Lambda Lifting: Transforming Programs to Recursive Equations

Thomas Johnsson

September 1985*
revised June 1986†

Abstract

Lambda lifting is a technique for transforming a functional program with local function definitions, possibly with free variables in the function definitions, into a program consisting only of global function (combinator) definitions which will be used as rewrite rules. Different ways of doing lambda lifting are presented, as well as reasons for rejecting or selecting the method used in our Lazy ML compiler. An attribute grammar and a functional program implementing the chosen algorithm is given.

---

*Originally publised in *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, Nancy, France, 1985. Springer Verlag.

†As part B of author's thesis. Main addition: the attribute grammar formulation.

1 Introduction

When compiling a lazy functional language using the technique described in [Joh84] it is presumed that the input program is in the form of a set of function definitions, possibly mutually recursive, together with an expression to be evaluated and printed as the value of the program. That is, programs are of the form

$f_1 x_1 ... x_{n_1} = e_1$
...
$f_m x_1 ... x_{n_m} = e_m$
e_0 (the value of the program)

where the function bodies e_i do not contain any lambda expressions, but may contain local definitions with **let** and **letrec**. To summarise the technique, each function definition

$f x_1 ... x_m = e$

is compiled into code for an abstract graph reduction machine, called the G-machine, that performs the graph rewriting

$f e_1 ... e_m \Rightarrow e[e_1 ... e_m / x_1 ... x_m]$

i.e., the graph of an application of the function f is rewritten into the graph for the value of the right hand side, substituting actual parameters e_i for formal parameters x_i.

Although recursive equations as above are a powerful programming language in their own right, for convenience and program clarity it is an advantage to be able to write lambda expressions and locally defined functions.

If we introduce expressions with local function definitions, like

let f x = x*x **in** ... f ...

the compiler would in principle be able to move the function definition out to the global level. Similarly, the compiler would transform the lambda expression $\lambda x.x*x$ into the expression f and also move the function definition $f\ x = x*x$ to the global level (f is a new unique identifier).

This simple scheme of course breaks down if the function body contains free variables; a free variable y as in the function definition $f\ x = ...y...$ would be undefined when the definition is moved to the global level.

Dealing with these free variables in function definitions is the main subject of this paper. The process of flattening out a program involving local function definitions, possibly with free variables, into a program consisting only of global function definitions, we call *lambda lifting*. As we proceed in our discussion, we will make choices which depend of the particular idiosyncrasies of the G-machine. The algorithm described at the end of this paper is used in a compiler for *Lazy ML*, *LML* for short, developed by L. Augustsson and myself. Further details on the compiler can be found in the papers [Aug84, Joh84, Aug85].

2 Different strategies for the transformation

2.1 Attempt 1: Translate everything into lambda expressions

In dealing with programs containing lambda expressions, each lambda expression could be lifted out to become a global function (rewrite rule). (In what follows $\lambda x.\lambda y.e$ is treated as

one lambda expression introducing two variables, rather than two nested lambda expressions. A definition like f x y = e is equivalent to f = λx.λy.e.) Before this can be safely done it is necessary to abstract the free variables from each lambda expression using the transformation rule

$$\lambda x.e \Rightarrow (\lambda y.\lambda x.e)y$$

i.e. beta substitution backwards, for each free variable y in e. The following example illustrates the idea.

$$\underbrace{(\lambda \mathrm{y.f}(\lambda \mathrm{x.y}))5 \Rightarrow}_{\mathrm{G}}$$
$$\underbrace{(\lambda \mathrm{y.f}(\underbrace{(\lambda \mathrm{y}.\lambda \mathrm{x.y})}_{\mathrm{F}}\mathrm{y}))\,5 \Rightarrow \ldots}_{\mathrm{G}}$$

Now the innermost lambda expression indicated by $\underbrace{\ldots}_{\mathrm{F}}$ is given the name F and made into a global function; the same thing is done with the outermost one $\underbrace{\ldots}_{\mathrm{G}}$, yielding:

```
def:   F y x = y
def:   G y = f (F y)
expr:  G 5
```

LML and many other functional languages allow expressions with local definitions. The traditional way of dealing with local nonrecursive and recursive definitions (**let** and **letrec**), both in defining their meaning and in implementations, is to treat them as syntactic sugaring for lambda expressions and the fix-point combinator Y [Lan66] [Tur79] [Hug82]. The following two transformation rules are then used.

let $x = e_1$ **in** $e_2 \Rightarrow (\lambda x.e_2)e_1$
letrec $x = e_1$ **in** $e_2 \Rightarrow$ **let** $x = Y(\lambda x.e_1)$ **in** e_2

The resulting expression contains lambda expressions in place of the **let**s and **letrec**s, and can then be treated as described before. The following example illustrates this.

let i = 5 **in letrec** f = λx.f i **in** f i $\Rightarrow$ { remove recursion }
let i = 5 **in let** f = Y(λf.λx.f i) **in** f i $\Rightarrow$ { remove **let** }
(λi.(λf.f i)(Y (λf.λx.f i))) 5 $\Rightarrow$ { abstract free variables }
(λi.((λi.λf.f i) i) (Y ((λi.λf.λx.f i) i))) 5

Giving names to the lambda expressions, and lifting them to the global level yields the following program.

```
def:   F i f = f i
def:   G i f x = f i
def:   H i = (F i) (Y (G i))
expr:  H 5
```

Treating expressions involving **let** and **letrec** in the manner just described is unsatisfactory for several reasons:

- For efficiency reasons we want a recursive function in the source program to remain recursive in the transformed program, since the *LML* compiler and the G-machine can deal with global recursive functions. Eliminating the recursion by using the fix-point combinator Y thus introduces unnecessary inefficiencies.

- **Let** and **letrec** expressions that involve no lambda expressions need not be tranformed further. They can be directly compiled into G-machine code that constructs or evaluates a shared or cyclic expression graph.

- The effect of this transformation scheme is that every free variable that occurs inside a lambda expression has to be abstracted out of the lambda expression and passed as a function argument all the way down to the place of usage. But if the right hand side of the definition is a lambda expression we would prefer to treat the definition as a global function constant whose occurrence thus would not need to be abstracted out.

Therefore we will turn our attention to lambda lifting schemes that allow us to keep recursion, **let** and **letrec** as far as possible.

2.2 Attempt 2: Keep let and letrec

In our next attempt, we keep the **let** and **letrec** expressions in the program, but otherwise proceed as before with the lambda expressions. For example, in

let i = 5 **in letrec** f = λx.f (i+i) **in** f (i*i)

both *f* and *i* are free in the lambda expression, and so are abstracted out:

let i = 5 **in letrec** f = (λf.λi.λx.f (i+i)) f i **in** f (i*i) $\Rightarrow$

def: F f i x = f (i+i)
expr: **let** i = 5 **in letrec** f = F f i **in** f (i*i)

Unfortunately, this modified strategy suffers from some of the same drawbacks as in the first attempt, namely that local function definitions (i.e. definitions where the right hand side is a lambda expression) are treated as all other definitions: variables are passed as arguments and their occurences inside other lambda expressions are abstracted out of the lambda expressions. In the example above this happened with the variable *f*, which was abstracted out of the definition of *f*. As a consequence, the resulting global function is still not recursive.

Another drawback of this scheme concerns the efficiency of function application in the G-machine setting. When it comes to evaluation of the application *f (i*i)* in the example above *f* has the value of an unreduced curried application of *F* (one argument of *F* is missing in *F f i* to be able to reduce it), and one has to revert to worst-case treatment of the application. On the other hand if the function being applied is a global one, or its graph value is a function node only, and has the same number of arguments as formal parameters, it is possible to evaluate the application much more efficiently. In this particular example *f(i*i)* is a tail call. Since *f* is a variable here, we know of no essentially better simpler way to implement the tail call than to than to build the graph for *f(i*i)* the

hard way. But if *f* were a global function with arity 1, then the call can be implemented much more efficiently with an assignment *i := graphfor(i∗i)* followed by a jump to the code for *f* — see [Joh84] for details. It is also more efficient to build the graph for an application if the function is a global one and the number of argument is the same as the number of formal parameters. In that case we can build a vector application node instead of a chain of apply nodes.

We aim for a strategy where the local function definitions can be lifted out to the global level to become global function constants, so that their occurrences need not be abstracted out, even in the presence of free variables in the function definitions.

2.3 Attempt 3

The trouble with attempt 2 is that after having abstracted out free variables from a local function definition, the right hand side of the function is no longer a lambda expression, because the lambda expression is applied to the abstracted variables. In our third attempt, we instead perform abstraction as follows:

- For each free variable in a local function definition, add an argument to the function (by lambda abstraction) as before. The function names themselves are treated as constants, and need not be abstracted out.
- Apply the same free variables to each use of the function, substituting $fx_1...x_n$ for f, where $x_1...x_n$ is the set of free variables in the definition of f. (If we make all identifiers unique before the whole lambda lifting process starts, no name clash can occur.)

So the only difference between version 2 and version 3 of our lambda lifting strategy is the place where the abstracted variables are passed as arguments. In version 2 the lambda expression of the function definition is applied to the abstracted variables, in version 3 each use of the function is applied to the abstracted variables. But as we shall see, this little difference will have far-reaching effects.

In our example

let i = 5 **in letrec** f = λx.f (i+i) **in** f (i∗i)

the variable *i* is free in the definition of *f*, and we get

let i = 5 **in letrec** f = λi.λx.f i (i+i) **in** f i (i∗i)

Finally, *f* can be safely lifted out to become a global function constant:[1]

def: f i x = f i (i+i)
expr: **let** i = 5 **in** f i (i∗i)

Let us try our new strategy on a nastier example, involving two mutually recursive function definitions, with different free variables in each of them.

[1]Incidentally, the definition *i=5* could also be made global, but here we keep it in the expression merely to introduce *i* as a free variable.

let a = ... **and** b = ...
in letrec f = λx. ... a ... g ...
  and g = λy. ... b ... f ...
  in ... f ... g ...

The variable *a* is free in *f*, and *b* is free in *g*, so if we apply our new lambda lifting strategy here, we get:

let a = ... **and** b = ...
in letrec f = λa.λx. ... a ... g b ...
  and g = λb.λy. ... b ... f a ...
  in ... f a ... g b ...

This step unfortunately introduced the variable *b* in *f* and *a* in *g*, and the function definitions cannot yet be lifted out because of these new free variables. So the lambda insertion step has to be repeated:

let a = ... **and** b = ...
in letrec f = λb.λa.λx. ... a ... g a b ...
  and g = λa.λb.λy. ... b ... f b a ...
  in ... f b a ... g a b ...

Neither *f* nor *g* now contain free variables, so finally we get the global functions, and expression:

def: f b a x = ... a ... g a b ...
def: g a b y = ... b ... f b a ...
expr: **let** a = ... **and** b = ... **in** ... f b a ... g a b ...

From this example it is obvious that if we adopt this abstraction method in our lambda lifting algorithm, the lambda-insertion-and-application has to be repeated until no free variables remain inside lambda expressions. It does not take such a complicated (and perhaps contrived) example as the one above to make repetition necessary. Consider the following one, which is not even recursive.

let x = ...
in let f = λy. ... x ...
  in let g = λz. ... f ...
    in ... f ... g ... ⇒

let x = ...
in let f = λx.λy. ... x ...
  in let g = λz. ... f x ...
    in ... f x ... g ... ⇒

let x = ...
in let f = λx.λy. ... x ...
  in let g = λx.λz. ... f x ...
    in ... f x ... g x ...

Performing lambda lifting in a compiler by repeatedly transforming the program in this manner is very costly. But it is possible to find the set of variables that has to be abstracted out from each function, in a less costly manner.

Let E_f denote the set of variables that has to be abstracted out of the definition of f. Then for each function definition we can set up an equation involving E_f, E_g etc. The resulting system of set equations can then be solved with respect to E_f, E_g etc. Again let us have a look at the 'nasty' example.

letrec f = λx. ... a ... g ...
and g = λy. ... b ... f ...
in ...

E_f obviously contains a, but also the variables that g will be applied to, i.e. the free variables of g. The two set equations we obtain from the example above are the following.

$$E_f = \{a\} \cup E_g$$

$$E_g = \{b\} \cup E_f$$

We now proceed to solve these equations. Substituting the first equation into the second we have

$$E_g = \{b\} \cup (\{a\} \cup E_g) \Rightarrow$$
$$E_g = \{a, b\} \cup E_g$$

which has the least solution

$$E_g = \{a, b\}.$$

and from the first equation we get

$$E_f = \{a, b\}$$

The above solutions now instruct us to

- add λa.λb. ... to the definition of f,
- substitute $f\ a\ b$ for f,

and similarly for g.

Solving the above set equations is equivalent to computing the transitive closure C^* of the relation C, where fCg is true if the function f has an occurrence of the function name g. Then each E_f is obtained by

$$E_f = \bigcup_{g \in X} S_g$$

where $X = \{h \mid fC^*h\}$
and S_g is the set of free variables in the function g.

The best time complexity known for the transitive closure problem is $\mathcal{O}(n^3)$ [AHU76] and that will also be the worst case complexity of our lambda lifting algorithm (n is the number of functions in the program) if all the equations for all the functions in the program are solved in one go.

But in general the situation is not quite as bad as that. In the previous example

let x = ...
in let f = λy. ... x ...
  in let g = λz. ... f ...
    in ... f ... g ...

we can see that f cannot contain g because of the scope rules of the language, so $E_f = \{x\}$ can be obtained directly without having to solve any set equations. Thus we can proceed top-down in the program to be lambda-lifted, and invoke the set equation solving machinery only for each set of mutually recursive function definitions in a **letrec** expression.

3 The lambda lifting algorithm

The lambda lifting algorithm that we finally adopt is based on solving set equations as described in the previous section.

1. Give all identifiers a unique name. (This is done early in the LML compiler, and is called the scope analysis.) This will avoid name clashes when doing the substitutions in step 4. Handling the set equations is also simplified if all functions have distinct names.

2. Anonymous lambda expressions (i.e. those not being the right hand side of a definition) are given names, substituting **let**$f = \lambda x.\lambda y...e$ **in**f for $\lambda x.\lambda y...e$, where in each case f is a new unique identifier. The purpose is to let these lambda expressions take part in the same equation solving machinery as the function definitions, and to give them the names they will have as global functions.

3. Traverse the program top-down. At each **letrec** expression

   letrec $f_1 x_{11}...x_{1m_1} = e_1$
   ... defined functions
   $f_n x_{n1}...x_{nm_n} = e_n$
   $v_1 = e'_1$
   ... defined variables
   $v_m = e'_m$
   in e

   compute the set of variables to be abstracted out of the defined functions, as follows.

   At this point we know three items obtained previously in the top-down traversal of the program:

   vars: is the set of variables in the current scope, i.e. they can occur as free variables in the letrec expression. At the start of the lambda lifting process this set is empty.

   funs: is similarly the set of functions in the current scope; for these functions we already know which variables that has to be abstracted out of the functions. Initially this set is also empty.

   sol: is the solutions of the set equations for the function names in the set funs.

(a) Each of the functions f_i yields a set equation

$$E_{f_i} = S_{f_i} \cup E_g \cup E_h \cup ...$$

where

$$\{g, h, ...\} = \{f_1, ...f_n\} \cap fi(e_i)$$
$$S_{f_i} = ((\{v_1, ...v_m\} \cup vars) \cap fi(e_i)) \cup E_{g_1} \cup ... \cup E_{g_n}$$
$$\{g_1, ...g_n\} = funs \cap fi(e_i)$$
$$fi(e_i) = \text{the free identifiers in } e_i$$

The functions $g, h, ...$ are the ones defined in the currently processed definition list. The functions $g_1 ... g_n$ are the functions defined on an outer level, and E_{g_i} comes from the solutions in sol. Note that if the definitions are not recursive then the set $\{g, h, ...\}$ above is empty and we have the solution $E_{f_i} = S_{f_i}$ directly.

(b) Solve the set equations, for example by repeated substitution.

(c) Continue down in the program tree, with

$$vars := vars \cup \{v_1, ...v_m\}$$
$$funs := funs \cup \{f_1, ...f_n\}$$
$$sol := sol@[(f_1, E_{f_1}); ...(f_n, E_{f_n})]$$

4. For each $E_{f_i} = \{x_1, ...x_m\}$ from the solution of the set equations, perform the following substitutions in the program:

   - in the definition of f_i: $f_i = e \Rightarrow f_i = \lambda x_1 ... \lambda x_m . e$
   - for each occurence of f_i: $f_i \Rightarrow \ f_i x_1 ... x_m$.

5. Lift out the functions to the global level. If a definition list then becomes empty, substitute **let in** $e \Rightarrow e$, and similarly for **letrec**.

What can be said about the complexity of this algorithm? Solving the set equations at each **letrec** expression has complexity $\mathcal{O}(n^2)$ set operations, or $\mathcal{O}(n^3)$ 'basic' operations, where n is the number of functions in the definition list. The rest of the algorithm is linear in the number of set operations, or $\mathcal{O}(n^2)$ 'basic' operations, where n here is the size of the program.

4 An attribute grammar and a functional program for lambda lifting

In this section we elaborate the lambda lifting algoritm in further details and present it in the form of an attribute grammar for the steps 3-5 in the previous section. This attribute grammar is then translated into a functional program evaluating the attributes. A further discussion of attribute grammar based functional programming can be found in [Joh87].

An attribute grammar can be thought of as a decoration of the parse tree with name-value pairs. Normally, the parse tree has been constructed (implicitly or explicitly) during parsing of the input string of lexical symbols. However, when presenting the attribute grammar for lambda lifting, instead of writing the attribute definitions in conjunction with the production rules of a context free grammar, we will write them in conjunction

with patterns that match nodes in the abstract syntax tree. For instance, instead of a production rule

$e_1 \rightarrow$ **letrec** d **in** e_2 *attribute definitions*

we will write

E = LETREC(d,e) *attribute definitions*

where the variables E, d and e correspond to nonterminals with attributes being assigned to them. The pattern *LETREC(d,e)* will appear in the function that evaluates the attributes.

We will deal with abstract syntax trees generated by the following LML type declarations:

```
Expr = LETREC(list(Def) # Expr) + APPL(Id # list(Expr))
Def  = VAR(Id # Expr) + FUN(Id # list(Id) # Expr)
```

For the sake of simplifying the discussion, we consider only recursive local definitions, as non-recursive ones are a degenerate case — no solving of set equations is necessary. *APPL* is curried application of an identifier to a list of expressions. A definition is either a variable definition (*VAR*) or a function definition (*FUN*) where the list of identifiers are the formal parameters of the function. The table below summarizes the cases that will be considered in the attribute grammar.

E = LETREC(d,e)	local recursive definition expression
E = APPL(f,a)	curried application expression
A = e.a	non-empty argument list
A = []	empty argument list
D = VAR(i,e).d	variable definition in a definition list
D = FUN(f,I,e).d	function definition in a definition list
D = []	empty definition list.

Thus variables E and e denote expressions, A and a denote argument lists, i.e. lists of expressions, and D and d denote definition lists.

The set operations used in the lambda lifting algorithm are the following:

Mkset	: list(Id)→ list(Id)	make a "set" of *Id*s from a list of *Id*s
U	: list(Id)→ list(Id)→ list(Id)	∪ (union)
Is	: list(Id)→ list(Id)→ list(Id)	∩ (intersection)
Mem	: Id→ list(Id)→ bool	∈ (membership)

We use lists instead of sets for the following reasons. The order of the identifiers in such a set is important in step 4 of the algorithm: the variables must of course be in the same order when abstracting them out of the function definition, as they occur when the function is applied to them. This can be achieved for example by letting the "set" operations maintain the list sorted (and without duplicates). Also, it is frequently convenient to actually treat the set as a list being concatenated, traversed by map etc. As a concession to readability we will use the conventional symbols ∪, ∩ and ∈ for union, intersection an set membership respectively. We will also use ∅ to denote the empty set (list), and {x} to denote singleton sets (lists).

The entities *vars*, *funs* and *sol* of the previous section are the inherited (i.e. propagating downwards in the tree) attributes of the attribute grammar. We have the following synthesized (i.e. propagating upwards) attributes:

ids: the set of identifiers occuring in the the expression or expression list or definition list, as applicable.
lifted: a list of lifted function definitions,
expr: the new expression (or expression list, or definition list, as applicable) where the lifted functions have been removed.

Furthermore, as a mere programming convenience when constructing the set equations, we have three additional synthesized attributes defined for definition lists:

dvars: the variables defined in the definition list,
dfuns: the functions defined in the definition list,
ss: a list of pairs of function names and the identifiers occuring in each definition.

Below we give the attribute grammar in its entirety, one attribute at a time. Our notation for an attribute *ids* of E, say, is *E'ids*.

```
      E = LETREC(d,e): E'ids = d'ids ∪ e'ids
      E = APPL(f,a):   E'ids = {f} ∪ a'ids
      A = e.a:         A'ids = e'ids ∪ a'ids
      A = []:          A'ids = ∅
      D = VAR(i,e).d:  D'ids = e'ids ∪ d'ids
      D = FUN(f,I,e).d: D'ids = e'ids ∪ d'ids
      D = []:          D'ids = ∅

      E = LETREC(d,e): E'lifted = d'lifted @ e'lifted
      E = APPL(f,a):   E'lifted = a'lifted
      A = e.a:         A'lifted = e'lifted @ a'lifted
      A = []:          A'lifted = []
      D = VAR(i,e).d:  D'lifted = e'lifted @ d'lifted
(1)   D = FUN(f,I,e).d: D'lifted = FUN(f, lookup D'sol f @ I, e'expr) . e'lifted @ d'lifted
      D = []:          D'lifted = []

      E = LETREC(d,e): E'expr = if d'expr=[] then e'expr else LETREC(d'expr,e'expr)
(2)   E = APPL(f,a):   E'expr = APPL(f, toexpr(lookup E'sol f) @ a'expr)
                                where toexpr = map(λi.APPL(i,[]))
      A = e.a:         A'expr = e'expr . a'expr
      A = []:          A'expr = []
      D = VAR(i,e).d:  D'expr = VAR(i,e'expr) . d'expr
(3)   D = FUN(f,I,e).d: D'expr = d'expr
      D = []:          D'expr = []
```

E = LETREC(d,e): d'vars = E'vars ∪ d'dvars
e'vars = E'vars ∪ d'dvars
E = APPL(f,a): a'vars = E'vars
A = e.a: e'vars = A'vars
a'vars = A'vars
D = VAR(i,e).d: d'vars = D'vars
e'vars = D'vars
D = FUN(f,I,e).d: e'vars = D'vars ∪ Mkset(I)
d'vars = D'vars

E = LETREC(d,e): d'funs = E'funs ∪ d'dfuns
e'funs = E'funs ∪ d'dfuns
E = APPL(f,a): a'funs = E'funs
A = e.a: e'funs = A'funs
a'funs = A'funs
D = VAR(i,e).d: d'funs = D'funs
e'funs = D'funs
D = FUN(f,I,e).d: e'funs = D'funs
d'funs = D'funs

(4) E = LETREC(d,e): d'sol = nsol
e'sol = nsol
where nsol = solve(map(λ(f,S).f,
Xset E'sol
(S ∩ E'funs)
(S ∩ (E'vars ∪ d'dvars)),
(S ∩ d'dfuns)
) d'ss) @ E'sol
E = APPL(f,a): a'sol = E'sol
A = e.a: e'sol = A'sol
a'sol = A'sol
D = VAR(i,e).d: d'sol = D'sol
e'sol = D'sol
D = FUN(f,I,e).d: e'sol = D'sol
d'sol = D'sol

D = VAR(i,e).d: D'dvars = {i} ∪ d'dvars
D = FUN(f,I,e).d: D'dvars = d'dvars
D = []: D'dvars = ∅

D = VAR(i,e).d: D'dfuns = d'dfuns
D = FUN(f,I,e).d: D'dfuns = {f} ∪ d'dfuns
D = []: D'dfuns = ∅

D = VAR(i,e).d: D'ss = d'ss
D = FUN(f,I,e).d: D'ss = (f,e'ids).d'ss
D = []: D'ss = []

The meat of the algorithm is contained in the places marked (1)–(4). In (1) the function definition is put in the list of lifted functions (note in (3) that the functions will be absent in the resulting expression, attribute *expr*). At (2) and also in (1) the substitutions of step 4 is performed, where the solutions of the set equations for f is obtained by looking it up in the attribute *sol*. Note that we always perform this lookup even if the identifier is not a function in our sense (i.e. a definition where the right hand side is a lambda expression). Thus when f is not a function *lookup* must return an empty "set". The function *lookup* can be defined as follows.

```
    lookup ((f,s).sol) i = if f=i then s else lookup sol i
||  lookup [] i = []
```

At (4) the steps 3(a) – 3(c) are performed. The set equations for a definition is obtained, solved, and the solution appended to the previously obtained solution. A set equation

$$E_{f_i} = S_{f_i} \cup E_g \cup E_h \cup ...$$

is represented by a triple

$$(f_i,\ S_{f_i},\ \{g,\ h,\ ...\}).$$

A list of such triples, one for each function in the definition list, is constructed from the *ss* attribute of the definition list. The expression

```
Xset E'sol (S ∩ E'funs) (S ∩ (E'vars ∪ d'dvars))
```

computes

$$((\{v_1, ...v_m\} \cup vars) \cap fi(e_i)) \cup E_{g_1} \cup ... \cup E_{g_n}$$

of step 3(a). the function *Xset* can be defined as follows.

```
    Xset sol (f.l) z = lookup sol f ∪ Xset sol l z
||  Xset sol [] z = z
```

The function *solve* solves the set equations. One possible formulation of *solve* which solves by repeated substitution is given below (see next page for the definitions of *map* and *itlist*).

```
    solve [] = []
||  solve ((f,s,e).[]) = [(f,s)]
||  solve ((f,s,e).l) = let sol = solve(map(λ(f1,s1,e1).if f ∈ e1
                                                        then (f1, s ∪ s1, e ∪ e1)
                                                        else (f1, s1, e1)) l)
                        in (f, itlist (λx.λp. lookup sol x ∪ p) e s).sol
```

Finally, below we give a functional program for doing lambda lifting, by evaluating attributes as defined in the attribute grammar above. The function *Lambdalift* contains the auxiliary functions *Le*, *La* and *Ld* doing the actual work in traversing the abstract syntax tree and evaluating the attributes. Apart from taking the tree as an argument, they also take as arguments the inherited attributes, and return a tuple of the synthesized attributes. Thus *Le* takes an expression, *La* takes an expression list and *Ld* takes a definition list as an argument. They also take as arguments the inhertied attributes *vars*,

funs and *sol*. *Le* and *La* returns a triple of the attributes *expr*, *lifted* and *ids*. *Ld* returns a 6-tuple of the attributes *expr*, *lifted*, *ids*, *dvars*, *dfuns*, and *ss*.

Note that although the attribute grammar describes a multipass algorithm, the resulting program traverses the abstract syntax tree only once. Note also the unconventional use of recursion resulting from this method of evaluating attributes, which is particularly conspicuous in the case *Le(LETREC(d,e))*. Components 4-6 of what *Ld* returns is used to compute the set equations for the functions defined in this definition list. These equations are solved, the extended set of solutions *nsol* passed further down in the tree, where *Le*, *La* and *Ld* uses it to compute the first and second component of the tuple. Circular programs such as this one requires lazy evaluation to work. A discussion of such programming techniques can be found in [Bir84] and in [Joh87].

```
Lambdalift expr =
      let Xset sol s z = itlist(λf.λp.U (assocdef f sol []) p) s z in
    letrec solve [] = []
        || solve ((f,s,e).[]) = [(f,s)]
        || solve ((f,s,e).l) =
              let so = solve(map(λt.let (f1,s1,e1) = t
                                    in if Mem f e1 then (f1, U s s1, U e e1) else t) l)
              in (f, itlist (λx.λp. U (assocdef x so []) p) e s).s
     in

    letrec Le (LETREC(d,e)) vars funs sol =
             letrec (ed,dd,id,dvars,dfuns,ss) = Ld d nvars nfuns nsol
               and (ee,de,ie) = Le e nvars nfuns nsol
               and nvars = U vars dvars
               and nfuns = U funs dfuns
               and nsol = solve(map(λ(f,S). f, Xset sol (Is S funs) (Is S nvars), Is S dfuns) ss)@sol
                 in ((if ed=[] then ee else LETREC(ed,ee)), de@dd, U id ie)
        || Le (APPL(f,E)) vars funs sol =
              let (eE,dE,iE) = La E vars funs sol
              in (APPL(f,map(λi.APPL(i,[]))(assocdef f sol []) @ eE), dE, U [f] iE)

      and La [] vars funs sol = ([], [], [])
        || La (e.l) vars funs sol =
              let (ee,de,ie) = Le e vars funs sol
              and (el,dl,il) = La l vars funs sol
              in (ee.el, de@dl, U ie il)

      and Ld [] vars funs sol = ([], [], [], [], [], [])
        || Ld (VAR(x,e).l) vars funs sol =
                 let (ee,de,ie) = Le e vars funs sol
                 and (el,dl,il,dvars,dfuns,ss) = Ld l vars funs sol
                 in (VAR(x,ee).el, de@dl, U ie il, U [x] dvars, dfuns, ss)
        || Ld (FUN(f,I,e).l) vars funs sol =
              let (ee,de,ie) = Le e (U vars (Mkset I)) funs sol
              and (el,dl,il,dvars,dfuns,ss) = Ld l vars funs sol
              in (el, FUN(f,assocdef f sol [] @ I, ee).de@dl, U ie il, dvars, U [f] dfuns, (f, ie).ss)

       in let (newexpr, liftedfuns, $) = Le expr [] [] []
          in LETREC(liftedfuns, newexpr)

Library functions
      map f [] = []
   || map f (x.l) = f x. map f l
 and itlist f [] z = z
   || itlist f (x.l) z = f x (itlist f l z)
 and assocdef x [] def = def
   || assocdef k ((x,y).l) def = if k = x then y else assocdef k l def
```

References

[AHU76] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Algorithms*. Addison-Wesley, Reading, Mass., 1976.

[Aug84] L. Augustsson. A Compiler for Lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, 1984.

[Aug85] L. Augustsson. Compiling Pattern Matching. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, Nancy, France, 1985.

[Bir84] R. S. Bird. Using Circular Programs to Eliminate Multiple Traverals of Data. *Acta Informatica*, 21:239–250, 1984.

[Hug82] R. J. M. Hughes. Super Combinators—A New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, 1982.

[Joh84] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, Montreal, 1984.

[Joh87] T. Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Proceedings 1987 Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 274, Portland, Oregon, U.S.A., 1987. Springer Verlag.

[Lan66] P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–164, 1966.

[Tur79] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software—Practice and Experience*, 9:31–49, 1979.