gamma_m; \Gamma_p; \Gamma_l; H; d > 0 \vdash \lambda^c x : T(\mu).e : T(\mu \rightarrow \mu') \Rightarrow \lambda^c x : T(\mu).e' \ ; \ H' \ ; \ F \backslash \{x : T(\mu)\} \end{array}}$$

Figure 12: The Lifting Algorithm For FLINT

The variable environment is split up into 3 environments – Γ_m which binds the monomorphic variables, Γ_p which binds the variables introduced by functor abstractions, and Γ_l which binds the variables introduced by polymorphic let expressions.

Lemma 17

Suppose $\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash e : \sigma \Rightarrow e' \ ; \ H' \ ; \ F$. 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 *tapp* and *tfn*.

case tapp For $(d > 0)$, the only variables occuring free in e' are z and the variables in L. The variable z is newly introduced and hence cannot occur in Γ_m, while L is returned. For $(d = 0)$, the only variables occuring free in e' are x and the variables in L. The variable x occurs in Γ_l, while L is returned.

case tapp2 For $(d > 0)$, the only variable occuring free in e' is z which is a newly introduced variable; hence does not occur in Γ_m. For (d = 0), x is the only free variable but x occurs in Γ_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 $\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash e : \sigma \Rightarrow e' \ ; \ H' \ ; \ F$. Then if $d = 0$, the header $H' = nil$.

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

□

Lemma 19

Suppose $\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash e : \sigma \Rightarrow e' \ ; \ H' \ ; \ F$. 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 donot 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 Γ_l.

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

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

□

Lemma 20

The variables bound in Γ_m, Γ_p and Γ_l are mutually disjoint; and the variables bound in Γ_m, Γ_p and Γ_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.

□

B.1 Type Preservation

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

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

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

$$(\textit{fct}) \qquad \frac{\Gamma \uplus \{x : \sigma_1\} \vdash e : \sigma_2}{\Gamma \vdash \lambda^m x : \sigma_1.e : \sigma_1 \to \sigma_2}$$

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

$$(\textit{app2}) \qquad \frac{\Gamma \vdash x_1 : T(\mu_1 \to \mu_2) \quad \Gamma \vdash x_2 : T(\mu_1)}{\Gamma \vdash @x_1 x_2 : T(\mu_2)}$$

$$(\textit{tfn}) \qquad \frac{\Gamma \vdash e_v : \sigma_1 \quad \Gamma \uplus \{x : \forall \overline{t_i}.\sigma_1\} \vdash e : \sigma_2 \qquad t_i \textit{ not free in } \Gamma}{\Gamma \vdash \texttt{let } x = \Lambda \overline{t_i}.e_v \texttt{ in } e : \sigma_2}$$

$$(\textit{tapp}) \qquad \frac{\Gamma \vdash x : \forall \overline{t_i}.\sigma \qquad k = i - j}{\Gamma \vdash x[\overline{\mu}_j] : \forall \overline{t}_k.[\mu_j / t_j]\sigma}$$

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

Figure 13: Static Semantics

Definition 6 $(\texttt{LET}(H, e))$

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

Definition 7 (Γ_H Header Type Environment)

Suppose $\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash e : \sigma \Rightarrow e' \ ; \ H' \ ; \ F$. 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 $\Gamma_m; \Gamma_p; \Gamma_{h_1 \ldots h_{i-1}} \vdash e_i : \sigma_i$; then $\Gamma_{h_i} := y_i \mapsto \sigma_i$.

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

H is well formed iff $\Gamma_l(x) = (\forall \bar{t}_i.\sigma, F)$ implies that $\Gamma_H(x) = \forall \bar{t}_i.Ty(F) \rightarrow \sigma$.

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 $\Gamma_m; \Gamma_p; \Gamma_l$, and for the target terms, Γ is substituted with $\Gamma_m; \Gamma_p; \Gamma_H$.

Lemma 21

Suppose $\Gamma_m; \Gamma_p; \Gamma_H \vdash \texttt{let}\ x = \texttt{LET}(H_1, e')\ \texttt{in}\ \texttt{LET}(H_2, e'') : \sigma$ and x does not occur free in either H_1 or H_2. Then $\Gamma_m; \Gamma_p; \Gamma_H \vdash \texttt{LET}(H_1 \parallel H_2, \texttt{let}\ x = e'\ \texttt{in}\ e'') : \sigma$

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

The header $H_2 ::= [h_1, \ldots, h_n]$ where each $h_i ::= y_i = e_i$. Therefore write $\texttt{LET}(H_2, e'')$ as $\texttt{let}\ y_1 = e_1\ \texttt{in}\ e$ where $e = \texttt{LET}([h_2, \ldots, h_n], e'')$. When we consider the typing rules for this expression, we get that $\Gamma_m[x \mapsto T(\mu)]; \Gamma_p; \Gamma_H \uplus \Gamma_{H_1} \vdash e_1 : \sigma_1$ and $\Gamma_m[x \mapsto T(\mu)]; \Gamma_p; \Gamma_H \uplus \Gamma_{H_1} \uplus [y_1 \mapsto \sigma_1] \vdash e : \sigma$ for some type σ_1. 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 – $\Gamma_m; \Gamma_p; \Gamma_H \uplus \Gamma_{H_1} \vdash e_1 : \sigma_1$ and $\Gamma_m; \Gamma_p; \Gamma_H \uplus \Gamma_{H_1} \uplus [y_1 \mapsto \sigma_1] \vdash e' : T(\mu)$ and $\Gamma_m[x \mapsto T(\mu)]; \Gamma_p; \Gamma_H \uplus \Gamma_{H_1} \uplus [y_1 \mapsto \sigma_1] \vdash e : \sigma$. This implies that $\Gamma_m; \Gamma_p; \Gamma_H \uplus \Gamma_{H_1} \vdash \texttt{let}\ y_1 = e_1\ \texttt{in}\ \texttt{let}\ x = e'\ \texttt{in}\ e : \sigma$. By continuing this process of breaking H_2 into its components, in the end, we get the following typing equations – $\Gamma_m; \Gamma_p; \Gamma_H \uplus \Gamma_{H_1} \uplus \Gamma_{H_2} \vdash e' : T(\mu)$ and $\Gamma_m[x \mapsto T(\mu)]; \Gamma_p; \Gamma_H \uplus \Gamma_{H_1} \uplus \Gamma_{H_2} \vdash e'' : \sigma$ which proves the lemma.

□

Lemma 22

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

Proof The header $H' ::= [h_1, \ldots, h_n]$ with each $h_i ::= y_i = e_i$. Therefore write $\texttt{LET}(H', e)$ as $\texttt{let}\ y_1 = e_1\ \texttt{in}\ e'$ where $e' ::= \texttt{LET}([h_2, \ldots, h_n], e)$. The typing rule for the expression $\lambda x : T(\mu).\texttt{LET}(H', e)$ can be written as $\Gamma_m[x \mapsto T(\mu)]; \Gamma_p; \Gamma_H \vdash e_1 : \sigma_1$ and $\Gamma_m[x \mapsto T(\mu)]; \Gamma_p; \Gamma_H[y_1 \mapsto \sigma_1] \vdash e' : \sigma'$. 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 $\Gamma_m; \Gamma_p; \Gamma_H \vdash e_1 : \sigma_1$ and $\Gamma_m[x \mapsto T(\mu)]; \Gamma_p; \Gamma_H[y_1 \mapsto \sigma_1] \vdash e' : \sigma'$. This implies that $\Gamma_m; \Gamma_p; \Gamma_H \vdash \texttt{let}\ y_1 = e_1\ \texttt{in}\ \lambda x : T(\mu).e' : T(\mu) \rightarrow \sigma'$. Continuing to break H' like this, we finally reach $\Gamma_m; \Gamma_p; \Gamma_H \vdash \texttt{LET}(H', \lambda x : T(\mu).e) : T(\mu) \rightarrow \sigma'$.

□

Theorem 23 (Type Preservation)

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

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

case int Both the source and target terms have type `Int`.

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

case var x occurs in Γ_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 $\Gamma_m; \Gamma_p; \Gamma_H \vdash x_2 : T(\mu_1)$. This implies that $\Gamma_m; \Gamma_p; \Gamma_H \vdash @x_1 x_2 : T(\mu_2)$.

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 $\Gamma_m; \Gamma_p; \Gamma_H \vdash @x_1 x_2 : \sigma_2$.

case let By the inductive hypothesis, $\Gamma_m; \Gamma_p; \Gamma_H \vdash \mathtt{LET}(H_1, e'_1) : T(\mu_1)$ and $\Gamma_m[x \mapsto T(\mu_1)]; \Gamma_p; \Gamma_H \vdash \mathtt{LET}(H_2, e'_2) : \sigma_2$. Since H_2 and H_1 are closed with respect to variables in Γ_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, $\Gamma_m; \Gamma_p; \Gamma_H \vdash \mathtt{LET}(H_1, e'_1) : \sigma_1$ and $\Gamma_m; \Gamma_p[x \mapsto \sigma_1]; \Gamma_H \vdash \mathtt{LET}(H_2, e'_2) : \sigma_2$. Since the variables bound in H_1 do not occur in $\mathtt{LET}(H_2, e'_2)$, we can introduce the variables in H_1 in the environment while typing $\mathtt{LET}(H_2, e'_2)$. Therefore we have that $\Gamma_m; \Gamma_p; \Gamma_H \uplus \Gamma_{H_1} \vdash e'_1 : \sigma_1$ and $\Gamma_m; \Gamma_p[x \mapsto \sigma_1]; \Gamma_H \uplus \Gamma_{H_1} \vdash \mathtt{LET}(H_2, e'_2) : \sigma_2$. The type preservation theorem follows from this.

case tapp From the antecedent we know that $\Gamma_l(x) = (\forall \overline{t}_i.\sigma, L)$. By the well formedness of H, we know that $\Gamma_H(x) = \forall \overline{t}_i.Ty(L) \rightarrow \sigma$. Therefore $\Gamma_m; \Gamma_p; \Gamma_H \vdash x[\overline{\mu}_j] : \forall \overline{t}_k.Ty(L) \rightarrow \sigma[\mu_j/t_j]$. Since L consists of the free variables of x, the type $Ty(L)$ cannot contain any of the t_i as a free type variable.

For the (d = 0) case, therefore $\Gamma_m; \Gamma_p; \Gamma_H \vdash x[\overline{\mu}_j][\overline{t'}_k] : Ty(L) \rightarrow \sigma[\mu_j/t_j][t'_k/t_k]$. Therefore the translated term has type $\forall \overline{t'}_k.\sigma[\mu_j/t_j]$ which is alpha equivalent to the source term.

For the $(d > 0)$ case, $\Gamma_m; \Gamma_p; \Gamma_H \vdash z : \forall \overline{t}_k.Ty(L) \rightarrow \sigma[\mu_j/t_j]$. This leads to $\Gamma_m; \Gamma_p; \Gamma_H \vdash z[\overline{t'}_k] : Ty(L) \rightarrow \sigma[\mu_j/t_j][t'_k/t_k]$ 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 in both the source and target terms, x has type σ. 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 $\Gamma_m; \Gamma_p; \Gamma_H \vdash \mathtt{LET}(H' \parallel H_2, e'_2) : \sigma_2$ which implies that $\Gamma_m; \Gamma_p; \Gamma_H \uplus \Gamma_{H'} \vdash \mathtt{LET}(H_2, e'_2) : \sigma_2$ must be true. By definition, $\Gamma_{H'} ::= x \mapsto \sigma'$ where $\Gamma_m; \Gamma_p; \Gamma_H \vdash \Lambda \overline{t}_i.\mathtt{LET}(H_1, \lambda^* F_1.e'_1) : \sigma'$. By the inductive hypothesis, we know that $\Gamma_m; \Gamma_p; \Gamma_H \vdash \mathtt{LET}(H_1, e'_1) : \sigma_1$. This implies that $\Gamma_m; \Gamma_p; \Gamma_H \vdash \lambda^* F_1.\mathtt{LET}(H_1, e'_1) : Ty(F_1) \rightarrow \sigma_1$ where F_1 is the set of free variables of e'_1 bound in Γ_m. By generalising Lemma 22, we get that $\Gamma_m; \Gamma_p; \Gamma_H \vdash \mathtt{LET}(H_1, \lambda^* F_1.e'_1) : Ty(F_1) \rightarrow \sigma_1$ 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 Γ_l; we now showed that $H \parallel H'$ is well formed with respect to $\Gamma_l[x \mapsto (\forall \overline{t}_i.\sigma_1, F_1)]$. Applying the inductive hypothesis now to the translation of e_2 leads to the type preservation theorem.

case tfn2 We know that $H_1 = H_2 = nil$(Lemma 18). Therefore we need to prove that $\Gamma_m; \Gamma_p; \Gamma_H \vdash \texttt{let}\ x = \Lambda\overline{t_i}.e_1'\ \texttt{in}\ e_2' : \sigma_2$. By induction on the translation of e_1, $(H_1 = nil)$, 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 $\Gamma_l[x \mapsto (\forall\overline{t_i}.\sigma_1, nil)]$. Applying the inductive hypothesis now to the translation of e_2 leads to the type preservation theorem.

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

□

B.2 Semantic Soundness

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

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

Type of a Value • $\Gamma \vdash i : int$

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

Notation E respects Γ

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

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

Implies that in a value environment E respecting Γ, e evaluates to v.

Henceforth in this section, we will assume that the value environment E respects the type environment Γ . 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 $\Gamma \vdash i : Int$ and $\Gamma \vdash i : Int$ and $i = i'$.
- $Clos\langle x^\sigma, e, E\rangle \approx Clos\langle x^\sigma, e', E'\rangle$ iff
  - $\Gamma \vdash \lambda x : \sigma.e : \sigma \to \sigma'$ and $\Gamma \vdash \lambda x : \sigma.e' : \sigma \to \sigma'$, and
  - $E[x \mapsto v_1] \vdash e \to v$ and $v_1 \approx v_1'$ implies that $E'[x \mapsto v_1'] \vdash e' \to v'$ and $v \approx v'$
- $Clos^t\langle\overline{t_i}, e, E\rangle \approx Clos^t\langle\overline{t_i}, e', E'\rangle$ iff
  - $\Gamma \vdash e : \forall\overline{t_i}.\sigma$ and $\Gamma \vdash e' : \forall\overline{t_i}.\sigma$, and
  - $E \vdash e[\mu_i/t_i] \to v$ implies that $E' \vdash e'[\mu_i/t_i] \to v'$, and $v \approx v'$.

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

$$(const/var) \qquad \frac{}{E \vdash i \to i} \qquad \frac{}{E \vdash x \to E(x)}$$

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

$$(fn) \qquad \frac{}{E \vdash \lambda^m x : \sigma.e \to Clos\langle x^{\sigma}, e, E\rangle}$$

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

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

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

$$(tapp) \qquad \frac{E \vdash x \mapsto Clos^t\langle \overline{t_i}, e_v, E'\rangle \qquad E' \vdash e_v[\mu_j/t_j] \to v}{E \vdash x[\overline{\mu}_j] \mapsto v}$$

Figure 14: Operational Semantics

Definition 9 (E_H Header Value Environment)

Suppose $\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash e : \sigma \Rightarrow e' \ ; \ H' \ ; \ F$. If $H ::= [h_1, \ldots, h_n]$, then $E_H = E_{h_1} \uplus \ldots \uplus E_{h_n}$. If $h_i ::= (y_i = e_i)$, and $E_m; E_p; E_{h_0 \ldots h_{i-1}} \vdash e_i \rightsquigarrow v_i$, then $E_{h_i} := y_i \mapsto v_i$.

Definition 10 (H is well founded)

Suppose $\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash e : \sigma \Rightarrow e' \ ; \ H' \ ; \ F'$ and $\Gamma_l(x) = (\forall \overline{t}_i.\sigma, F)$. 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 iff, $\forall x \in \Gamma_l$, $E_m; E_p; E_l \vdash x[\overline{\mu}_i] \rightsquigarrow v$ implies that $E'_m; E'_p; E_H \vdash @^* x[\overline{\mu}_i]F \rightsquigarrow v'$ and $v \approx v'$.

In the above definition, note that the environments E_m, E'_m respect Γ_m; the environments E_p, E'_p respect Γ_p; the environment E_H respects Γ_H and the environment E_l respects Γ_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.

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 $E_m; E_p; E_l$ and for the target terms, E is substituted with $E'_m; E'_p; E_H$.

Lemma 24

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

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 $(E_m; E_p; E_H)$.

□

Theorem 25 (Semantic Soundness)

Suppose $\Gamma_m; \Gamma_p; \Gamma_l; H; d \vdash e : \sigma \Rightarrow e' \ ; \ H' \ ; \ F'$. Suppose further that E_m, E'_m are compatible and E_p, E'_p are compatible. If H is well founded and $E_m; E_p; E_l \vdash e \rightsquigarrow v$, then $E'_m; E'_p; E_H \vdash \mathtt{LET}(H', e') \rightsquigarrow v'$ 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 $E_m; E_p; E_l \vdash e_1 \rightsquigarrow v_1$, then $E'_m; E'_p; E_H \vdash \mathtt{LET}(H_1, e_1) \rightsquigarrow v'_1$ and $v_1 \approx v'_1$. Similarly, by induction on e_2, if $E_m[x \mapsto v_1]; E_p; E_l \vdash e_2 \rightsquigarrow v_2$, then $E'_m[x \mapsto v'_1]; E'_p; E_H \vdash \mathtt{LET}(H_2, e'_2) \rightsquigarrow v'_2$ and $v_2 \approx v'_2$ This implies that $E'_m; E'_p; E_H \vdash \mathtt{let}\ x = \mathtt{LET}(H_1, e'_1)\ \mathtt{in}\ \mathtt{LET}(H_2, e'_2) \rightsquigarrow v'_2$. By Lemma 19, x does not occur free in either H_1 or H_2. But by Lemma 24, this implies that $E'_m; E'_p; E_H \vdash \mathtt{LET}(H_1 \parallel H_2, \mathtt{let}\ x = e'_1\ \mathtt{in}\ e'_2) \rightsquigarrow v'_2$. The required equivalence follows from this.

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

case tapp Consider first the case of (d = 0). We need to prove that if $E_m; E_p; E_l \vdash x[\overline{\mu}_j] \rightsquigarrow v$, then $E'_m; E'_p; E_H \vdash \Lambda\overline{t'}_k.@^* x[\overline{\mu}_j][\overline{t'}_k]L \rightsquigarrow v'$ and that $v \approx v'$. From the type of $x[\overline{\mu}_j]$, 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 $E_m; E_p; E_l \vdash v[\overline{\mu}_k] \rightsquigarrow v_1$, then $E'_m; E'_p; E_H \vdash v'[\overline{\mu}_k] \rightsquigarrow v'_1$, and $v_1 \approx v'_1$. This implies we must prove that if $E_m; E_p; E_l \vdash x[\overline{\mu}_j][\overline{\mu}_k] \rightsquigarrow v_1$, then $E'_m; E'_p; E_H \vdash @^* x[\overline{\mu}_j][\overline{\mu}_k]L \rightsquigarrow v'_1$, 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 $E_m; E_p; E_l \vdash x[\overline{\mu}_j] \rightsquigarrow v$, then $E'_m; E'_p; E_H \vdash \mathtt{LET}(z = x[\overline{\mu}_j], \Lambda\overline{t'}_k.@^* z[\overline{t'}_k]L) \rightsquigarrow v'$ 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 $E'_m; E'_p; E_H \vdash \Lambda\overline{t'}_k.@^* x[\overline{\mu}_j][\overline{t'}_k]L \rightsquigarrow v'$ and $v \approx v'$. This can be proved in an exactly similar way to the proof of soundness for (d = 0).

case tapp2 In the source term, x evaluates to $E_p(x) = v$(say). In the target term, x evaluates to $E'_p(x) = v'$(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 $E_m; E_p; E_l \vdash \texttt{let}\ x = \Lambda\bar{t}_i.e_1\ \texttt{in}\ e_2 \leadsto v_2$, then $E'_m; E'_p; E_H \vdash \texttt{LET}(H' \parallel H_2, e'_2) \leadsto v'_2$ and $v_2 \approx v'_2$. We will first prove that the augmented environment $H \parallel H'$ is well founded during the translation of e_2. By definition, $E_{H'} ::= x \mapsto Clos^t\langle\bar{t}_i, \texttt{LET}(H_1, \lambda^* F_1.e'_1), E'_m, E'_p, E_H\rangle$

Now for any compatible pairs of environments; E_{m_1} and E'_{m_1}, E_{p_1} and E'_{p_1} and environments E_{l_1}, E'_{H_1}, we need to prove that if $E_{m_1}; E_{p_1}; E_{l_1} \vdash x[\overline{\mu}_i] \leadsto v$ then $E'_{m_1}; E'_{p_1}; E_{H_1} \vdash @^* x[\overline{\mu}_i]F_1 \leadsto v'$ 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 $E'_{m_1}; E'_{p_1}; E_{H_1} \vdash @^* x[\overline{\mu}_i]F_1 \leadsto v'$ and $v \approx v'$. We will simplify the evaluation of the target term. $E'_{m_1}; E'_{p_1}; E'_{H_1} \vdash x[\overline{\mu}_i]$ reduces to $E'_m; E'_p; E_H \vdash \texttt{LET}(H_1, \lambda^* F_1.e'_1)[\mu_i/t_i]$. This in turn reduces to $Clos\langle F_1^{Ty(F_1)}, e'_1[\mu_i/t_i], E'_m, E'_p, E_H \uplus E_{H_1[\mu_i/t_i]}\rangle$. Therefore, the evaluation of the target term reduces to $E'_{m_1}; E'_{p_1}; E'_{H_1} \vdash @^* Clos\langle F_1^{Ty(F_1)}, e'_1[\mu_i/t_i], E'_m, E'_p, E_H \uplus E_{H_1[\mu_i/t_i]}\rangle F_1$. This in turn reduces to $E'_m \uplus E'_{m_1}(F_1); E'_p; E_H \uplus E_{H_1[\mu_i/t_i]} \vdash e'_1[\mu_i/t_i]$. Since variables are bound only once, $E'_{m_1}(F_1) = E'_m(F_1)$. Therefore $E'_{m_1}(F_1) \uplus E'_m = E'_m$. And so the evaluation of the target term can be reduced finally to $E'_m; E'_p; E_H \uplus E_{H_1[\mu_i/t_i]} \vdash e'_1[\mu_i/t_i] \leadsto v'$. But by induction on the translation of e_1, we get that $v \approx v'$. Hence $H \parallel H'$ is well founded with respect to $E_l[x \mapsto v]$. And so we can apply the inductive hypothesis to the translation of e_2 which leads directly to the semantic soundness result.

case tfn2 We know by Lemma 18 that $H_1 = H_2 = nil$. Therefore we need to prove that if $E_m; E_p; E_l \vdash \texttt{let}\ x = \Lambda\bar{t}_i.e_1\ \texttt{in}\ e_2 \leadsto v$ then $E'_m; E'_p; E_H \vdash \texttt{let}\ x = \Lambda\bar{t}_i.e'_1\ \texttt{in}\ e'_2 \leadsto v'$ and $v \approx v'$. We will first prove that the header $H \parallel H'$ is well founded. By definition, $E_{H'} ::= x \mapsto Clos^t\langle\bar{t}_i, e'_1, E'_m, E'_p, E_H\rangle$. Now for any compatible pairs of environments; E_{m_1} and E'_{m_1}, E_{p_1} and E'_{p_1} and environments E_{l_1}, E'_{H_1}, we need to prove that if $E_{m_1}; E_{p_1}; E_{l_1} \vdash x[\overline{\mu}_i] \leadsto v$ then $E'_{m_1}; E'_{p_1}; E_{H_1} \vdash x[\overline{\mu}_i] \leadsto v'$ 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 $E'_m; E'_p; E_H \vdash e'_1[\mu_i/t_i] \leadsto v'$ and $v \approx v'$. 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 $E_m; E_p[x \mapsto v_2]; E_l \vdash e \leadsto v$, then $E'_m; E'_p[x \mapsto v'_2]; E_H \vdash \texttt{LET}(H', e') \leadsto v'$ 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 $E'_m; E'_p; E_H \vdash \lambda^m x : \sigma.e' \leadsto v'_1$ 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 $E'_m; E'_p; E_H \vdash \texttt{LET}(H', \lambda^c x : T(\mu).e') \leadsto v'$ and $v \approx v'$. By the inductive hypothesis, we know that if $E_m[x \mapsto v_1]; E_p; E_l \vdash e \leadsto v_2$, then $E'_m[x \mapsto v'_1]; E'_p; E_H \vdash \texttt{LET}(H', e') \leadsto v'_2$ 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 $h_i ::= y_i = e_i$. Consider the expression $\texttt{LET}(H', e')$ as $\texttt{let}\ y_1 = e_1\ \texttt{in}\ e''$ where $e'' = \texttt{LET}([h_2, \ldots, h_n], e')$. From the evaluation rules we can deduce that $E'_m[x \mapsto v'_1]; E'_p; E_H \vdash e_1 \leadsto u_1$ (for some u_1) and $E'_m[x \mapsto v'_1]; E'_p; E_H[y_1 \mapsto u_1] \vdash e'' \leadsto v'_2$. 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 $E'_m; E'_p; E_H \vdash e_1 \leadsto u_1$ and $E'_m[x \mapsto v'_1]; E'_p; E_H[y_1 \mapsto u_1] \vdash e'' \leadsto v'_2$. By continuing this process of splitting H', we can prove the required equivalence.

□