Lambda-Lifting in Quadratic Time *

Olivier Danvy Ulrik P. Schultz
BRICS † ISIS Katrinebjerg
Department of Computer Science
University of Aarhus ‡

June 17, 2004

*A preliminary version of this article was presented at FLOPS'02 [17].

†Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.

‡IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
E-mail addresses: {`danvy,ups`}`@daimi.au.dk`
Home pages: `http://www.daimi.au.dk/~{danvy,ups}`

Abstract

Lambda-lifting is a program transformation that is used in compilers, partial evaluators, and program transformers. In this article, we show how to reduce its complexity from cubic time to quadratic time, and we present a flow-sensitive lambda-lifter that also works in quadratic time.

Lambda-lifting transforms a block-structured program into a set of recursive equations, one for each local function in the source program. Each equation carries extra parameters to account for the free variables of the corresponding local function *and of all its callees*. It is the search for these extra parameters that yields the cubic factor in the traditional formulation of lambda-lifting, which is due to Johnsson. This search is carried out by computing a transitive closure.

To reduce the complexity of lambda-lifting, we partition the call graph of the source program into strongly connected components, based on the simple observation that *all functions in each component need the same extra parameters and thus a transitive closure is not needed*. We therefore simplify the search for extra parameters by treating each strongly connected component instead of each function as a unit, thereby reducing the time complexity of lambda-lifting from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$, where n is the size of the program.

Since a lambda-lifter can output programs of size $\mathcal{O}(n^2)$, our algorithm is asympotically optimal.

Keywords

Block structure, lexical scope, functional programming, inner classes in Java.

Accepted to the Journal of Functional and Logic Programming (JFLP 55).

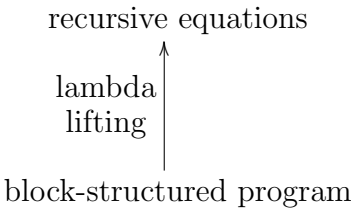
Contents

List of Figures

1 Lambda-lifting

1.1 Setting and background

Lambda-lifting: what. In the mid 1980's, Augustsson, Hughes, Johnsson, and Peyton Jones devised 'lambda-lifting', a meaning-preserving transformation from block-structured functional programs to recursive equations [7, 26, 27, 39].



Recursive equations provide a propitious format for graph reduction because they are scope free.

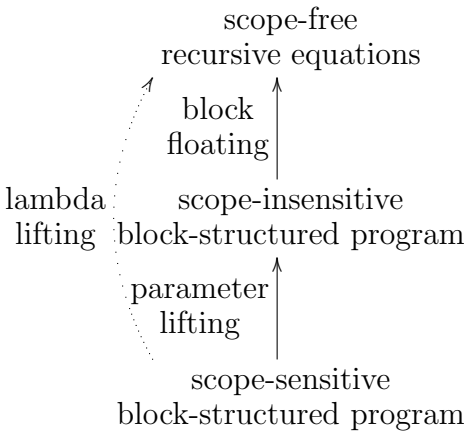
Lambda-lifting: where. Today, a number of systems use lambda-lifting as an intermediate phase: the PAKCS implementation of Curry [23, 24], the Stratego optimizer [44], the Escher compiler [19, Section 3.2.3.1], the PreScheme compiler [37], the Pell-Mell partial evaluator [32], the Schism partial evaluator [13], and the Similix partial evaluator [10] all lambda-lift source programs and generate scope-free recursive equations. Compilers such as Larceny [12] and Moby [40] use local, incremental versions of lambda-lifting in their optimizations, and so did an experimental version of the Glasgow Haskell Compiler [41]. Program generators such as Bakewell and Runciman's least general common generalization operate on lambda-lifted programs [8] and so does Ohori's logical abstract machine [36]. Graunke, Findler, Krishnamurthi, and Felleisen also use lambda-lifting to restructure functional programs for the web [22].

Lambda-lifting, however, is not restricted to functional programs. In Section 6, we show how it is used to compile inner classes in Java.

Lambda-lifting: when. In a compiler, the effectiveness of lambda-lifting hinges on the tension between passing many actual parameters vs. passing few actual parameters, and between referring to a formal parameter vs. referring to a free variable.

In practice, though, programmers often stay away both from recursive equations and from maximally nested programs. Instead, they write in a mixed style that both abides by Perlis's epigram "If you have a procedure with ten parameters, you probably missed some." and by Turner's recommendation that good Miranda style means little nesting. In this mixed style, and to paraphrase another of Perlis's epigrams, one man's parameter is another man's free variable.

Lambda-lifting: how. Lambda-lifting operates in two stages: *parameter lifting* and *block floating*.



A block-structured program is scope-sensitive because of free variables in local functions. Parameter lifting makes a program scope-insensitive by passing extra variables to each function. These variables account both for the free variables of each function but also for variables occurring free further in the call path. Block floating flattens block structure by making each local function a global recursive equation.

Parameter lifting: Parameter-lifting a program amounts to making all the free variables of a function formal parameters of this function. All callers of the function must thus be passed these variables as arguments as well. A set of solutions pairing each function with a least set of additional parameters is built by traversing the program. Each block of locally defined functions gives rise to a collection of set equations describing which variables should be passed as arguments to its local functions. The names of functions, however, are not included in the

sets, since all functions become globally visible when the lambda-lifting transformation is complete. The solution of each set equation extends the current set of solutions, which is then used to analyze the header (i.e., the local declarations) and the body of the block.

Block floating: After parameter lifting, a program is scope insensitive. Block floating is thus straightforward: the program is merely traversed, all local functions are collected and all blocks are replaced by their bodies. The collected function definitions are then appended to the program as global mutually recursive functions, making all functions globally visible.

1.2 Two examples to illustrate lambda-lifting

We first illustrate first-order lambda-lifting with the classical `foldr` functional, and then higher-order lambda-lifting with a use of `foldr` to calculate the value of a polynomial. Throughout, we use Standard ML [34].

Example 1: We consider the classical fold function for lists, defined with a local function.

```
(*  foldr : ('a * 'b -> 'b) * 'b * 'a list -> 'b  *)
fun foldr (f, b, xs)
    = let (*  walk : 'a list -> 'b  *)
          fun walk nil
              = b
            | walk (x :: xs)
              = f (x, walk xs)
      in walk xs
      end
```

This program is block structured because of the local function `walk`. It is scope sensitive because `walk` has two free variables, `f` and `b`.

Parameter-lifting this scope-sensitive block-structured program parameterizes `walk` with `f` and `b`. The result is the following scope-insensitive

block-structured program:

```
(*  foldr : ('a * 'b -> 'b) * 'b * 'a list -> 'b  *)
fun foldr (f, b, xs)
    = let (*  walk : ('a * 'b -> 'b) * 'b -> 'a list -> 'b  *)
          fun walk (f, b) nil
              = b
            | walk (f, b) (x :: xs)
              = f (x, walk (f, b) xs)
      in walk (f, b) xs
      end
```

This program is block structured because of the local function `walk`. It is scope insensitive because `walk` is closed.

Block-floating this scope-insensitive block-structured program yields two scope-free recursive equations. One corresponds to the original entry point, `foldr`, and the other to the local function, `walk`:

```
(*  foldr : ('a * 'b -> 'b) * 'b * 'a list -> 'b  *)
fun foldr (f, b, xs)
    = foldr_walk (f, b) xs
(*  foldr_walk : ('a * 'b -> 'b) * 'b -> 'a list -> 'b  *)
and foldr_walk (f, b) nil
    = b
  | foldr_walk (f, b) (x :: xs)
    = f (x, foldr_walk (f, b) xs)
```

Example 2: We represent the polynomial $c_0 + c_1x + c_2x^2 + c_3x^3 + ... + c_nx^n$ as the list of coefficients $[c_0, c_1, c_2, c_3, ..., c_n]$. Calculating the value of a polynomial at some x is done by traversing the list of coefficients as follows:

```
(*  val_of_pol : int list * int -> int  *)
fun val_of_pol (cs, x)
    = let (*  walk : int * int list -> int  *)
          fun walk (x_n, nil)
              = 0
            | walk (x_n, c :: cs)
              = c * x_n + walk (x * x_n, cs)
      in walk (1, cs)
      end
```

We can also express this function with `foldr`, naming all anonymous functions. The result is the following scope-sensitive block-structured program:

```
(*  val_of_pol : int list * int -> int  *)
fun val_of_pol (cs, x)
    = let (*  cons : int * int -> int  *)
          fun cons (c, a)
              = let (*  aux : int -> int  *)
                    fun aux x_n
                        = c * x_n + a (x * x_n)
                in aux
                end
          (*  null : int -> int  *)
          fun null x_n
              = 0
      in foldr (cons, null, cs) 1
      end
```

Three local functions occur: `cons`, which has one free variable, `x`; `aux`, which has three free variables, `c`, `a`, and `x`; and `null`, which is closed.

Parameter-lifting this scope-sensitive block-structured program parameterizes `cons` with `x` and `aux` with `c`, `a`, and `x`. The result is the following scope-insensitive block-structured program:

```
(*  val_of_pol : int list * int -> int  *)
fun val_of_pol (cs, x)
    = let (*  cons : int -> int * int -> int  *)
          fun cons x (c, a)
              = let (*  aux : int * int * int -> int -> int  *)
                    fun aux (c, a, x) x_n
                        = c * x_n + a (x * x_n)
                in aux (c, a, x)
                end
          (*  null : int -> int  *)
          fun null x_n
              = 0
      in foldr (cons x, null, cs) 1
      end
```

This program is block structured because of the local functions `cons`, `aux`, and `null`. It is scope insensitive because each of these functions is closed.

Block-floating this scope-insensitive block-structured program yields four scope-free recursive equations. One corresponds to the original entry point and the three others to the local functions:

```
(*  val_of_pol : int list * int -> int  *)
fun val_of_pol (cs, x)
    = foldr (val_of_pol_cons x, val_of_pol_null, cs) 1
(*  val_of_pol_cons : int -> int * int -> int  *)
and val_of_pol_cons x (c, a)
    = val_of_pol_cons_aux (c, a, x)
(*  val_of_pol_cons_aux : int * int * int -> int -> int  *)
and val_of_pol_cons_aux (c, a, x) x_n
    = c * x_n + a (x * x_n)
(*  val_of_pol_null : int -> int  *)
and val_of_pol_null x_n
    = 0
```

As illustrated by this example, lambda-lifting naturally handles higher-order functions. Before lambda-lifting, the free variables of a function are implicitly passed at the definition site to construct a closure. Lambda-lifting transforms the definition site into a call site where the free variables are explicitly passed to the lifted function.

In practice, for efficiency, polynomials are usually represented in Horner form $c_0+x(c_1+x(c_2+x(c_3+...)))$ and are calculated more directly as follows:

```
(*  val_of_pol : int list * int -> int  *)
fun val_of_pol (cs, x)
    = foldr (fn (c, a) => c + x * a, 0, cs)
```

This definition has only one functional value with one free variable. It is lambda-lifted into the following recursive equations:

```
(*  val_of_pol : int list * int -> int  *)
fun val_of_pol (cs, x)
    = foldr (val_of_pol_cons x, 0, cs)
(*  val_of_pol_cons : int -> int * int -> int  *)
and val_of_pol_cons x (c, a)
    = c + x * a
```

The extra parameter needed after lambda-lifting, `x`, is explicitly passed to `val_of_ pol_cons`, a technique that was initially developed for the POP-2 programming language [11].

1.3 Two examples to illustrate the time complexity of lambda-lifting

We first consider an example involving multiple local functions and variables occurring free further in any call path, and then an example involving mutually recursive local functions.

Example 1: The following scope-sensitive block-structured program adds its two parameters.

```
(*  main : int * int -> int  *)
fun main (x, y)
    = let (*  add : int -> int  *)
          fun add p
              = add_to_x p
          (*  add_to_x : int -> int  *)
          and add_to_x q
              = q + x
      in add y
      end
```

Two local functions occur: `add`, which is closed, and `add_to_x`, which has one free variable, `x`.

Parameter-lifting this program parameterizes `add_to_x` with `x`, which in turn requires us to parameterize `add` with `x` as well. The result is the following scope-insensitive block-structured program:

```
(*  main : int * int -> int  *)
fun main (x, y)
    = let (*  add : int -> int -> int  *)
          fun add x p
              = add_to_x x p
          (*  add_to_x : int -> int -> int  *)
          and add_to_x x q
              = q + x
      in add x y
      end
```

Block-floating this program yields three recursive equations, one for the original entry point and two for the local functions:

```
(*  main : int * int -> int  *)
fun main (x, y)
    = main_add x y
(*  main_add : int -> int -> int  *)
and main_add x p
    = main_add_to_x x p
(*  main_add_to_x : int -> int -> int  *)
and main_add_to_x x q
    = q + x
```

As illustrated by this example, during parameter lifting, each function needs to be passed not only the value of its free variables, but also the values of the free variables of all its callees.

Example 2: The following scope-sensitive block-structured program multiplies its two parameters with successive additions, using mutual recursion.

```
(*  mul : int * int -> int  *)
fun mul (x, y)
    = let (*  loop : int -> int  *)
          fun loop z
              = if z = 0 then 0 else add_to_x z
          (*  add_to_x : int -> int  *)
          and add_to_x z
              = x + loop (z - 1)
      in loop y
      end
```

Two local functions occur: `loop`, which is closed, and `add_to_x`, which has one free variable, `x`.

Parameter-lifting this program parameterizes `add_to_x` with `x`, which in turn requires us to parameterize its caller `loop` with `x` as well. When `add_to_x` calls `loop` recursively, it must pass the value of `x` to `loop`, so that `loop` can pass it back in the recursive call. The result is the following scope-insensitive

block-structured program:

```
(*  mul : int * int -> int  *)
fun mul (x, y)
    = let (*  loop : int -> int -> int  *)
          fun loop x z
              = if z = 0 then 0 else add_to_x x z
          (*  add_to_x : int -> int -> int  *)
          and add_to_x x z
              = x + loop x (z - 1)
      in loop x y
      end
```

Block-floating this program yields three recursive equations, one for the original entry point and two for the local functions:

```
(*  mul : int * int -> int  *)
fun mul (x, y)
    = mul_loop x y
(*  mul_loop : int -> int -> int  *)
and mul_loop x z
    = if z = 0 then 0 else mul_add_to_x x z
(*  mul_add_to_x : int -> int -> int  *)
and mul_add_to_x x z
    = x + mul_loop x (z - 1)
```

This final example illustrates our insight: during parameter lifting, mutually recursive functions must be passed the same set of free variables as parameters.

1.4 Overview

Lambda-lifting, as specified by Johnsson, takes cubic time (Section 2). We show how to reduce this complexity to quadratic time (Section 3). We also present a flow-sensitive extension to lambda-lifting, where flow information is used to eliminate redundant formal parameters generated by the standard algorithm (Section 4).

Throughout the main part of the article, we consider Johnsson's algorithm [27, 28]. Other styles of lambda-lifting, however, exist: we describe them as well, together with addressing related work (Section 5). Finally, we describe

an instance of lambda-lifting in Java compilers (Section 6) and point out how it could benefit from lambda-lifting in quadratic time.

1.5 Notation and assumptions

We operate on the subset of ML conforming to the simple syntax of Figure 1, where we distinguish between function names and variable names.

$$
\begin{array}{lll}
p \in \mathsf{Program} & ::= & \{d_1, \ldots, d_m\} \\
d \in \mathsf{Def} & ::= & f \equiv \lambda(v_1, \ldots, v_n).e \\
e \in \mathsf{Exp} & ::= & \underline{\text{literal}} \mid v \mid f \mid \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \mid \\
 & & e_0 \ \ldots \ e_n \mid \text{letrec } \{d_1, \ldots, d_k\}\, e_0 \\
v \in \mathsf{Variable} & & \\
f \in \mathsf{FunctionName} \cup \mathsf{PredefinedFunction} & &
\end{array}
$$

Figure 1: Syntax of source programs

Our complexity analysis assumes that sets of variables are represented using bit vectors, where element insertion and removal are performed in constant time and set union is performed in linear time.

The algorithm for parameter lifting, in Figure 9, makes use of a number of graph and list procedures. These procedures are specified in Figure 2.

Graph.add-edge :: $\mathsf{Graph}(\alpha) \to (\alpha, \alpha) \to (\alpha, \alpha)$
Graph.add-edge G (n_1, n_2) : Updates G to contain the nodes n_1 and n_2 as well as an edge between the two.

Graph.coalesceSCC :: $\mathsf{Graph}(\alpha) \to \mathsf{Graph}(\mathsf{Set}(\alpha))$
Graph.coalesceSCC G : Returns G with its strongly connected components coalesced into sets [1].

Graph.reverseBreadthFirstOrdering :: $\mathsf{Graph}(\alpha) \to \mathsf{List}(\alpha)$
Graph.reverseBreadthFirstOrdering G : Returns a list containing the nodes of G, in a reverse breadth-first ordering.

Figure 2: Graph and list procedures

2 Lambda-lifting in cubic time

2.1 Johnsson's parameter-lifting algorithm

Johnsson's parameter-lifting algorithm is shown in Figures 3 and 4. It assumes that variable names are unique. The functions FV and FF map expressions to their set of free variables and of free function names. The algorithm descends recursively through the program structure and calculates the minimal set of variables that are needed by each function. The descent is performed primarily by the function parameterLiftExp, whose parameter S denotes the current set of solutions (i.e., needed variables). The set of solutions is used to compute the set of solutions for each inner scope, by solving set equations describing the dependencies between functions. First, the sets of free variables (V_{f_i}) and free functions (F_{f_i}) are computed, and S is used to extend each V_{f_i} for each free function from the enclosing scope. Then, the free variables are propagated by adding V_{f_i} to V_{f_j} when f_i is in F_{f_j}. The dependencies between the functions can be arbitrarily complex since a function can depend on any variable or function that is lexically visible. In particular, mutually recursive functions depend upon each other, and so they give rise to mutually recursive set equations.

We analyze the complexity of this algorithm as follows. The mutually recursive set equations are solved using fixed-point iteration. A program containing m function declarations gives rise to m set equations. In a block-structured program the functions are distributed across the program, so we solve the set equations in groups as we process each block of local functions. Each set equation unifies $\mathcal{O}(m)$ sets of size $\mathcal{O}(v)$, where v is the number of variables in the program. However, the total size of all the equations is bounded by the size of the program n, so a single iteration involves $\mathcal{O}(n)$ set-union operations. Each set-union operation takes times $\mathcal{O}(v)$, so a single iteration takes time $\mathcal{O}(nv)$. The number of iterations needed to perform a full transitive closure is $\mathcal{O}(m)$, so the time needed to solve all the set equations is $\mathcal{O}(mnv)$, or simply $\mathcal{O}(n^3)$, which is the overall running time of parameter lifting.

Figure 5 shows the standard (globally applied) block-floating algorithm. Johnsson's original lambda-lifting algorithm includes steps to explicitly name anonymous lambda expressions and replace non-recursive let blocks by applications. These steps are trivial and omitted from the figure. To block-float a program of size n, the scope-insensitive functions are simply collected, which

```
parameterLiftProgram :: Program → Program
parameterLiftProgram p = map (parameterLiftDef ∅) p

parameterLiftDef :: Set(FunName × Set(Variable)) → Def → Def
parameterLiftDef S (f ≡ λ(v_1, ..., v_n).e) =
    applySolutionToDef S (f ≡ λ(v_1, ..., v_n).(parameterLiftExp S e))

parameterLiftExp :: Set(FunName × Set(Variable)) → Exp → Exp
parameterLiftExp S literal = literal
parameterLiftExp S v = v
parameterLiftExp S f = applySolutionToExp S f
parameterLiftExp S (if e_0 then e_1 else e_2) =
    let e'_i = parameterLiftExp S e_i  for 0 ≤ i ≤ 2
    in if e'_0 then e'_1 else e'_2
parameterLiftExp S (e_0 ... e_n) =
    let e'_i = parameterLiftExp S e_i  for 0 ≤ i ≤ n
    in e'_0 ... e'_n
parameterLiftExp S (LetRec {d_1, ..., d_k} e_0) =
    foreach (f_i ≡ l_i) ∈ {d_1, ..., d_k} do
        V_{f_i} := FV(l_i);
        F_{f_i} := FF(l_i)
    foreach F_{f_i} ∈ {F_{f_1}, ..., F_{f_k}} do
        foreach (g, V_g) ∈ S such that g ∈ F_{f_i} do
            V_{f_i} := V_{f_i} ∪ V_g;
            F_{f_i} := F_{f_i} \ {g}
    fixpoint over {V_{f_1}, ..., V_{f_k}} by
      foreach F_{f_i} ∈ {F_{f_1}, ..., F_{f_k}} do
          foreach g ∈ F_{f_i} do
              V_{f_i} := V_{f_i} ∪ V_g
    let S' = S ∪ {(f_1, V_{f_1}), ..., (f_k, V_{f_k})}
        f_s = map (parameterLiftDef S') {d_1, ..., d_k}
        e'_0 = parameterLiftExp S' e_0
    in letrec f_s e'_0
```

Figure 3: Johnsson's parameter-lifting algorithm (part 1/2) — time $\mathcal{O}(n^3)$.

$$
\begin{array}{l}
\text{applySolutionToDef} :: \mathsf{Set}(\mathsf{FunName} \times \mathsf{Set}(\mathsf{Variable})) \to \mathsf{Def} \to \mathsf{Def} \\
\text{applySolutionToDef } \{\ldots, (f, \{v_1, \ldots, v_n\}), \ldots\}\ (f \equiv \lambda(v'_1, \ldots, v'_{n'}).e) = \\
\quad (f \equiv \lambda(v_1, \ldots, v_n).\lambda(v'_1, \ldots, v'_{n'}).e) \\
\text{applySolutionToDef } S\ d = d \\
\\
\text{applySolutionToExp} :: \mathsf{Set}(\mathsf{FunName} \times \mathsf{Set}(\mathsf{Variable})) \to \mathsf{Exp} \to \mathsf{Exp} \\
\text{applySolutionToExp } (S \textbf{ as } \{\ldots, (f, \{v_1, \ldots, v_n\}), \ldots\})\ f = \\
\quad (f\ (v_1\ \ldots\ v_n)) \\
\text{applySolutionToExp } S\ e = e
\end{array}
$$

Figure 4: Johnsson's parameter-lifting algorithm (part 2/2) — time $\mathcal{O}(n^3)$.

can be done in one pass and therefore in time $\mathcal{O}(n)$. Therefore, the overall running time of Johnsson's lambda-lifting algorithm is $\mathcal{O}(n^3)$.

2.2 An alternative specification based on graphs

Lambda-lifting can be specified with a graph rather than with set equations. This graph describes the lexical dependencies between source functions. Peyton Jones also uses such a dependency graph [39], but for a different purpose (see Section 5.1). Each node in the graph corresponds to a function in the program, and is associated with the free variables of this function. An edge in the graph from a node `f` to a node `g` indicates that the function `f` depends on `g`, because it refers to `g`. Mutually recursive dependencies give rise to cycles in this graph. Rather than solving mutually recursive set equations, we propagate the variables associated with each node backwards through the graph, from callee to caller, merging the variable sets, until a fixed point is reached.

2.3 Example

Figure 6 shows a small program that uses three mutually recursive functions, each of which has a different free variable.

We can describe the dependencies between the local block of functions using set equations, as shown in Figure 7. To solve these set equations, we need to perform three fixed-point iterations, since there is a cyclic dependency

$\text{blockFloatProgram} :: \mathsf{Program} \to \mathsf{Program}$
$\text{blockFloatProgram}\ \{d_1, \ldots, d_m\} =$
$\quad (\text{blockFloatDef}\ d_1) \cup \ldots \cup (\text{blockFloatDef}\ d_m)$

$\text{blockFloatDef} :: \mathsf{Def} \to \mathsf{Set(Def)}$
$\text{blockFloatDef}\ (f \equiv \lambda(v_1, \ldots, v_n).e) =$
$\quad \textbf{let}\ (F,\ e') = \text{blockFloatExp}\ e$
$\quad \textbf{in}\ F \cup \{f \equiv \lambda(v_1, \ldots, v_n).e'\}$
$\text{blockFloatExp} :: \mathsf{Exp} \to \mathsf{Set(Def)} \times \mathsf{Exp}$
$\text{blockFloatExp}\ \underline{\text{literal}} = (\emptyset,\ \underline{\text{literal}})$
$\text{blockFloatExp}\ v = (\emptyset,\ v)$
$\text{blockFloatExp}\ f = (\emptyset,\ f)$
$\text{blockFloatExp}\ (\text{if}\, e_0\, \text{then}\, e_1\, \text{else}\, e_2) =$
$\quad \textbf{let}\ (F_i,\ e_i') = \text{blockFloatExp}\ e_i\ \ \textbf{for}\ 0 \leq i \leq 2$
$\quad \textbf{in}\ (F_0 \cup F_1 \cup F_2,\ \text{if}\, e_0'\, \text{then}\, e_1'\, \text{else}\, e_2')$
$\text{blockFloatExp}\ (e_0\ \ldots\ e_n) =$
$\quad \textbf{let}\ (F_i,\ e_i') = \text{blockFloatExp}\ e_i\ \ \textbf{for}\ 0 \leq i \leq n$
$\quad \textbf{in}\ (F_0 \cup \ldots \cup F_n,\ e_0'\ \ldots\ e_n')$
$\text{blockFloatExp}\ (\text{letrec}\, \{d_1, \ldots, d_k\}\, e_0) =$
$\quad \textbf{let}\ F = (\text{blockFloatDef}\ d_1) \cup \ldots \cup (\text{blockFloatDef}\ d_k)$
$\qquad (F_0,\ e_0') = \text{blockFloatExp}\ e_0$
$\quad \textbf{in}\ (F \cup F_0,\ e_0')$

Figure 5: Block floating: block structure is flattened

```
fun main (x, y, z, n)
    = let fun f1 i
              = if i = 0 then 0 else x + f2 (i - 1)
          and f2 j
              = let fun g2 b = b * j
                in if j = 0 then 0 else g2 y + f3 (j - 1)
                end
          and f3 k
              = let fun g3 c = c * k
                in if k = 0 then 0 else g3 z + f1 (k - 1)
                end
      in f1 n
      end
```

Figure 6: Three mutually recursive functions

$$\left(\begin{array}{rcl} S_{\mathtt{f1}} & = & \{\mathtt{x}\} \cup S_{\mathtt{f2}} \\ S_{\mathtt{f2}} & = & \{\mathtt{y}\} \cup S_{\mathtt{f3}} \\ S_{\mathtt{f3}} & = & \{\mathtt{z}\} \cup S_{\mathtt{f1}} \end{array}\right)$$
$$S_{\mathtt{g2}} = \{\mathtt{j}\}$$
$$S_{\mathtt{g3}} = \{\mathtt{k}\}$$

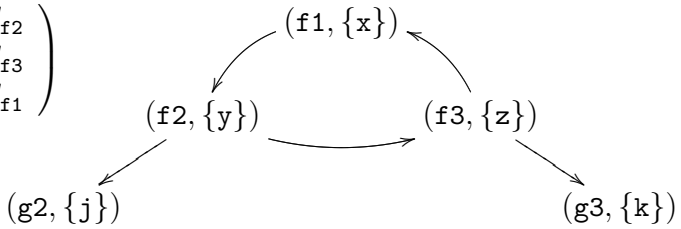


Figure 7: Dependencies between the local functions in Figure 6

```
fun main (x, y, z, n)
    = f1 (x, y, z) n
and f1 (x, y, z) i
    = if i = 0 then 0 else x + f2 (x, y, z) (i - 1)
and f2 (x, y, z) j
    = if j = 0 then 0 else g2 j y + f3 (x, y, z) (j - 1)
and g2 j b
    = b * j
and f3 (x, y, z) k
    = if k = 0 then 0 else g3 k z + f1 (x, y, z) (k - 1)
and g3 k c
    = c * k
```

Figure 8: Lambda-lifted counterpart of Figure 6

of size three. Similarly, we can describe these dependencies using a graph, also shown in Figure 7. The calculation of the needed variables can be done using this representation, by propagating variable sets backwards through the graph. A single propagation step is done by performing a set union over the variables associated with a node and the variables associated with its successors. Similarly to the case of the set equations, each node must be visited three times before a fixed point is reached.

When the set of needed variables has been determined for each function, solutions describing how each function must be expanded with these variables are added to the set of solutions. The result is shown in Figure 8.

3 Lambda-lifting in quadratic time

3.1 The basic idea

We consider the variant of the parameter-lifting algorithm that operates on a dependency graph (Section 2.2). This variant propagates needed variables backwards through the graph since the caller needs the variables of each callee.

It is our observation that functions that belong to the same strongly connected component of the call graph must be parameter-lifted with the same

set of variables (as was illustrated in Section 1.3). We can thus treat these functions in a uniform fashion, by coalescing the strongly connected components of the dependency graph. Each function must define at least its free variables together with the free variables of the other functions of the strongly connected component. Coalescing the strongly connected components of the dependency graph produces a directed acyclic graph with sets of function names for nodes. A breadth-first backwards propagation of variables can then be done in linear time, which eliminates the need for a fixed-point computation.

Our contribution is

- to characterize the fixed-point operations on the set equations as propagation through the dependency graph, and
- to recognize that functions in the same strongly connected component require the same set of variables.

We can therefore first determine which variables need to be known by each function in a strongly connected component, and then add them as formal parameters to these functions. In each function, those variables not already passed as parameters to the function should be added as formal parameters.

This approach can be applied locally to work like Johnsson's algorithm, processing each block independently. It can also be applied globally to the overall dependency graph. The global algorithm limits the propagation of free variables to their point of definition.

3.2 The new algorithm

Figure 9 shows the main part of our (locally applied) $\mathcal{O}(n^2)$ parameter-lifting algorithm; the remainder of the algorithm is identical to Johnsson's algorithm as presented in Figures 3 and 4. The algorithm makes use of several standard graph and list operations that were described in Figure 2, page 14. Again, the set of solutions S is constructed during the recursive descent of the program structure by the function parameterLiftExp. For each block of mutually recursive functions, the dependency graph is constructed and the strongly connected components are coalesced. The local function propagateFunNames propagates free variables through the graph, as follows. For each node, the solution for all functions associated with this node is extended with the solutions of the other functions from that node and the solutions of all the

```
parameterLiftExp S literal = literal
parameterLiftExp S v = v
parameterLiftExp S f = applySolutionToExp S f
parameterLiftExp S (if e0 then e1 else e2) =
    let e'_i = parameterLiftExp S e_i  for 0 ≤ i ≤ 2
    in if e'_0 then e'_1 else e'_2
parameterLiftExp S (e_0 ... e_n) =
    let e'_i = parameterLiftExp S e_i  for 0 ≤ i ≤ n
    in e'_0 ... e'_n
parameterLiftExp S (LetRec {d_1, ..., d_k} e_0) =
    let G = ref (∅, ∅)
        V_{f_i} = ref (FV(f_i))  for 1 ≤ i ≤ k and d_i = (f_i ≡ λ(v_1, ..., v_n).e)
    in foreach f_i ∈ {f_1, ..., f_k} do
           foreach g ∈ FF(f_i) ∩ {f_1, ..., f_k} do
               Graph.add-edge G( f_i, g)
       let (G' as (V', E')) = Graph.coalesceSCC G
           succ_p = {q ∈ V' | (p, q) ∈ E'}, for each p ∈ V'
           F_p = ⋃_{q∈succ_p} q, for each p ∈ V'
           propagateFunNames :: List(Set(FunName)) → ()
           propagateFunNames [] = ()
           propagateFunNames (p :: r) =
               let V = (⋃_{f∈p} V_f) ∪ (⋃_{g∈F_p} V_g)
               in foreach f ∈ p do V_f := V;
                  propagateFunNames r
       in propagateFunNames (Graph.reverseBreadthFirstOrdering G');
       let S' = S ∪ {(f_1, V_{f_1}), ..., (f_k, V_{f_k})}
           f_s = map (parameterLiftDef S') {d_1, ..., d_k}
           e'_0 = parameterLiftExp S' e_0
       in letrec f_s e'_0
```

Figure 9: The improved parameter lifting algorithm — time $\mathcal{O}(n^2)$

successor nodes. To achieve the backward propagation, the nodes are processed in reverse breadth-first ordering, so that the successors of a node are processed before the node itself.

3.3 Revisiting the example of Section 2.3

Applying the algorithm of Figure 9 to the program of Figure 6 processes the function `main` by processing its body. The letrec block of the body is processed by first constructing a dependency graph similar to that shown in Figure 7 (except that we simplify the description to not include the sets of free variables in the nodes). Coalescing the strongly connected components of this graph yields three nodes, one of which contains the three functions {`f1`,`f2`,`f3`}. The free variables of `g2` and `g3` are propagated backwards to their callees. For the node containing {`f1`,`f2`,`f3`}, the propagation step serves to associate each function in the node with the union of the free variables of each of the functions in the component. These variable sets directly give rise to a new set of solutions.

Each of the functions defined in the letrec block and its body are traversed and expanded with the variables indicated by the set of solutions. Block floating according to the algorithm of Figure 5 yields the program of Figure 8.

3.4 Complexity analysis

The parameter-lifting algorithm must first construct the dependency graph, which is done by computing the sets of free functions; this takes time $\mathcal{O}(n)$, where n is the size of the program. The resulting graph has m nodes, where m is the number of local functions, and $\mathcal{O}(n)$ edges (one edge for every free function). Each node contains a set of size $\mathcal{O}(v)$, where v is the number of variables in the program. The strongly connected components of the graph can then be computed in time $\mathcal{O}(m+n)$, or simply $\mathcal{O}(n)$, whereas coalescing the nodes takes time $\mathcal{O}(mv)$. The reversed breadth-first ordering of these nodes can then be computed in time $\mathcal{O}(n)$. The ensuing propagation requires one step for each node since we are now operating on a directed acyclic graph. Each propagation step consists of a number of set operations, each of which take at most $\mathcal{O}(v)$ time. Specifically, computing the set V inside the function propagateFunNames for a given node consists of unifying the variable sets associated with the functions of the node and the variable sets associated with the successor nodes (which have already been computed). Thus, the total number of sets which are unified over the $\mathcal{O}(m)$ steps is bounded by the number of edges in the graph. The number of edges is bounded by the number of function calls in the program, which is bounded by the size of

```
fun main x_1 ... x_k y =
    let fun f_1 z
            = f_2 (z + x_1)
        ...
        and f_k z
            = f_1 (z + x_k)
    in f_1 y
    end
```

Figure 10: Lower-bound example

the program $\mathcal{O}(n)$. Each set union operation takes time $\mathcal{O}(v)$, so the overall running time is $\mathcal{O}(nv)$ or simply $\mathcal{O}(n^2)$, where n is the size of the program and v is the number of variables.

3.5 Lower bound and optimality

Consider the program shown in Figure 10. The main function has k formal parameters $\{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ and declares k mutually recursive local functions, each of which has a different variable from $\{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ as a free variable. For this program, $k = \Theta(n) = \Theta(m)$, where n is the size of the program and m is the number of functions in the program. Lambda-lifting this program produces the program shown in Figure 11. This program has k new global functions, each of which has been expanded with the k formal parameters of the formerly enclosing function. The output program is therefore of size $\Omega(m^2)$, which in this case is also $\Omega(n^2)$. One thus cannot perform lambda-lifting faster than $\Omega(n^2)$, which means that our time complexity of $\mathcal{O}(n^2)$ is optimal. The lower bound also implies that our algorithm operates in time $\Theta(n^2)$.

In contrast, Johnsson's algorithm operates in time $\mathcal{O}(n^3)$. Again, we can use the program of Figure 10 to find a lower bound. Johnsson's algorithm will for this program construct k set equations which each perform one set union operation. To solve the set equations, k propagation steps are needed: the free variable of each function must be propagated to all the other functions. Since the sets grow by one element at each step, propagation step i operates on sets of size i. On average, each step takes time $\Theta(\frac{k}{2}k) = \Theta(k^2)$. The total time taken for this program is thus $\Theta(k^3)$ which in this case is $\Theta(n^3)$.

```
fun main x_1 ... x_k y =
    let fun f_1 (x_1, ..., x_k) z
            = f_2 (x_1, ..., x_k) (z + x_1)
        ...
        and f_k (x_1, ..., x_k) z
            = f_1 (x_1, ..., x_k) (z + x_k)
    in f_1 y
    end
```

Figure 11: The program of Figure 10, after parameter-lifting

Johnsson's algorithm thus has a worst-case lower bound of $\Omega(n^3)$ [14]. As shown above, for this worst-case program, our algorithm operates in time $\Theta(n^2)$.

4 Flow-sensitive lambda-lifting in quadratic time

The value of a free variable might be available within a strongly connected component under a different name. Johnsson's algorithm (and therefore our algorithm as well), however, includes all the variables from the outer scopes as formal parameters because it only looks at their name. It therefore can produce redundant lambda-lifted programs with aliasing.

4.1 A simple example of aliasing

The following program adds its parameter to itself.

```
(*  main : int -> int  *)
fun main x
    = let (*  add : int -> int  *)
          fun add y
              = x + y
      in add x
      end
```

In the definition of `add`, the free variable `x` is an alias of the formal parameter

`y`. (NB. Unless one were willing to duplicate the definition of `add`, there would be no such aliasing if there were additional calls to `add` with other actual parameters than `x`.)

Lambda-lifting this program yields two recursive equations:

```
(*  main : int -> int  *)
fun main x
    = main_add x x
(*  main_add : int -> int -> int  *)
and main_add x y
    = x + y
```

The extraneous parameter of the second recursive equation illustrates the aliasing mentioned above. Such aliased parameters can for example occur after macro expansion, inlining, refactoring, or partial evaluation

In extreme cases, the number of extraneous parameters can explode: consider the lower bound example of Section 3.5, where if the k formal parameters were aliases, there would be $\Theta(k^2)$ extraneous parameters. Such extra parameters can have a dramatic effect. For example, Appel's compiler uses register-allocation algorithms that are not linear in the arity of source functions [2]. Worse, in partial evaluation, one of Glenstrup's analyses is exponential in the arity of source functions [21].

4.2 Handling aliasing

Making the lambda-lifting algorithm context-sensitive would require us to look at the flow graph of the source program, as we did for a related transformation, lambda-dropping [16]. Variables coming from an outer scope that are present in a strongly connected component and that retain their identity through all recursive invocations do not need to be added as formal parameters. Doing so would solve the aliasing problem and yield what we conjecture to be "optimal lambda-lifting."

Looking at the flow graph is achieved by a first-order flow analysis that computes the unique definition point (if any) of the value bound to each formal parameter of the first-order functions of the program. Such a use/def-chain analysis works in one pass and therefore its time complexity is linear in the size of the program.

The parameter-lifting algorithm presented in Figure 9 can be modified to perform flow-sensitive lambda-lifting. Given a program annotated with

use/def chains, parameter lifting proceeds as in the flow-insensitive case, except that a free variable already available as a formal parameter is not added to the set of solutions, but is instead substituted for the formal parameter that it aliases. The block-lifting algorithm remains unchanged. Since the time complexity of use/def-chain analysis is linear, the overall time complexity of the flow-sensitive lambda-lifting algorithm is quadratic.

4.3 Revisiting the example of Section 4.1

Getting back to the program of Section 4.1, the flow-sensitive lambda-lifter yields the following recursive equations.

```
(*  main : int -> int  *)
fun main x
    = main_add x
(*  main_add : int -> int  *)
and main_add y
    = y + y
```

This lambda-lifted program does not have redundant parameters.

5 Related work

We review alternative approaches to handling free variables in higher-order, block-structured programming languages, namely supercombinator conversion, closure conversion, lambda-dropping, and incremental versions of lambda-lifting and closure conversion. Finally, we address the issues of formal correctness and typing.

5.1 Supercombinator conversion

Peyton Jones's textbook describes the compilation of functional programs towards the G-machine [39]. Functional programs are compiled into supercombinators, which are then processed at run time by graph reduction. Supercombinators are closed lambda-expressions. Supercombinator conversion [18, 26, 38] generalizes bracket abstraction [9] and produces a series of closed terms. It thus differs from lambda-lifting that produces a series of

mutually recursive equations where the names of the equations are globally visible [35].

Peyton Jones also uses strongly connected components for supercombinator conversion. First, dependencies are analyzed in a set of recursive equations. The resulting strongly connected components are then topologically sorted and the recursive equations are rewritten into nested letrec blocks. There are two reasons for this design: (1) it makes type-checking faster and more precise; and (2) it reduces the number of parameters in the ensuing supercombinators. Supercombinator conversion is then used to process each letrec block, starting outermost and moving inwards. Each function is expanded with its own free variables, and made global under a fresh name. Afterwards, the definition of each function is replaced by an application of the new global function to its free variables, including the new names of any functions used in the body. This application is mutually recursive in the case of mutually recursive functions, relying on the laziness of the source language; it effectively creates a closure for the functions.

Peyton Jones's algorithm thus amounts to first applying dependency analysis to a set of mutually recursive functions and then to perform supercombinator conversion. As for dependency analysis, it is only used to optimize type checking and to minimize the size of closures.

In comparison, applying our algorithm locally to a letrec block would first partition the functions into strongly connected components, like dependency analysis. We use the graph structure, however, to propagate information, not to obtain an ordering of the nodes for creating nested blocks. We also follow Johnsson's algorithm, where the names of the global recursive equations are free in each recursive equations, independently of the evaluation order. Johnsson's algorithm passes all the free variables that are needed by a function and its callees, rather than just the free variables of the function.

To sum up, Peyton Jones's algorithm and our revision of Johnsson's algorithm both coalesce strongly connected components in the dependency graph, but for different purposes, our purpose being to reduce the time complexity of lambda-lifting from cubic to quadratic.

5.2 Closure conversion

The notion of closure originates in Landin's seminal work on functional programming [30]. A closure is a functional value and consists of a pair: a code pointer and an environment holding the denotation of the variables that oc-

cur free in this code. Making this pair explicit in the source program is called 'closure conversion'; it yields scope-insensitive programs, and is a key step in Standard ML of New Jersey [5, 6]. Closure conversion is akin to supercombinator conversion, though in the case of mutually recursive definitions, the closure environments hold the values of the free variables of the mutually recursive definitions, whereas in supercombinator conversion, closures are created through a mutually recursive application.

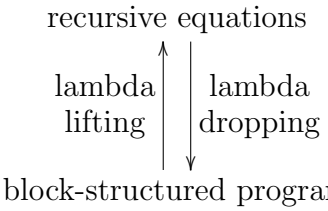
In his textbook [39], Peyton Jones concluded his comparison between lambda-lifting and supercombinator/closure conversion by pointing out a tension between

- passing all the [denotations of the] free variables of all the callees but not the values of the mutually recursive functions (in lambda-lifting), and
- passing all the values of the mutually recursive functions but not the free variables of the callees (in closure conversion).

He left this tension unresolved, stating that future would tell which algorithm (lambda-lifting or closure conversion) would prevail. Today, most compilers for functional languages (Haskell, ML, Scheme) use closure conversion, most compilers for functional logic languages (Curry, Escher) use lambda-lifting, and most program transformers (Similix, Stratego, etc.) use lambda-lifting.

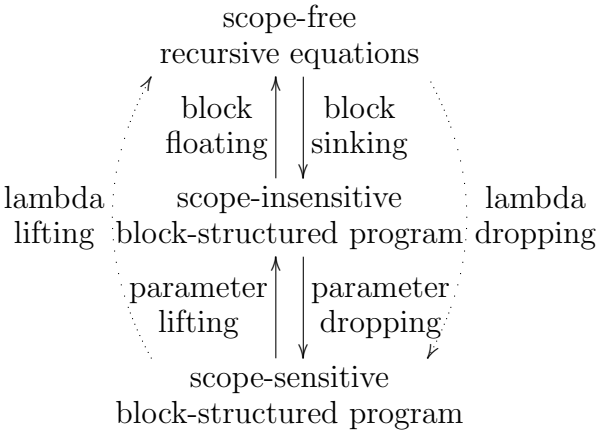
5.3 Lambda-dropping

Lambda-dropping is the inverse of lambda-lifting [16]:



We developed it to be able to compile programs after program transformation. Indeed program transformers tend to be geared to lambda-lifted source programs and they tend to yield lambda-lifted residual programs. In contrast, compilers tend to be geared to source programs written by humans

and therefore with few parameters.[1] Therefore, high numbers of formal parameters are not optimized and often they form a major run-time overhead to invoke procedures. Lambda-dropping reduces the number of formal parameters by restoring block structure:



The block-floating phase of lambda-lifting is reversed by a block-sinking phase. This phase creates block structure by (1) creating local blocks and (2) relocating the definition of functions that are used in only one function into the local block of this function. The parameter-lifting phase of lambda-lifting is reversed by a parameter-dropping phase. This phase removes redundant formal parameters that are originally defined in an outer scope and that always take on the same value.

A few years ago, Appel pointed out a correspondence between imperative programs in SSA form and functional programs using block structure and lexical scope [3]; he showed how to transform an SSA program into its functional representation [4]. We were struck by the fact that this transformation corresponds to performing block sinking on the recursive equations defining the program. As for the transformation into optimal SSA form (which diminishes the number of Φ-nodes), it is equivalent to parameter dropping. Lambda-dropping can therefore be used to transform programs in SSA form into optimal SSA form [16]. This observation prompted us to improve the complexity of the lambda-dropping algorithm to $\mathcal{O}(n \log n)$, where n is the size of the program, by using the dominance graph of the dependency graph. We then re-stated lambda-lifting in a similar framework using graph algo-

[1] For example, the magic numbers of parameters, in OCaml, are 0 to 7.

rithms, which led us to the result presented in the present article.

5.4 Flow sensitivity, revisited

We observe that lambda-dropping is flow sensitive, in the sense that it removes the aliased parameters identified as a possible overhead for lambda-lifting in Section 4. Therefore flow-sensitive lambda-lifting can be achieved by first lambda-dropping the program, and then lambda-lifting the result in the ordinary flow-insensitive way. Since the time complexity of lambda-dropping is lower than the time complexity of lambda-lifting and since lambda-dropping never increases the size of the program, using lambda-dropping as a preprocessing transformation does not increase the overall time complexity of lambda-lifting.

5.5 Mixed style

In order to preserve code locality, compilers such as Twobit [12] or Moby [40] often choose to lift parameters only partially. The result is in the mixed style described in Section 1.1.

In more detail, rather than lifting all the free variables of the program, parameter lifting is used incrementally to lift only a subset of the free variables of each function. If a function is to be moved to a different scope, however, it needs to be passed the free variables of its callees as parameters. As is the case for global lambda-lifting, propagating the additional parameters through the dependency graph requires cubic time. To improve the time complexity, our quadratic-time parameter-lifting algorithm can be applied to the subsets of the free variables instead. The improvement in time complexity for incremental lambda-lifting is the same as what we observed for the global algorithm.

We note that a partial version of closure conversion also exists, namely Steckler and Wand's [42], that leaves some variables free in a closure because this closure is always applied in the scope of these variables. We also note that combinator-based compilers [45] can be seen as using a partial supercombinator conversion.

5.6 Correctness issues

Only idealized versions of lambda-lifting and lambda-dropping have been formally proven correct. Danvy has related the fixed points of lambda-lifted functionals and of lambda-dropped functionals [15]. Fischbach and Hannan have capitalized on the symmetry of lambda-lifting and lambda-dropping to formalize them in a logical framework, for a simply typed source language [20].

Nevertheless, although there is little doubt that Johnsson's original algorithm is correct, its formal correctness still remains to be established.

5.7 Typing issues

Fischbach and Hannan have shown that lambda-lifting is type-preserving for simply typed programs [20]. Thiemann has pointed out that lambda-lifted ML programs are not always typeable, due to let polymorphism [43]. Here is a very simple example. In the following block-structured program, the locally defined function has type `'a -> int`.

```
fun main ()
    = let fun constant x
              = 42
      in (constant 1) + (constant true)
      end
```

The corresponding lambda-lifted program, however, is not typeable because of ML's monomorphic recursion rule [34]. Since `constant` is defined recursively, its name is treated as lambda-bound, not let-bound:

```
fun main ()
    = (constant 1) + (constant true)
and constant x
    = 42
```

The problem occurs again when one of the free variables of a local recursive function is polymorphically typed.

To solve this problem, one could think of making lambda-lifting yield not just one but several groups of mutually recursive equations, based on a dependency analysis [39]. This would not, however, be enough because a

local polymorphic function that calls a surrounding function would end up in the same group of mutually recursive equations as this surrounding function.

There is no generally accepted solution to the problem. Thiemann proposes to parameter-lift some function names as well, as in supercombinator conversion [43]. Fischbach and Hannan propose to use first-class polymorphism instead of let-polymorphism [20]. Yet another approach would be to adopt a polymorphic recursion rule, i.e., to shift from the Damas-Milner type system to the Milner-Mycroft type system, and to use a dependency analysis as in a Haskell compiler. Milner-Mycroft type inference, however, is undecidable [25] and in Haskell, programmers must supply the intended polymorphic type; correspondingly, a lambda-lifter should then supply the types of lifted parameters, when they are polymorphic.

6 Lambda-lifting in Java

The Java programming language supports block structure and lexical scope in the form of inner classes [29]. The Java virtual machine on the other hand only supports scope-insensitive programs [31]. For this reason, the compilation process from Java source code to Java bytecode must make inner classes scope insensitive. We observe that part of this process is based on lambda-lifting.

6.1 Java inner classes

In Java, an inner class can be declared as a member of an enclosing class or as a local declaration within a method. The free variables of an inner class can be divided into two categories: variables that are declared as a field of some enclosing class and variables that are declared locally to an enclosing method.

The fields of an enclosing class are accessed using a static link. Specifically, the program is transformed by the compiler to access the free field variables from the enclosing class using a static link stored in a field. The static link is initialized by the constructor when instances of the inner class are created. A special classfile attribute is added to both the enclosing class and the inner class to allow the inner class to access private members of the enclosing class.

The Java language specification states that local variables which are accessed by an inner class must be declared `final`, i.e., immutable once they are initialized [29, §8.1.2]. Therefore their denotation can be safely copied. And indeed, variables that are declared locally to an enclosing method are accessed by copying the value they denote when an instance of the inner class is created. Specifically, the program is transformed to access the values of the free local variables from the immediately enclosing method through local copies stored in fields of the inner class, and to access the values of the free local variables from the outer enclosing methods through the static link. The values of the local variables are passed as constructor arguments when instances of the inner class are created.

As a net effect of the transformation, the inner classes are scope insensitive and the compiler can lift them.

6.2 A simple example of inner classes

Figure 12 illustrates inner classes declared within methods. The program is written in a functional style of programming using objects as closures, and is essentially equivalent to the ML program shown in Figure 13. The interface `Function` describes objects that represent functions that map an integer to an integer. The class `Make_fn` has a method `make_fn` which returns a `Function` object created using the two inner classes `Add_x` and `Add_x_Add_y`. The inner class `Add_x` has `x` as a free variable, whereas `Add_x_Add_y` has `y` as a free variable. A use of this class could be:

```
Function f = (new Make_fn()).make_fn(1,2);
int result = f.apply(3);
```

The effect is that `result` is assigned the value 6.

The compiled version of the program of Figure 12 is shown in Figure 14, after decompilation (for our purposes, Java byte code and Java source code contain the same information, so for readability we use decompiled Java source code). The class `Add_x` (compiled name: `Make_fn$1Add_x`) now takes the enclosing class and the variable `x` as additional constructor parameters and stores them in the fields `this$0` and `val$x`. Similarly, the class `Add_y_Add_x` takes the enclosing class and the variable `y` as additional constructor parameters and stores them in fields. However, since it also needs to create an instance of the class `Add_x`, it needs the value of `x`, and is therefore

```
public interface Function {
  public int apply(int i);
}

public class Make_fn {
  Function make_fn(final int x, final int y) {

    class Add_x implements Function {
      public int apply(int i) {
        return i+x;
      }
    }

    class Add_y_Add_x implements Function {
      public int apply(int i) {
        return (new Add_x()).apply(i)+y;
      }
    }

    return new Add_y_Add_x();
  }
}
```

Figure 12: Inner classes with free variables in Java

```
fun make_fn (x, y)
    = let fun add_x i
              = i+x
          fun add_x_add_y i
              = (add_x i) + y
      in add_x_add_y
      end
```

Figure 13: ML counterpart of Figure 12

```
public interface Function {
  public int apply(int i);
}

public class Make_fn {
  Function make_fn(final int x, final int y) {
    return new A$1Add_y_Add_x(this,x,y);
  }
}

class Make_fn$1Add_x implements Function {
  private final Make_fn this$0;
  private final int val$x;

  public Make_fn$1Add_x(Make_fn a, int x) {
    this$0=a; val$x=x;
  }

  public int apply(int i) {
    return i+$val$x;
  }
}

class Make_fn$1Add_y_Add_x implements Function {
  private final Make_fn this$0;
  private final int val$x;
  private final int val$y;

  public Make_fn$1Add_y_Add_x(Make_fn a, int x, int y) {
    this$0=a; val$x=x; val$y=y;
  }

  public int apply(int i) {
    (new Make_fn$1Add_x(this$0,val$x)).apply(i)+val$y;
  }
}
```

Figure 14: The program of Figure 12, after compilation and decompilation

```
fun make_fn (x, y)
    = let fun add_x x i
              = i+x
          fun add_x_add_y (x, y) i
              = (add_x x i) + y
      in add_x_add_y (x, y)
      end
```

Figure 15: ML counterpart of Figure 14

also passed `x` as an additional constructor parameter which is stored in a field.

The ML counterpart of Figure 14 is displayed in Figure 15. It is the parameter-lifted version of Figure 13.

For local variables occurring free in inner classes, the transformation from Java source code to Java byte code therefore coincides with lambda-lifting, since the free variables are passed as additional arguments to the constructor. Moreover, as illustrated by the example, the free variables of other inner classes that are instantiated within the inner class are also passed as constructor arguments, in a transitive fashion, again similarly to parameter lifting.

6.3 Time complexity

In actuality, lambda-lifting in Java is simpler than lambda-lifting in a functional language because inner classes defined within methods only have forward visibility. In the absence of mutual recursion, the dependencies between inner classes always form a directed acyclic graph. Therefore the propagation of free variables can be done in quadratic time, as in our lambda-lifting algorithm.

Should Java be revised one day to allow mutually recursive inner classes defined within methods, Java compilers would need to perform full lambda-lifting. It is our point here that they could do so in quadratic time rather than in cubic time.

7 Conclusion

We have shown that a transitive closure is not needed for lambda-lifting. We have reformulated lambda-lifting as a graph algorithm and improved its time complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$, where n is the size of the program. Based on a simple example where lambda-lifting generates a program of size $\Omega(n^2)$, we have also demonstrated that our improved complexity is optimal.

The quadratic-time algorithm can replace the cubic-time instances of lambda-lifting in any compiler, partial evaluator, or program transformer, be it for global or for incremental lambda-lifting.

Acknowledgements: We are grateful to Mads Sig Ager, Lars R. Clausen, Daniel Damian, Kristoffer Arnsfelt Hansen, Julia Lawall, Jan Midtgaard, Kevin S. Millikin, Laurent Réveillère, Henning Korsholm Rohde, and Kristian Støvring Sørensen for their comments on an earlier version of this article, and to Andrzej Filinski for a clarification about ML typing. Thanks are also due to the anonymous referees for very perceptive and useful reviews.

This work is supported by the ESPRIT Working Group APPSEM (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-03-0545.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. World Student Series. Addison-Wesley, Reading, Massachusetts, 1986.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[3] Andrew W. Appel. *Modern Compiler Implementation in {C, Java, ML}*. Cambridge University Press, New York, 1998.

[4] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, April 1998.

[5] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In Michael J. O'Donnell and Stuart Feldman, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of*

Programming Languages, pages 293–302, Austin, Texas, January 1989. ACM Press.

[6] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Małuszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.

[7] Lennart Augustsson. A compiler for Lazy ML. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, Texas, August 1984. ACM Press.

[8] Adam Bakewell and Colin Runciman. Automatic generalisation of function definitions. In Middeldorp and Sato [33], pages 225–240.

[9] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.

[10] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[11] Rod M. Burstall and Robin J. Popplestone. POP-2 reference manual. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, volume 5, pages 207–246. Edinburgh University Press, 1968. `http://www-robotics.cs.umass.edu/~pop/functional.html`.

[12] William Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, pages 128–139, Orlando, Florida, June 1994. ACM Press.

[13] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, Copenhagen, Denmark, June 1993. ACM Press.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, second edition, 2001.

[15] Olivier Danvy. An extensional characterization of lambda-lifting and lambda-dropping. In Middeldorp and Sato [33], pages 241–250.

[16] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000.

[17] Olivier Danvy and Ulrik P. Schultz. Lambda-lifting in quadratic time. In Zhenjiang Hu and Mario Rodriguez-Artalejo, editors, *Sixth International Symposium on Functional and Logic Programming*, number 2441 in Lecture Notes in Computer Science, pages 134–151, Aizu, Japan, September 2002. Springer-Verlag. Extended version available as the technical report BRICS-RS-03-26.

[18] Gilles Dowek. Lambda-calculus, combinators and the comprehension scheme. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications*, number 902 in Lecture Notes in Computer Science, pages 154–170, Edinburgh, UK, April 1995. Springer-Verlag. Extended version available as the INRIA research report 2535.

[19] Kerstin I. Eder. *EMA: Implementing the Rewriting Computational Model of Escher*. PhD thesis, Department of Computer Science, University of Bristol, Bristol, UK, November 1998.

[20] Adam Fischbach and John Hannan. Specification and correctness of lambda lifting. *Journal of Functional Programming*, 13(3):509–543, 2003.

[21] Arne John Glenstrup. Terminator II: Stopping partial evaluation of fully recursive programs. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, June 1999.

[22] Paul T. Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. In Martin S. Feather and Michael Goedicke, editors, *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*,

pages 211–222, Coronado Island, San Diego, California, USA, November 2001. IEEE Computer Society.

[23] Michael Hanus (ed.). Curry: An integrated functional logic language (version 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.

[24] Michael Hanus (ed.). PAKCS 1.6.0 the Portland Aachen Kiel Curry system user manual. Available at `http://www.informatik.uni-kiel.de/~pakcs`, 2004.

[25] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[26] John Hughes. Super combinators: A new implementation method for applicative languages. In Daniel P. Friedman and David S. Wise, editors, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, Pennsylvania, August 1982. ACM Press.

[27] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, September 1985. Springer-Verlag.

[28] Thomas Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[29] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, 2nd edition, 2000.

[30] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[31] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[32] Karoline Malmkjær, Nevin Heintze, and Olivier Danvy. ML partial evaluation using set-based analysis. In John Reppy, editor, *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Rapport de recherche N° 2265, INRIA*, pages 112–119, Orlando, Florida, June 1994.

[33] Aart Middeldorp and Taisuke Sato, editors. *Fourth Fuji International Symposium on Functional and Logic Programming*, number 1722 in Lecture Notes in Computer Science, Tsukuba, Japan, November 1999. Springer-Verlag.

[34] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[35] Flemming Nielson and Hanne Riis Nielson. 2-level λ-lifting. In Harald Ganzinger, editor, *Proceedings of the Second European Symposium on Programming*, number 300 in Lecture Notes in Computer Science, pages 328–343, Nancy, France, March 1988. Springer-Verlag.

[36] Atsushi Ohori. The logical abstract machine: A Curry-Howard isomorphism for machine code. In Middeldorp and Sato [33], pages 300–318.

[37] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1/2):111–182, 1995.

[38] Simon L. Peyton Jones. An introduction to fully-lazy supercombinators. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, number 242 in Lecture Notes in Computer Science, pages 176–208, Val d'Ajol, France, 1985. Springer-Verlag.

[39] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987.

[40] John Reppy. Optimizing nested loops using local CPS conversion. *Higher-Order and Symbolic Computation*, 15(2/3):161–180, 2002.

[41] André Santos. *Compilation by transformation in non-strict functional languages*. PhD thesis, Department of Computing, University of Glasgow, Glasgow, Scotland, 1996.

[42] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.

[43] Peter Thiemann. ML-style typing, lambda lifting, and partial evaluation. In *Proceedings of the 1999 Latin-American Conference on Functional Programming, CLAPF '99*, Recife, Pernambuco, Brasil, March 1999.

[44] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer et al., editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2004. (To appear). See `http://www.stratego-language.org/Stratego/LiftDefinitionsToTopLevel` for a discussion of lambda-lifting in the Stratego Compiler.

[45] Mitchell Wand. From interpreter to compiler: a representational derivation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, pages 306–324, Copenhagen, Denmark, October 1985. Springer-Verlag.