ML-Style Typing, Lambda Lifting, and Partial Evaluation

Peter Thiemann*
thiemann@informatik.uni-freiburg.de

Institut für Informatik
Universität Freiburg, Germany

Abstract. Lambda lifting transforms local function definitions in functional programming languages into global ones to facilitate their efficient implementation. When considered as a source-to-source transformation for an ML-style typed language, all published algorithms for lambda lifting break typability.
We break down two representative lambda lifting algorithms into four elementary transformation steps, analyze their typing properties, and define alternative steps that preserve ML typability where appropriate. Putting the revised steps together, we define a lambda lifter that preserves ML-style typability. Our algorithm provides a starting point for implementing efficient generating extensions that perform offline partial evaluation for polymorphically typed languages.

1 Introduction

Lambda lifting is one of the standard transformations in the tool box of the implementor of functional programming languages. Its task is to close local function definitions by parameterizing them over their free variables. Once all functions have been closed in this way (turned into combinators), they are moved to the top-level where the compiler generates efficient code for them [9].

Lambda lifting is also a key transformation in program specializers, because it increases code sharing of specialized functions [4]. The intention is that the specializer runs at most once for every pair of source function and specialization-time arguments. A log keeps track of the started specializations and — before starting a new specialization — the specializer checks the log whether the work has been done already. Local function definitions require a more elaborate log structure which does not guarantee sharing of their specializations [13].

It is also advantageous to perform specialization in a polymorphic language: if a function is polymorphic in a first-order specialization-time argument then its specialization does not depend on the particular value of the argument. This is an immediate consequence of Wadler's free theorems [28] which are based on

* This work has been done while at the School of Computer Science and Information Technology, University of Nottingham, UK. The author acknowledges support by EPSRC grant GR/M22840 "Semantics of Specialization".

Reynolds' work on parametricity [22]. Therefore, it is wasteful to specialize with respect to different values of first-order polymorphic arguments. Of course, it might still make sense to specialize with respect to a polymorphic function.

Our approach to an efficient implementation of partial evaluation is based on the cogen-by-hand approach [11]. In this approach, we want to transform—for the above reasons—the lambda-lifted polymorphic source program into its own *generating extension* (a specializer dedicated to specialize that particular source program). Ideally, this generating extension is implemented in the same language as the source program to preserve the semantics of the specialization-time computation. Since the generating extension must obey the same typing discipline as the source program, we have to worry whether lambda lifting preserves typability. It turns out that none of the published lambda lifters does. This does not come as a complete surprise because closure conversion [21] — a close relative to lambda lifting — gives rise to typing problems, too [15, 27, 26, 2].

Essentially, all presentations of various aspects of lambda lifting deal with untyped languages [10, 17, 19, 20, 6]. This is fine, because they are solving an important problem in an untyped intermediate language after type checking has been performed. In addition, most of them are informal, which makes it hard to analyze the properties of the transformations and to compare them.

To enable the analysis of the typing properties, we have subdivided lambda lifting in four elementary transformations. Two of them can give rise to typing problems: one transformation step changes a let-bound variable into a lambda-bound variable and in another transformation step a problem arises due to ML's restriction to monomorphic recursion. We analyze two representative algorithms in this respect, Johnsson's [10] which suffers from both problems and the one in Peyton Jones's book [17, Ch.13] which avoids the problem with monomorphic recursion, but is not directly applicable to strict languages. The main difference between the two algorithm lies in the order in which they apply the elementary transformations and which restrictions they impose. The basic idea of Peyton Jones's algorithm is close enough to the remaining algorithms [9, 19, 20] that we do not have to consider them separately.

In the following sections, we formalize the elementary transformation steps of lambda lifting, we examine their typing properties in an ML-style type system, and we outline a refined lambda lifter that overcomes the typing problems. Our algorithm is applicable to both strict and non-strict languages. It does not duplicate computations, so it neither affects the complexity nor the laziness properties of the program.

No lambda lifter can preserve typability in the presence of polymorphic recursion, but polymorphic recursion and first-class polymorphism together is enough to ensure that every lambda lifter can preserve typability.

$$\textit{(var)}\ \frac{x : \forall\alpha_1 \ldots \alpha_k.\tau \text{ in } A}{A \vdash x : \tau[\overline{\alpha_k \mapsto \tau_k}]}$$

$$\textit{(abs)}\ \frac{A\{x : \tau_2\} \vdash e : \tau_1}{A \vdash \lambda x.e : \tau_2 \to \tau_1}$$

$$\textit{(app)}\ \frac{A \vdash e_1 : \tau_2 \to \tau_1 \quad A \vdash e_2 : \tau_2}{A \vdash e_1\ e_2 : \tau_1}$$

$$\textit{(letrec)}\ \frac{A\{\overline{x_n : \tau_n}\} \vdash e_i : \tau_i \quad A\{\overline{x_n : GEN(A, \tau_n)}\} \vdash e : \tau}{A \vdash \mathtt{let}\ \{\overline{x_n = e_n}\}\ \mathtt{in}\ e : \tau}$$

Fig. 1. ML-style typing rules

2 Lambda-ML

Lambda-ML is a lambda calculus with variables, lambda abstractions, applications, and let-expressions which install a *set* of mutually recursive bindings.

$$e ::= x \mid \lambda x.e \mid e\ e \mid \mathtt{let}\ \{\overline{x_n = e_n}\}\ \mathtt{in}\ e$$

The notation $\overline{x_n = e_n}$ abbreviates $x_1 = e_1, \ldots, x_n = e_n$. The set of free variables of an expression $FV(e)$ is defined as usual with

$$FV(\mathtt{let}\ \{\overline{x_n = e_n}\}\ \mathtt{in}\ e) = (FV(e) \cup \bigcup FV(e_i)) \setminus \{x_1, \ldots, x_n\}$$

We write $e_1 \equiv e_2$ for textual equivalence of expressions. We always consider expressions modulo α-conversion so that transformations do not accidentally capture variables. The notation $e[x \mapsto e']$ denotes the capture-avoiding substitution of e' for the free occurrences of x in e. A context C is an expression with a hole $[\ ]$. The hole can be filled with an expression e as in $C[e]$. This operation captures variables.

The type language features type variables and function types; type schemes are defined as usual in ML.

$$\begin{array}{ll} \tau ::= \alpha \mid \tau \to \tau & \text{types} \\ \sigma ::= \tau \mid \forall\alpha.\sigma & \text{type schemes} \end{array}$$

A type assumption A is a finite list of pairs $x : \sigma$ where each variable occurs at most once in A. The set $FTV(\sigma)$ of free type variables of σ is defined as usual, $FTV(A)$ is the union of $FTV(\sigma)$ for all $x : \sigma$ in A. $GEN(A, \tau)$ is defined to be $\forall\alpha_1 \ldots \alpha_k.\tau$ if $\{\alpha_1, \ldots, \alpha_k\} = FTV(\tau) \setminus FTV(A)$. Figure 1 recalls the ML-style typing rules for the judgement $A \vdash e : \tau$, i.e., e has type τ under assumption A.

We leave the semantics of the language unspecified because we want to consider both call-by-value and call-by-name variants of it.

The examples have been type checked using the Objective Caml system [12]. Caml uses `fun` x `->` e for $\lambda x.e$ and `let rec` $x_1 = e_1$ `and` $\ldots$ `and` $x_n = e_n$ `in` e instead of `let` $\{\overline{x_n = e_n}\}$ `in` e.

3 Transformation steps

Lambda lifting comprises four elementary transformation steps, DEFINE, EXPAND, SPLIT and LIFT. This section defines them and analyzes their typing properties. Each algorithm places additional constraints on the applicability of the transformation rules to preserve operational properties.

The translation is given as a relation $e_s \leadsto e_t$ between expressions. If the definition of a rule does not explicitly state a context then the rule is applicable in all possible contexts. We sometimes restrict the relation to the application of a certain rule or rule set by indexing it as in $e_s \overset{(1)}{\leadsto} e_t$.

3.1 DEFINE

The DEFINE transformation introduces names for anonymous lambda expressions. To ensure the termination of the exhaustive application of the DEFINE transformation and to avoid introducing useless names, a lambda lifter restricts the applicability to contexts so that

- it only names the outermost lambda of an expression $\lambda x_1. \ldots \lambda x_n.e$;
- it does not name a lambda that is already named.

The context N captures this restriction.

$$\begin{aligned} N &::= [\,] \mid B \\ B &::= \lambda x.B \mid N\ e \mid e\ N \mid \texttt{let}\ \{\ldots x = B \ldots\}\ \texttt{in}\ e \mid \texttt{let}\ \{\ldots\}\ \texttt{in}\ N \end{aligned}$$

The transformation rule below does not affect typability.

$$N[\lambda x.e] \leadsto N[\texttt{let}\ \{f = \lambda x.e\}\ \texttt{in}\ f] \tag{1}$$

Proposition 1. *Suppose $A \vdash e_s : \tau$ and $e_s \overset{(1)}{\leadsto} e_t$ then $A \vdash e_t : \tau$.*

3.2 EXPAND

The transformation EXPAND considers an arbitrary binding $x_0 = e_0$ in a let-expression. If e_0 has a free variable $y \not\equiv x_0$ EXPAND abstracts y from e_0 and replaces all occurrences of x_0 in the scope of the let-binding by $x_0\ y$.

$$\frac{y \in FV(e) \setminus x_0 \quad e' \equiv e[x_0 \mapsto x_0\ y] \quad e'_i \equiv e_i[x_0 \mapsto x_0\ y]}{\begin{array}{c}\texttt{let}\ \{x_0 = e_0, \overline{x_n = e_n}\}\ \texttt{in}\ e \\ \leadsto \\ \texttt{let}\ \{x_0 = \lambda y.e'_0, \overline{x_n = e'_n}\}\ \texttt{in}\ e'\end{array}} \tag{2}$$

Typing properties Let's analyze the typing properties of $e_s \overset{(2)}{\leadsto} e_t$. Let A be a type assumption with $y : \sigma$ in A and $A \vdash e_s : \tau$. Without lack of generality, we assume $e_s \equiv \texttt{let}\ \{x_0 = e_0, \overline{x_n = e_n}\}\ \texttt{in}\ e$ so that the last step of the type derivation for e_s must be an application of the *(letrec)* rule. We try to prove $A \vdash e_t : \tau$. There are two cases (using the notation from the rule).

1. Suppose $\sigma = \tau_y$. From the assumption of *(letrec)*,
$$A\{x_0 : \tau_0, \overline{x_n : \tau_n}\} \vdash e_i : \tau_i$$
for $0 \le i \le n$ and that
$$A\{x_0 : GEN(A, \tau_0), \overline{x_n : GEN(A, \tau_n)}\} \vdash e : \tau.$$
First, we have to show that $A\{x_0 : \tau_y \to \tau_0, \overline{x_n : \tau_n}\} \vdash e_i' : \tau_i$ for $0 \le i \le n$. The proof is by induction on the type derivation for e_i, the only interesting step being the variable case for x_0 where in e_s we have $A\{x_0 : \tau_0, \overline{x_n : \tau_n}\} \vdash x_0 : \tau_0$ and in e_t we have $A\{x_0 : \tau_y \to \tau_0, \overline{x_n : \tau_n}\} \vdash x_0\ y : \tau_0$ by rule *(app)*.
Second, we have to show that
$$A\{x_0 : GEN(A, \tau_y \to \tau_0), \overline{x_n : GEN(A, \tau_n)}\} \vdash e : \tau.$$
It is easy to see that $\alpha \in FTV(\tau_0) \setminus FTV(A)$ if and only if $\alpha \in FTV(\tau_y \to \tau_0) \setminus FTV(A)$ because $y : \tau_y$ is in A. Hence, $GEN(A, \tau_0) = \forall \alpha_1 \ldots \alpha_k . \tau_0$ and $GEN(A, \tau_y \to \tau_0) = \forall \alpha_1 \ldots \alpha_k . \tau_y \to \tau_0$ for the *same variables* $\alpha_1, \ldots, \alpha_k$. Now the proof is an easy induction on the type derivation for e, the only interesting case being (again) the variable case for x_0 where in e_s we have $A\{x_0 : \forall \alpha_1 \ldots \alpha_k . \tau_0 \ldots\} \vdash x_0 : \tau_0[\overline{\alpha_k \mapsto \tau_k'}]$ and in e_t we have $A\{x_0 : \forall \alpha_1 \ldots \alpha_k . \tau_y \to \tau_0 \ldots\} \vdash x_0\ y : \tau_0[\overline{\alpha_k \mapsto \tau_k'}]$ by one application of *(app)*.
2. Suppose $\sigma = \forall \alpha_1 \ldots \alpha_k . \tau_y$ and $k > 0$. Now, $\lambda y . e_0'$ is well-typed if and only if all occurrences of y in e_0' use the same instance of σ. In this case, the type instantiation is moved from the occurrences of y in e' to the applications $x\ y$.
Otherwise, the transformed expression is not well-typed anymore.

The first case applies in the following occasions, corresponding to e_s occurring in different contexts:

- y is one of $x_1, \ldots, x_n$: these are monomorphically typed in their own definition, by the *(letrec)* rule;
- y is lambda-bound in the context, i.e., the context has the form $C_1[\lambda y . C_2]$ where C_1 is an arbitrary context and C_2 is a context that does not bind y;
- y is bound in a let-expression in the context, but e_s does not occur in the body of the let, i.e., the context has the form $C_1[\texttt{let}\ \{\ldots y = \ldots C_2 \ldots\}\ \texttt{in}\ e']$ (C_2 is not necessarily the right side of y's definition) where C_1 is an arbitrary context and C_2 does not bind y.

The second case applies in only one occasion:

- y is bound in a let-expression in the context, it is used at different types in e_s, and e_s occurs in the body of this let, i.e., the context has the form $C_1[\texttt{let}\ \{\ldots y = \ldots\}\ \texttt{in}\ C_2]$.

The problem Here is an example for the second case (in a slightly extended language with pairs where `fst` and `snd` project the first or second component):

```
let i = fun x -> x in
let h = fun v -> (let y = (i,v) in
                  let g = fun z -> (fst y) (fst y) (snd y) in
                  g v) in
h
```

The variable `y` of type $\forall\alpha.(\alpha \to \alpha) \times \beta$ occurs twice in the definition of `g` at type instances $(\beta \to \beta) \times \beta$ and $((\beta \to \beta) \to (\beta \to \beta)) \times \beta$. Applying the EXPAND transformation to `g` and `y` yields the expression

```
let i = fun x -> x in
let h = fun v -> (let y = (i,v) in
                  let g = fun y z -> (fst y) (fst y) (snd y) in
                  g y v) in
h
```

which does not type check. It is not possible to lift the offending definition `y = (i,v)` to the top-level, due to the occurrence of `v`.

A solution We propose to introduce one parameter for each different type instance at which the variable is used. In the example it amounts to

```
let i = fun x -> x in
let h = fun v -> (let y = (i,v) in
                  let g = fun y y1 z -> (fst y1) (fst y) (snd y) in
                  g y y v) in
h
```

Here is the revised transformation rule EXPAND'.

$$\frac{\begin{array}{c} y \in FV(e_0) \setminus \{x_0, x_1, \ldots, x_n\} \quad z \notin FV(e_0) \\ \text{let } y^{(1)}, \ldots, y^{(m)} \text{ be all occurrences of } y \text{ in } e \\ y^{(1)}, \ldots, y^{(k)} \text{ are used at the same instance type (for some } k \in \{1, \ldots, m\}) \\ e_0' \equiv e_0[y^{(1)} \mapsto z, \ldots, y^{(k)} \mapsto z, x_0 \mapsto x_0\ z] \\ e_i' \equiv e_i[x_0 \mapsto x_0\ y] \\ e' \equiv e[x_0 \mapsto x_0\ y] \end{array}}{\texttt{let } \{x_0 = e_0, \overline{x_n = e_n}\} \texttt{ in } e \leadsto \texttt{let } \{x_0 = \lambda z.e_0', \overline{x_n = e_n'}\} \texttt{ in } e'} \tag{3}$$

This transformation preserves typability.

Proposition 2. *If $A \vdash e_s : \tau$ and $e_s \overset{(3)}{\leadsto} e_t$ then $A \vdash e_t : \tau$.*

In effect, EXPAND' moves the implicit type instantiation at the uses of y in e_0 to the uses of y in the expressions $x_0\ y$. EXPAND' is only applicable a finite number of times because each application removes at least one occurrence of a free variable.

Polymorphic recursion EXPAND' does not help in a language with polymorphic recursion like Haskell [8] or Hope [5] because polymorphic recursion can use a function at an infinite number of different types. The following example (in Haskell syntax) serves as an illustration.

```
data Nlist a = Nil | Fork a (Nlist [a])
flatten :: Nlist a -> [a]
flatten = \nl ->
        case nl of
        Nil -> []
        Fork a (nll) -> a : concat (flatten nll)
```

EXPAND would parameterize `flatten` over `concat` leading to

```
flatten :: ([[a]] -> [a]) -> Nlist a -> [a]
flatten = \concat -> \nl ->
        case nl of
        Nil -> []
        Fork a (nll) -> a : concat (flatten concat nll)
```

This definition does not type check since the two occurrences of `concat` are required at different types: the first at type `[[a]] -> [a]` and the second at type `[[[a]]] -> [[a]]`. Using first-class polymorphism we can provide the correct type for `concat`.

Remarks Johnsson's lambda lifting algorithm [10] only applies rule (2) if e_0 is a lambda abstraction and if y is not bound to a lambda abstraction. Without this restriction, an evaluator might duplicate the evaluation of e_0.

The alternatives to EXPAND' are to duplicate definitions or to introduce parameters to non-functional values, both of which could lead to duplication of computation. Our solution preserves the operational properties.

The EXPAND transformation does not cause problems in an explicitly typed language with first-class polymorphism (such languages are popular intermediate languages, cf. [18, 7]). In such a language, EXPAND would transform $x = e$ with a free variable $y : \sigma$ to $x = \lambda y : \sigma . e[x := x\ y]$ without endangering typability.

The EXPAND' transformation might be beneficial *before* transforming into an explicitly typed language because it decreases the number of type applications. This can be an advantage for an implementation that maintains type information at runtime, for example to do tag-free garbage collection [25] or debugging [1], where a type application actually performs some work. In fact, this motivation has driven Saha and Shao [23] to define a transformation that eliminates all type applications at runtime. Their transformation is quite similar to lambda lifting.

3.3 SPLIT

The SPLIT rule splits up a let-expression in two parts if its bindings are not really mutually recursive.

$$\frac{\bigcup FV(e_i) \cap \{x'_1, \ldots, x'_m\} = \emptyset}{\begin{array}{c}\texttt{let } \{\overline{x_n = e_n}, \overline{x'_m = e'_m}\} \texttt{ in } e \\ \rightsquigarrow \\ \texttt{let } \{\overline{x_n = e_n}\} \texttt{ in let } \{\overline{x'_m = e'_m}\} \texttt{ in } e\end{array}} \tag{4}$$

It is uncritical with respect to typability. Indeed, in the transformed expression the types of $x_1, \ldots, x_n$ are generalized already in the e'_m. Compilers usually apply SPLIT based on the results of a dependency analysis before type checking [16, 17].

3.4 LIFT

The LIFT transformation floats let-bindings outside of other structures. It comprises five rules.

$$\frac{x \notin \bigcup FV(e_i) \setminus \{x_1, \ldots, x_n\}}{\lambda x.\texttt{let } \{\overline{x_n = e_n}\} \texttt{ in } e_0 \rightsquigarrow \texttt{let } \{\overline{x_n = e_n}\} \texttt{ in } \lambda x.e_0} \tag{5}$$

$$(\texttt{let } \{\overline{x_n = e_n}\} \texttt{ in } e_0)\ e \rightsquigarrow \texttt{let } \{\overline{x_n = e_n}\} \texttt{ in } (e_0\ e) \tag{6}$$

$$e\ (\texttt{let } \{\overline{x_n = e_n}\} \texttt{ in } e_0) \rightsquigarrow \texttt{let } \{\overline{x_n = e_n}\} \texttt{ in } (e\ e_0) \tag{7}$$

$$\frac{\{x_1, \ldots, x_n\} \cap (\bigcup FV(e'_j) \setminus \{x'_1, \ldots, x'_m\}) = \emptyset}{\begin{array}{c}\texttt{let } \{\overline{x_n = e_n}\} \texttt{ in let } \{\overline{x'_m = e'_m}\} \texttt{ in } e \\ \rightsquigarrow \\ \texttt{let } \{\overline{x'_m = e'_m}\} \texttt{ in let } \{\overline{x_n = e_n}\} \texttt{ in } e\end{array}} \tag{8}$$

$$\frac{\{x_1, \ldots, x_n\} \cap (\bigcup FV(e'_j) \setminus \{x'_1, \ldots, x'_m\}) = \emptyset}{\begin{array}{c}\texttt{let } \{\ldots x_i = \texttt{let } \{\overline{x'_m = e'_m}\} \texttt{ in } e_i \ldots\} \texttt{ in } e \\ \rightsquigarrow \\ \texttt{let } \{\overline{x'_m = e'_m}\} \texttt{ in let } \{\overline{x_n = e_n}\} \texttt{ in } e\end{array}} \tag{9}$$

Typing properties The LIFT transformation preserves typability.

Proposition 3. *Suppose $A \vdash e_s : \tau$ and $e_s \rightsquigarrow e_t$ by rule (5), (6), (7), (8), or (9).*

Then $A \vdash e_t : \tau$.

The problem So, what's the problem? The problem is that some algorithms [10] replace the last transformation rule (9) by a different rule:

$$\begin{array}{c}\texttt{let}\ \{\ldots x_i = \texttt{let}\ \{\overline{x'_m = e'_m}\}\ \texttt{in}\ e_i \ldots\}\ \texttt{in}\ e \\ \rightsquigarrow \\ \texttt{let}\ \{\overline{x'_m = e'_m}, \overline{x_n = e_n}\}\ \texttt{in}\ e\end{array} \tag{10}$$

This rule can transform a typable expression into an untypable expression as the following example shows. Let $\texttt{null} : \forall\alpha.\alpha\ \texttt{list} \to \texttt{bool}$ and $\texttt{tl} : \forall\alpha.\alpha\ \texttt{list} \to \alpha\ \texttt{list}$.

```
let rec f = fun y ->
            (let rec g = fun x -> if null x then f []
                                                else g (tl x)
             in  if null y then 0
                              else g [0] + g [true])
in  f []
```

This expression is type correct with `g` having type $\forall\alpha.\alpha\ \texttt{list} \to \texttt{int}$. The nested function `g` is the obvious candidate for the LIFT transformation. However, Johnsson's algorithm would not lift the reference to `f` out of `g` since `f` is bound to a lambda expression. Due to the occurrence of `f`, rule (9) is not applicable. Applying rule (10) yields

```
let rec g = fun x -> if null x then f []
                                else g (tl x)
and     f = fun y -> if null y then 0
                                else g [0] + g [true]
in  f []
```

which does not type check because the applications `g [0]` and `g [true]` require `g` to have a polymorphic type. However, the *(letrec)* typing rule requires that `g` is used monomorphically in the right sides of `g` and `f`.

A solution The correct way to deal with the example is to apply EXPAND' first until the lifting rule (9) is applicable. Application of EXPAND' yields

```
let rec f = fun y ->
            (let rec g = fun f x -> if null x then f []
                                                  else g f (tl x)
             in  if null y then 0
                              else g f [0] + g f [true])
in  f []
```

This step preserves typability, because all uses of `f` in the right side of its definition are monomorphic. Applying (9) yields a typable expression:

```
let rec g = fun f x -> if null x then f []
                                 else g f (tl x)
in
let rec f = fun y -> if null y then 0
                               else g f [0] + g f [true]
in
    f []
```

Remarks Again, in a language with polymorphic recursion, the transformation rule (10) would not violate typability: in the transformed term in Sec. 3.4, it would simply annotate `g` with its type scheme.

4 Revised algorithm

In this section, we put together the various pieces in an algorithm.

1. Apply DEFINE exhaustively.
2. Apply EXPAND' to x_0 and y for each expression

$$\texttt{let}\ \{x_0 = e_0, \overline{x_n = e_n}\}\ \texttt{in}\ e$$

   such that $e_0 \equiv \lambda z.e_0'$ and $y \in FV(e_0) \setminus \{x_0\}$ and
   - either y is lambda-bound in the context, i.e., the context has the form $C_1[\lambda y.C_2]$ where C_1 is an arbitrary context and C_2 does not bind y;
   - or the binding $y = e''$ is in the context and $e'' \not\equiv \lambda z.e'''$, i.e., the context has the form $C_1[C[C_2]]$ where C_1 is arbitrary, C_2 does not bind y and C is defined by

$$C ::= \texttt{let}\ \{y = \ldots [\,] \ldots\}\ \texttt{in}\ e \mid \texttt{let}\ \{y = \ldots\}\ \texttt{in}\ [\,]$$

   - or the binding $y = e''$ is in a context of the form $C_1[\texttt{let}\ \{y = \ldots C_2 \ldots\}\ \texttt{in}\ e]$ where C_1 is arbitrary and C_2 does not bind y.
3. Apply SPLIT exhaustively.
4. Apply the LIFT rules (5), (6), (7), (8), and (9) until every let-expression that contains function definitions is top-level, i.e., until it occurs in context T generated by

$$T ::= [\,] \mid \texttt{let}\ \{\overline{x_n = e_n}\}\ \texttt{in}\ T.$$

5 Another Lambda Lifter

The lambda lifter described in Peyton Jones's textbook [17] has a slightly different structure, thus avoiding the problem with monomorphic recursion discussed in Sec. 3.4. Essentially, it employs a different EXPAND transformation that lifts *all* free variables out of lambda abstractions. Since this transformation to combinators comes *before* introducing names, it only applies to the outermost lambda of an expression of the form $\lambda x_1. \ldots \lambda x_n.e$. The context M captures this restriction.

$$M ::= [\,] \mid B'$$
$$B' ::= \lambda x.B' \mid M\ e \mid e\ M \mid \texttt{let}\ \{\ldots x = M \ldots\}\ \texttt{in}\ e \mid \texttt{let}\ \{\ldots\}\ \texttt{in}\ M$$

and the EXPAND1 rule is then

$$\frac{y \in FV(\lambda x.e)}{M[\lambda x.e] \leadsto M[(\lambda y.\lambda x.e)\ y]}$$

This rule has the same type checking problems as the EXPAND rule. They can be solved with an EXPAND1' rule analogous to the EXPAND' rule.

Once a lambda has been turned into a combinator, the DEFINE, SPLIT, and LIFT transformations lift it out to the top-level. The previous problem with the LIFT transformation (see Sec. 3.4) disappears, because EXPAND1' also abstracts uses of other functions, including recursive uses of the function itself (if the original lambda is bound in a let).

Applying EXPAND1' twice (to the free variables `f` and `g` of `fun x -> ...`) to the example from Sec 3.4 yields

```
let rec f = fun y ->
            (let rec g = (fun f -> fun g -> fun x ->
                          if null x then f []
                                    else g (tl x)) f g
             in  if null y then 0
                           else g [0] + g [true])
in  f []
```

Subsequent DEFINE and LIFT transformation steps lead to

```
let rec h = fun f -> fun g -> fun x ->
            if null x then f []
                      else g (tl x)
in
let rec f = fun y ->
            (let rec g = h f g
             in  if null y then 0
                           else g [0] + g [true])
in  f []
```

While the last two expressions are type correct according to our rules, they type check neither in Objective Caml nor in Standard ML. This is caused by the value restriction, which only allows to generalize the types of let-bound syntactic values [14]. While `fun x -> ...` is a syntactic value, an application like `h f g` is not. Eta-expansion of `h f g` is possible, but it duplicates the computation of the value of `h f g`. In contrast, the algorithm that we propose in Sec. 4 neither transforms let-bound values to non-values nor does it transform let-bound non-values to values. While the first transformation only affects typability under the value restriction, the reverse direction can change the semantics in a strict language: a non-terminating non-value expression might be transformed into a (terminating) value by delaying the evaluation of the non-value. Even worse, if the non-value has side effects, these might be executed out of the intended order or they might even be duplicated. Such duplication of computation can even change the complexity from linear to exponential.

In a lazy language, the transformation from value to non-value does not hurt, but the reverse transformation has the potential of duplicating computation and affecting laziness, thus changing the complexity, again.

6 Related work

6.1 Partial evaluation

Malmkjær and Ørbæk [13] investigate the alternative to having a global log: in their scenario, lambda lifting is not necessary and each local function definition has its own log. While this approach gives rise to interesting problems, it seems quite complicated to implement. In addition, as pointed out in the introduction, not having a global log may reduce opportunities for sharing specializations. Especially local functions which are used repeatedly in their scope would benefit. A more global effect could be achieved by merging duplicated lambda-lifted definitions. Given our type preserving lambda lifter, the main point in favor of local logs is increased efficiency in accessing the logs. However, the same efficiency may be obtained by careful structuring of the global log.

The MLmix specializer for Standard ML [3] circumvents the problem with lambda lifting completely: firstly, it requires monomorphic source programs. All lambda lifting algorithms preserve monomorphic typing. Secondly, it does not perform lambda lifting. Instead it employs local logs but imposes additional restrictions with respect to Malmkjær and Ørbæk's work.

6.2 Closure conversion

Although lambda lifting has not been considered for typed languages before, its close cousin closure conversion has. The basic idea of closure conversion is to abstract each function over its free variables and then represent the function by the abstracted function paired with the values of the free variables (the environment). It goes beyond lambda lifting because it reifies the environment as a data structure.

Closure conversion alone does not solve the problem of redundant specialization outlined in the introduction. However, some specialization-time functions need to be closure-converted anyway so that they can be compared. This alone gives rise to typing problems already in a simply-typed setting [24].

Minamide et al [15] demonstrate how to perform closure conversion in an explicitly typed polymorphic intermediate language. They need to employ existential types to avoid type clashes between functions of the same type but with different environments.

The next two approaches differ in that their closures do not contain functions anymore, but rely on separate dispatch functions (*apply*) that inspect the closure's tag and call the respective function [21]. The outcomes of their transformations are first-order programs.

Oliva and Tolmach [27, 26] employ a solution to closure conversion quite similar to ours [24]. It is based on introducing a special closure datatype for each function type that arises the program. They have some refinements that allow them to perform inter-module calls and to incorporate the results of a flow analysis. Their language is monomorphic.

Bell et al [2] consider closure conversion for a polymorphic language. In addition to Oliva and Tolmach's work, their algorithm performs demand-driven monomorphic expansion.

7 Conclusion

We have shown that the traditional algorithms for lambda lifting interact poorly with ML typability. We have analyzed the situation by identifying the elementary steps of lambda lifting and checking their typing properties. We have replaced the faulty steps by correct ones and defined a lambda lifter working as a source-to-source transformation for languages with ML-style polymorphism. Our algorithm works in the presence of the value restriction and it preserves the algorithmic complexity (no duplication of computation and preservation of laziness).

Lambda lifting breaks typability in the presence of polymorphic recursion (e.g., in Haskell). Only the addition of first-class polymorphism regains typability of the lambda lifted program, in general. In cases where type applications perform actual work [25, 1] our EXPAND' transformation can decrease their number before transforming to an explicitly typed calculus.

Finally, our breakdown of lambda lifting into its elementary steps simplifies proofs of further properties of lambda lifting. For example, considering lambda dropping [6] — an inverse to lambda lifting — it is easy to see that only SPLIT applied backwards gives rise to typing problems. It is also easy to show that our lambda lifter does not change the complexity of any function.

Acknowledgment Thanks to Simon Helsen for implementing the transformation.

References

1. Shail Aditya and Alejandro Caro. Compiler-directed type reconstruction for polymorphic languages. In Arvind, editor, *Proc. Functional Programming Languages and Computer Architecture 1993*, pages 74–82, Copenhagen, Denmark, June 1993. ACM Press, New York.
2. Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In Mads Tofte, editor, *Proc. International Conference on Functional Programming 1997*, pages 25–37, Amsterdam, The Netherlands, June 1997. ACM Press, New York.

3. Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Rapport 93/22, DIKU, University of Copenhagen, 1993.
4. Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.
5. Rod Burstall, David MacQueen, and Don Sannella. HOPE: An experimental applicative language. Report CSR-62-80, Computer Science Dept, Edinburgh University, 1980.
6. Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs. In Charles Consel, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '97*, pages 90–106, Amsterdam, The Netherlands, June 1997. ACM Press.
7. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995. ACM Press.
8. Haskell 1.4, a non-strict, purely functional language. April 1997. http://haskell.systemsz.cs.yale.edu/onlinereport/.
9. R. J. M. Hughes. Super-combinators: A new implementation method for applicative languages. In *ACM Symposium on LISP and Functional Programming*, Pittsburgh, Pennsylvania, 1982. ACM Press.
10. Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Functional Programming Languages and Computer Architecture 1985*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.
11. John Launchbury and Carsten Kehler Holst. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218, Skye, Scotland, 1991. Glasgow University.
12. Xavier Leroy. *The Objective Caml system release 1.07, Documentation and user's manual*. INRIA, France, December 1997. From http://pauillac.inria.fr/caml.
13. Karoline Malmkjær and Peter Ørbæk. Polyvariant specialisation for higher-order, block-structured languages. In William Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 66–76, La Jolla, CA, June 1995. ACM Press.
14. Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
15. Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Fla., January 1996. ACM Press.
16. Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *International Symposium on Programming, 6th Colloquium*, pages 217–228, Toulouse, April 1984. Springer-Verlag. LNCS 167.
17. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
18. Simon L Peyton Jones, Cordelia Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, 1993.
19. Simon L. Peyton Jones and David R. Lester. A modular fully-lazy lambda lifter in HASKELL. *Software - Practice and Experience*, 21(5):479–506, April 1991.
20. Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages*. Prentice-Hall, 1992.

21. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740, July 1972.
22. John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing '83*, pages 513–523. North Holland, Amsterdam, 1983.
23. Bratin Saha and Zhong Shao. Optimal type lifting. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation (TIC'98)*, volume 1473 of *Lecture Notes in Computer Science*, pages 57–70, Kyoto, Japan, March 1998.
24. Peter Thiemann. Implementing memoization for partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '96)*, volume 1140 of *Lecture Notes in Computer Science*, pages 198–212, Aachen, Germany, September 1996. Springer-Verlag.
25. Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, pages 1–11, Orlando, Florida, USA, June 1994. ACM Press.
26. Andrew Tolmach. Combining closure conversion with closure analysis using algebraic types. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.
27. Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 1998. To appear.
28. Philip L. Wadler. Theorems for free! In *Proc. Functional Programming Languages and Computer Architecture 1989*, pages 347–359, London, GB, 1989.