Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure

Olivier Danvy and Ulrik P. Schultz

BRICS *

Department of Computer Science
University of Aarhus †

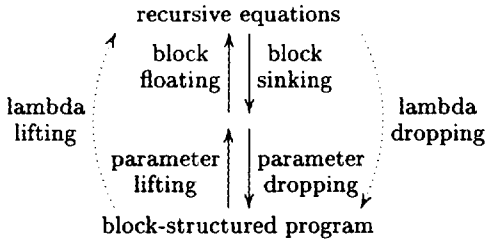
Abstract

Lambda-lifting a functional program transforms it into a set of recursive equations. We present the symmetric transformation: lambda-dropping. Lambda-dropping a set of recursive equations restores block structure and lexical scope.

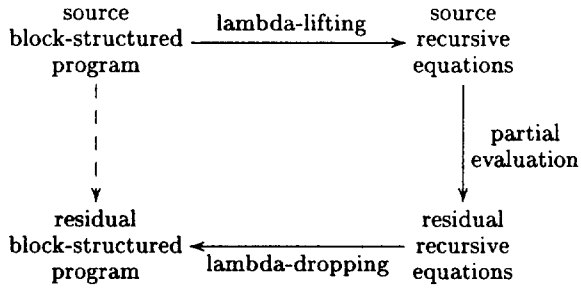
For lack of scope, recursive equations must carry around all the parameters that any of their callees might possibly need. Both lambda-lifting and lambda-dropping thus require one to compute a transitive closure over the call graph:

- for lambda-lifting: to establish the Def/Use path of each free variable (these free variables are then added as parameters to each of the functions in the call path);
- for lambda-dropping: to establish the Def/Use path of each parameter (parameters whose use occurs in the same scope as their definition do not need to be passed along in the call path).

Without free variables, a program is scope-insensitive. Its blocks are then free to float (for lambda-lifting) or to sink (for lambda-dropping) along the vertices of the scope tree.



We believe lambda-lifting and lambda-dropping are interesting per se, both in principle and in practice, but our prime application is partial evaluation: except for Malmkjær and Ørbæk's case study presented at PEPM'95, most polyvariant specializers for procedural programs operate on recursive equations. To this end, in a pre-processing phase, they lambda-lift source programs into recursive equations. As a result, residual programs are also expressed as recursive equations, often with dozens of parameters, which most compilers do not handle efficiently. Lambda-dropping in a post-processing phase restores their block structure and lexical scope thereby significantly reducing both the compile time and the run time of residual programs.



1 Introduction and Motivation

Block structure and lexical scope stand at the foundation of functional programming, but are they so much in everyday use?

Evidence says that they are not. Consider the standard append function defined as a recursive equation:

```
fun append-lifted (nil, ys) = ys
  | append-lifted (x :: xs, ys) =
                  x :: (append-lifted (xs, ys))
```

Using block structure and lexical scope, append could have been defined as follows:

```
fun append-dropped (xs, ys) =
    let fun loop nil = ys
          | loop (x :: xs) = x :: (loop xs)
    in loop xs end
```

In the lifted version, the second argument is passed during the whole traversal of the first argument, only to be used in the base case. In the dropped version, the second argument is free in the traversal of the first argument. The dropped version follows more closely the inductive definition of lists. It does so by exploiting two linguistic features: scope and block structure.

*Basic Research in Computer Science,
Centre of the Danish National Research Foundation.
†Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
Phone: (+45) 89 42 33 69. Fax: (+45) 89 42 32 55.
E-mail: {danvy,ups}@brics.dk
Home pages: http://www.brics.dk/~{danvy,ups}

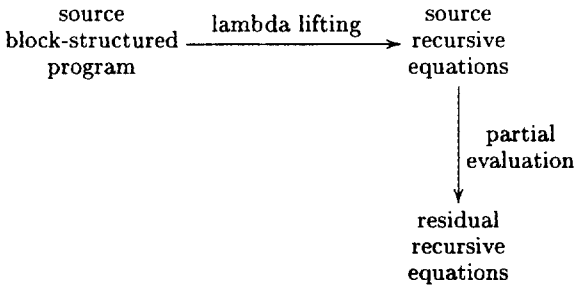
PEPM '97 Amsterdam, ND

This example might appear overly simple, but there are many others — map for example: the mapped function is passed as a parameter during the whole traversal of the list. Fold functions over lists pass *two* unchanging parameters (the folded function and the initial value of the accumulator) during the traversal of the list. Lambda-interpreters thread the environment through every syntactic form instead of keeping it as a global variable and making an appropriate recursive call when encountering a binding form.

These examples are symptomatic of a programming malaise: recursive equations offer no linguistic support to write modular programs, and they suffer from a chronic inflation of parameters.[1] Auxiliary functions must be written as extra recursive equations. This makes it possible to call them directly with non-sensical initial values, e.g., for accumulators. Instead, auxiliary functions should be local — a programming style that is not possible with recursive equations.

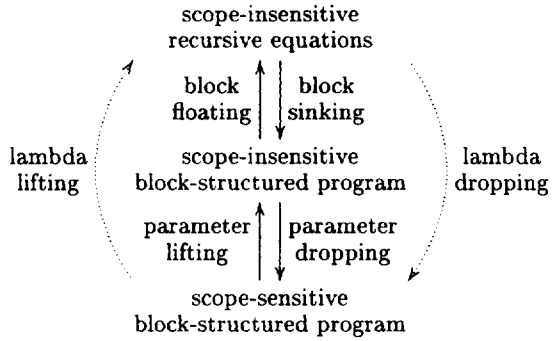
Yet recursive equations can be efficiently implemented. This led Hughes, Johnsson, and Peyton Jones, in the mid 80's, to devise a meaning-preserving transformation from block-structured programs to recursive equations: lambda-lifting [18, 19, 28]. We review lambda-lifting in Section 2.

Recursive equations also offer a convenient format in Mix-style partial evaluation [22]. Modern partial evaluators such as Schism and Similix lambda-lift source programs before specialization [8, 10].



As a result, residual programs are also expressed as recursive equations. If partial evaluation is to be seen as a source-to-source program transformation, however, residual programs should be block structured. To this end, we present lambda-dropping: the transformation of recursive equations into block-structured and lexically scoped programs.

Overview: The rest of this article is organized as follows. Sections 2 to 5 describe lambda-lifting and lambda-dropping. We present them in a symmetric way:



Parameter lifting makes a program scope-insensitive by passing extra variables to each function to account for variables occurring free further down the call path. Block floating eliminates block structure by globalizing each block, making each of its locally defined functions a global recursive equation. Block sinking restores block structure by localizing (strongly connected) groups of equations in the call graph. Parameter dropping exploits scope by not passing variables whose end use occurs in the scope of their initial definition.

Section 6 investigates applications of lambda-lifting and lambda-dropping. Section 7 surveys related work. Section 8 concludes.

Throughout the article, we assume variable hygiene, i.e., that no name clashes can occur. Also, the term "block" is assumed to mean a collection of declarations, possibly functions, followed by a main expression.

2 Lambda-Lifting

We consider Johnsson's algorithm [3, 19, 20]. Johnsson's target is the G-machine, which can run recursive equations efficiently. Lambda-lifting transforms a block-structured program into a set of recursive equations. These recursive equations correspond to local functions in the block-structured program. Each equation is passed variables that would have occurred free in a block-structured program.

2.1 The basics of lambda-lifting

Lambda-lifting is achieved using two transformations that are applied iteratively:

1. *Eta-expansion.* Each lambda-abstraction is eta-expanded with its free variables, both at its definition site and at all its application sites.

2. *Letrec floating.* Each set of local functions that do not refer to local free variables is moved to the enclosing scope level.

Eta-expanding an application can introduce more free variables, which are handled in a later iteration. Function names do not need to be passed as parameters, since they are used as the names of the recursive equations and thus are globally visible.

The two transformations listed above can be freely interleaved. Johnsson's algorithm, however, factorizes the transformations into two stages.

2.2 Implementing lambda-lifting

In a program with proper variable hygiene, a let expression can be replaced by a letrec expression. Thus, we assume that the program being lambda-lifted uses letrec expressions for both function and value bindings. Johnsson's algorithm proceeds in two stages.

1. *Parameter lifting.*

   Free variables are eliminated by eta-expanding each local function definition. Each time the definition of

[1] As Alan J. Perlis's epigram goes, "if you have a procedure with ten parameters, you probably missed some."

1. Naming anonymous lambda abstractions.

   Each $(\lambda x.e)$ is replaced by (letrec $f = \lambda x.e$ in f), where f is fresh.

2. Lifting the parameters of each function, by traversing the program top-down.

   During the traversal we use a set to describe the information that is gathered. This set associates the name of a function with the set of variables it needs to be eta-expanded with. We call this set the "set of solutions."

   - We process each occurrence of a function name according to the set of solutions. If the function is associated with the empty set, no change is made. If a function is associated with a non-empty set of variables, the function is eta-expanded into an application to these variables.
   - We process each letrec block by computing a new set of solutions describing the functions.
     (a) To describe the occurrences of variables and functions in each local function, we create a group of recursive set equations. We associate a local function g with:
       - A set V_g holding the free variables of g. These variables are needed by the function.
       - A set F_g holding all function names (the callees) occurring in the body of g. The lifted definition of g must provide variables for all the callees.

     (b) First we process the occurrences of functions not declared in the block. In each set F_g, each function h which is described in the set of known solutions S, is removed from F_g. The set V_g is extended with the variables associated with h in S.

     (c) Then we process each function h remaining in F_g by adding all elements of V_h to V_g. We repeat this step until a fixed point is reached.

     (d) The new set of solutions is the union of the old solutions with the set equations.

   We traverse the body and each binding of the letrec block with the newly computed set of solutions.

Figure 1: Parameter lifting. Free variables are made parameters.

1. Moving each block to the global level.

   We process each letrec block by removing all of its function definitions and making them global. If no local declarations remain, we replace the block by its body.

2. Removing the remaining letrec blocks.

   We remove the remaining (now non-recursive) letrec blocks by converting each one into an application of a global function that has a fresh name. The formal parameters of the global function are the variables that were bound by the letrec block, and the body of the function is the body of the block.

Figure 2: Block floating. Flattening of block structure.

a function is eta-expanded, all call sites of this function must be correspondingly eta-expanded. In the resulting program, no function has free variables. This process is detailed in Figure 1.

2. *Block floating.*

   The floating step is trivial because each function is scope-insensitive after parameter lifting. Definitions can freely be moved outwards through the block structure of the program. In the resulting program, all function definitions are global. This process is detailed in Figure 2.

Other styles and implementations of lambda-lifting exist. We review them in Section 7.2.

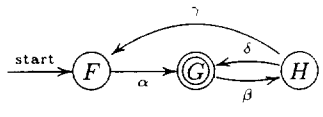
2.3 A detailed example

As an example, we consider an implementation of a Deterministic Finite Automata (DFA), as displayed in Figure 3. The DFA accepts the language defined by the regular expression $R = \alpha(\beta(\delta|\gamma\alpha))^*$, given the alphabet $\{\alpha, \beta, \gamma, \delta\}$. The DFA reads symbols from an input stream, and invokes the associated functions `a`, `b`, `c` and `d`, as long as the input symbols conform to R. If the input is non-conforming, the DFA aborts. The formal parameter `die?` determines whether to signal an error or just return the empty list. The stream is terminated by either the symbol \$ or the end of the list.

The next two sections describe the process of lambda-lifting this program according to the algorithm of Figures 1 and 2. Some of the variables share the same name. This lack of hygiene does not interfere with the algorithm here, so we retain the names for the sake of clarity.

2.3.1 Parameter lifting

1. The anonymous lambda-abstraction within the body of `f` needs to be explicitly named:




```
(define (r a b c d die? xs)
  (letrec ([err
            (lambda (x)
              (if (null? x)
                  (error 'r "end of stream~%")
                  (error 'r "token ~a~%" x)))]
           [empty?
            (lambda (s)
              (or (null? s) (eq? (car s) '$)))]
           [h (lambda (reject xs)
                (if (empty? xs)
                    (reject '())
                    (case (car xs)
                      [(gamma) (c (f reject (cdr xs)))]
                      [(delta) (d (g reject (cdr xs)))]
                      [else (reject (car xs))])))]
           [g (lambda (reject xs)
                (if (empty? xs)
                    xs
                    (case (car xs)
                      [(beta) (b (h reject (cdr xs)))]
                      [else (reject (car xs))])))]
           [f (lambda (reject xs)
                (if (empty? xs)
                    (reject '())
                    (case (car xs)
                      [(alpha) (a (g reject (cdr xs)))]
                      [else (reject (car xs))])))])
    (f (if die? err (lambda (x) '())) xs)))
```

Figure 3: Our original DFA.

```
(define (r a b c d die? xs)
  ((f a b c d) (if die? err fresh0) xs))
(define (fresh0 x)
  x)
(define (err x)
  (if (null? x)
      (error 'r "unexpected end of stream~%")
      (error 'r "unexpected token ~a~%" x)))
(define (empty? s)
  (or (null? s) (eq? (car s) '$)))
(define (h a b c d)
  (lambda (reject xs)
    (if (empty? xs)
        (reject '())
        (case (car xs)
          [(gamma) (c ((f a b c d) reject (cdr xs)))]
          [(delta) (d ((g a b c d) reject (cdr xs)))]
          [else (reject (car xs))]))))
(define (g a b c d)
  (lambda (reject xs)
    (if (empty? xs)
        '()
        (case (car xs)
          [(beta) (b ((h a b c d) reject (cdr xs)))]
          [else (reject (car xs))]))))
(define (f a b c d)
  (lambda (reject xs)
    (if (empty? xs)
        (reject '())
        (case (car xs)
          [(alpha) (a ((g a b c d) reject (cdr xs)))]
          [else (reject (car xs))]))))
```

Figure 4: The DFA after lambda-lifting.

```
(f (if die?
       r-error
       (lambda (x) '()))
 xs)
```

becomes

```
(f (if die?
       r-error
       (letrec ([fresh0 (lambda (x) '())])
         fresh0))
 xs)
```

where `fresh0` is a fresh variable.

2. The entire program is traversed. At all times S represents the current set of solutions.

   `(define (r ...) (letrec ...))` with $S = \emptyset$:

   A global function contains no free variables. Thus, its parameters do not need any lifting.

   `(letrec ...)` with $S = \emptyset$:

   We process a block by extending the set of solutions. We start by computing the sets of free variables for each function:

$$\begin{aligned} V_{\mathbf{err}} &= V_{\mathbf{empty?}} = \emptyset \\ V_h &= \{c, d\} \\ V_g &= \{b\} \\ V_f &= \{a\} \end{aligned}$$

Then we compute the sets of function names describing references to other functions:

$$\begin{aligned} F_{\mathbf{err}} &= F_{\mathbf{empty?}} = \emptyset \\ F_h &= \{f, g, \mathbf{empty?}\} \\ F_g &= \{h, \mathbf{empty?}\} \\ F_f &= \{g, \mathbf{empty?}\} \end{aligned}$$

These sets are used to express the set equations:

$$\begin{aligned} V_{\mathbf{err}} &:= V_{\mathbf{err}} = \emptyset \\ V_{\mathbf{empty?}} &:= V_{\mathbf{empty?}} = \emptyset \\ V_h &:= V_h \cup V_f \cup V_g \cup V_{\mathbf{empty?}} = \{c, d\} \\ V_g &:= V_g \cup V_h \cup V_{\mathbf{empty?}} = \{b\} \\ V_f &:= V_f \cup V_g \cup V_{\mathbf{empty?}} = \{a\} \end{aligned}$$

To solve these set equations, we iterate the assignments until a fixed point is reached.

Iteration 1:

$$\begin{aligned} V_{\mathbf{err}} &:= \emptyset \\ V_{\mathbf{empty?}} &:= \emptyset \\ V_h &:= \{c, d\} \cup \{a\} \cup \{b\} \cup \emptyset = \{a, b, c, d\} \\ V_g &:= \{b\} \cup \{c, d\} \cup \emptyset = \{b, c, d\} \\ V_f &:= \{a\} \cup \{b\} \cup \emptyset = \{a, b\} \end{aligned}$$

Iteration 2:

$$\begin{aligned} V_{\texttt{err}} &:= \emptyset \\ V_{\texttt{empty?}} &:= \emptyset \\ V_h &:= \{a,b,c,d\} \cup \{a,b\} \cup \{b,c,d\} \cup \emptyset \\ &= \{a,b,c,d\} \\ V_g &:= \{b,c,d\} \cup \{a,b,c,d\} \cup \emptyset = \{a,b,c,d\} \\ V_f &:= \{a,b\} \cup \{b,c,d\} \cup \emptyset = \{a,b,c,d\} \end{aligned}$$

Iteration 3:

$$\begin{aligned} V_{\texttt{err}} &:= \emptyset \\ V_{\texttt{empty?}} &:= \emptyset \\ V_h &:= \{a,b,c,d\} \cup \{a,b,c,d\} \cup \{a,b,c,d\} \\ &\quad \cup \emptyset \\ &= \{a,b,c,d\} \\ V_g &:= \{a,b,c,d\} \cup \{a,b,c,d\} \cup \emptyset \\ &= \{a,b,c,d\} \\ V_f &:= \{a,b,c,d\} \cup \{a,b,c,d\} \cup \emptyset \\ &= \{a,b,c,d\} \end{aligned}$$

We have reached a fixed point. The new set equations are:

$$\begin{aligned} V_{\texttt{err}} &= \emptyset \\ V_{\texttt{empty?}} &= \emptyset \\ V_h &= V_g = V_f = \{a,b,c,d\} \end{aligned}$$

We can now process the local functions and the body of the letrec block, using the new set equations.

`(letrec ([err ...] ...) ...)` :
$S = \{r = \emptyset, \ldots, h = \{a,b,c,d\}, \ldots\}$.
S associates `err` with the empty set, so there is no need to eta-expand the definition of `err`. The body contains no functions described in the set of solutions, so no changes need to be made here either.

`(letrec (... [empty? ...]) ...)` is handled like `err`.

`(letrec (... [h ...]) ...)` is handled as follows:
$S = \{r = \emptyset, \ldots, h = \{a,b,c,d\}, \ldots\}$.
The solutions associate `h` with the set $\{a,b,c,d\}$. Thus, we must eta-expand the definition:

```
[h (lambda (reject xs) ...
```

is replaced by

```
[h (lambda (a b c d) (lambda (reject xs) ...
```

The body of `h` contains references to `empty?` and `g`. The solutions associate `empty?` with the empty set, so it is unchanged. The solutions also associate `g` with the set $\{a,b,c,d\}$. The occurrence of `g` as

```
(g reject (cdr xs))
```

is replaced by

```
((g a b c d) reject (cdr xs))
```

`(letrec (...[g ...]...) ...)` is handled like `h`.

`(letrec (...[f ...]...) ...)` is handled like `h`.

`(letrec (...) (f ...))` is handled as follows:
$S = \{r = \emptyset, \ldots, h = \{a,b,c,d\}, \ldots\}$.
We eta-expand the occurrence of `f` with the appropriate variables. The declaration and occurrence of `fresh0` need not be eta-expanded.

2.3.2 Function floating

1. We remove all functions from the letrec block declaring `f` to the outermost (global) one, and we replace this block by its body. Ditto for the block declaring `fresh0`.
2. Since no letrec blocks are left, there is no need to convert them into applications.

Figure 4 displays the final result.

3 Reversing Lambda-Lifting

As described in Section 2, lambda-lifting starts by making functions scope-insensitive through eta-expansion, and then proceeds to make all functions global through letrec-floating. To reverse lambda-lifting, we could make the appropriate global functions local, and then make them scope-sensitive through eta-reduction. To simplify the process, we always generate letrec blocks, even if a let block would suffice.

Localizing a function in a letrec block moves it into the context where it is used. Once a function is localized, it is no longer visible outside the letrec block. This localization often makes the program easier to understand for a human reader, and simplifies compilation.[2] Localization, however, is not always possible. Functions used in different parts of the program might not be localizable to all these places, unless one is willing to duplicate code.[3]

3.1 Block sinking

To reverse the effect of lambda-lifting, let us examine the program of Figure 4, which was lambda-lifted in Section 2.3. The main function of the program is `r`. All other functions are used by `r`, and are thus localizable to `r`. We replace the body of `r` with a block declaring these functions and having the original body of `r` as its body.

```
define r = letrec f = ...
               g = ...
               h = ...
               err = ...
               empty? = ...
               fresh0 = ...
           in ...
```

[2] "Simplify" in the sense that often, efficient compilers are tuned to typical handwritten programs. For example, Chez Scheme and Standard ML of New Jersey handle functions with few parameters better than functions with many parameters. This is bad news for partial evaluation: for example, a partial evaluator such as Pell-Mell [14], because it uses lightweight symbolic values, tends to produce recursive equations with dozens of parameters. Compiling these residual programs de facto becomes a bottleneck.

[3] The Glasgow Haskell compiler is currently the theater of intensive and so far encouraging experiments in block floating and block sinking [26]. In that work, floating is only relative in that blocks are moved either inwards or outwards, and in the latter case, not necessarily at the top level. Source programs, however, are no longer systematically lambda-lifted.

```
(define (r a b c d die? xs)
  (letrec ([f (lambda (reject xs)
                (letrec ([g (lambda (reject xs)
                              (letrec ([h (lambda (reject xs)
                                           (if (empty? xs)
                                               (reject '())
                                               (case (car xs)
                                                 [(gamma) (c (f reject (cdr xs)))]
                                                 [(delta) (d (g reject (cdr xs)))]
                                                 [else (reject (car xs))])))])
                                (if (empty? xs)
                                    xs
                                    (case (car xs)
                                      [(beta) (b (h reject (cdr xs)))]
                                      [else (reject (car xs))])))))]
                         [empty?
                          (lambda (s)
                            (or (null? s) (eq? (car s) '$)))])
                  (if (empty? xs)
                      (reject '())
                      (case (car xs)
                        [(alpha) (a (g reject (cdr xs)))]
                        [else (reject (car xs))])))))]
           [fresh0
            (lambda (x) x)]
           [err
            (lambda (x)
              (if (null? x)
                  (error 'r "unexpected end of stream~%")
                  (error 'r "unexpected token ~a~%" x)))])
    (f (if die? err (lambda (x) '())) xs)))
```

Figure 5: The DFA program after lambda-dropping.

We can see that the body of `r` refers to only the functions `f`, `err` and `fresh0`. The functions `g`, `h` and `empty?`, however, are used only by `f`. Therefore it makes sense to localize them to `f`.

```
define r = letrec f = letrec g = ...
                              h = ...
                              empty? = ...
                          in ...
                   err = ...
                   fresh0 = ...
           in ...
```

Examining the newly localized functions more closely reveals that `h` is only called by `g`. It is therefore possible to localize `h` in `g`. The function `empty?` is used by `f`, `g` and `h`, but we cannot localize it any further, since it needs to be accessible to `f`. The result of our function localization is:

```
define r = letrec f = letrec g = letrec h = ...
                                     in ...
                              empty? = ...
                          in ...
                   err = ...
                   fresh0 = ...
           in ...
```

The functions of the program cannot be localized any further. This lambda-dropped version has more block structure than the original version. It is our experience that one tends to write such incompletely lambda-dropped programs.

3.2 Parameter dropping

To reverse the parameter lifting performed during lambda-lifting, we need to determine the origins of each formal parameter. The functions `f`, `g` and `h` all pass the variables `a`, `b`, `c` and `d` to each other. These formal parameters always correspond to the variables of the same name defined by the function `r`. Since these parameters all are now visible where the three functions are declared, there is no need to pass them around as parameters. We can simply remove the four formal parameters from the declaration of each function, and refrain from passing them as arguments at each application site.

The function `err` is passed as an argument to `f`. We would thus need to perform a control-flow analysis to determine which arguments are passed to `err` (in this case a control-flow analysis is very simple). If parameter lifting had been performed on `err`, `err` would have been applied directly to any such variables. Since `err` is not applied, we can safely assume that it has not been parameter lifted. The function `empty?` is never passed as an argument, but it is passed arguments that are not themselves formal variables. These arguments could not have been the produced by the lambda-lifter, so we need not consider dropping the parameters of this function either.

We have thus removed all the formal parameters of the functions `f`, `g` and `h`. What remains in each case is a function of no parameters which returns a function of two parameters — the two parameters of the function declaration from the original program. Since these thunks are always applied, it makes sense to eliminate them, leaving only the part of the function corresponding to the original program.

Figure 5 displays the final result of our reversal process. Comparing with the original program, and taking into account that we generate more block structure, a single difference remains: the anonymous lambda-abstraction from the body of `r` is still explicitly named.

4 Lambda-Dropping

We now specify lambda-dropping more formally. Lambda-dropping a program minimises parameter passing, and serves in principle as an inverse of lambda-lifting. Function definitions are localised maximally using lexically scoped block structure. Parameters made redundant by the newly created scope are eliminated.

4.1 The basics of lambda-dropping

Lambda-dropping is achieved using two transformations that are iteratively applied:

1. *Letrec sinking.* Any set of functions that is referenced by a single function only is made local to this function. This is achieved by sinking this set inside the definition of the function.
2. *Eta reduction.* A function with a formal parameter that is bound to the same variable in every invocation can potentially be parameter-dropped. If the variable is lexically visible at the definition of the function, the formal parameter can actually be dropped. A formal parameter is dropped from a function by removing the formal parameter from the parameter list, removing the corresponding argument from all invocations of the function, and substituting the name of the variable for the name of the formal parameter throughout the body of the function.

4.2 Implementing lambda-dropping

To ensure scope insensitivity of functions, we start by lambda-lifting the source program. If the program contains global function declarations only, the lambda-lifting step is not necessary. The lambda-dropping algorithm then works in two stages. It handles other binding constructs (such as let or the letType construct of Schism [10]), but we have made no attempt to drop parameters which are bound to variables defined by these constructs.

1. *Block sinking.*

   Each reference to a function introduces restrictions on where it can be declared. Expressing these restrictions as a tree gives guidelines for translating from recursive equations to block structure. In the resulting program, function definitions are declared locally wherever possible. This process is detailed in Figure 6.

2. *Parameter dropping.*

   Two things determine whether some of the parameters of a function can be dropped: the scope in which the function is declared and the set of variables it is passed as arguments. In the resulting program, a formal parameter is never passed as an argument to a function if it instead could have been substituted throughout the body of the function. This process is detailed in Figure 7.

1. Creating a scope graph.

   We construct a graph describing the scope constraints imposed by references to global functions:

   - For each function definition, we create a node.
   - If a function f refers to some other function g by its name, we introduce an edge from the node of f to the node of g.
   - We create an empty node, the root node. This root node will point to the node of any function that should remain as a global definition.

   During this pass, we trivially create Def/Use chains for functions. We also tag any function passed as an argument as "higher order."

2. Transforming the scope graph into a DAG.

   We isolate the strongly connected components of the scope graph. We collapse components that contain at most a single cycle, into a node. We associate the resulting node with the set of functions represented by the nodes of the component. If the component only has a single entry point and contains more than one node, we try to reduce the size of the component. We remove edges from nodes in the component to the node that is the entry point of the component. We iterate the transformation step on the nodes of the component.

3. Transforming the DAG into a tree.

   - Two disjoint paths p_1 and p_2 from a node N to some other node N' indicate that a common successor exists in the graph. We must detach the common successor N' from at least one of its direct predecessors, and re-attach it without violating the scope restrictions imposed by the graph. Detaching N' from those of its predecessors that lie on p_1 and p_2, and making it a direct successor of N preserve scope restrictions. We collapse all successors of N' into a single node, which we place at the previous location of N'.
   - We merge every node associated with more than one function, and that is the father of a node, with this child node. We remove the child node from the graph, re-attaching any edges onto the father node.

4. Re-constructing the program from the tree.

   The tree representation is isomorphic to a program. We make the direct successors of the root node global functions. Then we traverse the tree, generating a function containing a letrec block at each branch. The functions associated with the direct successors of this node we declare locally in this block. If the node has no successors, we generate no letrec block.

Figure 6: Block sinking. Re-creation of block structure.

1. Determining Use/Def information for each formal parameter.

   We annotate each use of a variable declared as a formal parameter with a reference to the function that declared it.

2. Constructing a flow graph.

   We process the formal parameters of each function definition separately, except when the function is marked as "higher order." In this case, we leave it unprocessed.

   - We create a definition node for the function that declares the formal parameter.
   - We create an application node for each application site of the function where the argument corresponding to the formal parameter is itself a formal parameter.
   - If the argument corresponding to the formal parameter is not a formal parameter, we discard the definition node and all corresponding application nodes, and we create no more nodes for this parameter.

   We collapse the strongly connected components of the graph, using the standard algorithm to detect strongly connected components [1]. Reversing the resulting DAG yields the flow graph.

3. Propagating variable identities through the flow graph.

   We assign a unique ID to all nodes without predecessors. All other nodes have uninitialized IDs. When all the predecessors of a node all have the same ID, we assign this ID to this node. When some predecessors of a node have different IDs, we assign a new, unique ID to this node.

4. Removing redundant formal parameters.

   We process the parameters of each function definition separately. If the U/D chain of a formal parameter p of a function definition D points to another formal parameter p', and D is declared in the scope of p', we remove p. We remove the formal parameter p from the parameter list of the declaration of D, and we substitute p' for p throughout the body of D. If all formal parameters are removed from D, a possibly redundant thunk encapsulates the body of D. If the body is a lambda form, we substitute this lambda form for D. If the body is a letrec block with a lambda form as its body, we move the block into the lambda form, and substitute the lambda form for D. In both cases, the function is "de-thunked."

5. Removing redundant arguments.

   We process each application site by removing the arguments corresponding to formal parameters that were removed. If the function being applied was de-thunked, we replace the application by a reference to the function.

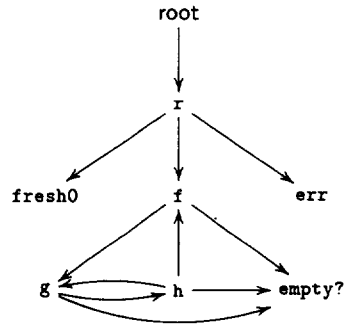
Figure 7: Parameter dropping. Removing parameters.

4.3 A detailed example

In Section 2.3, we demonstrated how the program of Figure 3 was lambda-lifted, resulting in the program displayed in Figure 4. Lambda-dropping can be applied to this program, as described in Figures 6 and 7. In this program, many variables have the same name. For the sake of clarity, we retain the original names, and annotate the variables to tell them apart when necessary.

4.3.1 Function sinking

1. We start by creating the scope graph of the program. The edges correspond to references to function names.



   During the creation of the scope graph, we also construct Def/Use chains for functions. Reversing the edges of the scope graph yields the Def/Use chains.

2. We iterate the Strongly Connected Component (SCC) algorithm on the scope graph, by removing edges and collapsing nodes into sets.

   (a) The nodes $\{h, g, f\}$ are in the same SCC.

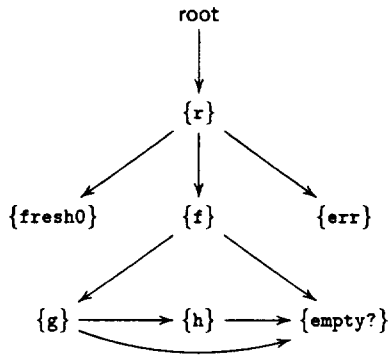
   (b) f is the entry point of this SCC (by the edge $\langle r \to f \rangle$), so $\langle h \to f \rangle$ is removed.

   (c) Now $\{h, g\}$ are in the same SCC.

   (d) g is the entry point of this SCC (by the edge $\langle f \to g \rangle$), so $\langle h \to g \rangle$ is removed.

   (e) There are no more SCC.

   The resulting DAG has the same structure as the original graph. Each node is a singleton, since recursive dependencies could be resolved by removing edges rather than collapsing nodes.

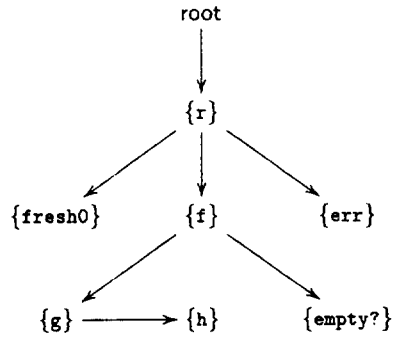


3. We transform the DAG into a tree:

- There are two disjoint paths from {f} to {empty?}: one direct from {f} itself, and one going through {g}. To remove the common successor, we remove the edges ⟨{f} → {empty?}⟩ and ⟨{g} → {empty?}⟩, and we add the edge ⟨{f} → {empty?}⟩.

  There are still two disjoint paths from f to empty?. One is direct from f and one goes through h. We remove the edges ⟨{f} → {empty?}⟩ and ⟨{h} → {empty?}⟩, and we add the edge ⟨{f} → {empty?}⟩.
- No node is associated with more than one function, so there is no need to collapse the tree.

The result of the transformation is the following tree:



4. We re-construct the program from the tree:
   - There are edges from {r} to {fresh0}, {err} and {f}. So we place the functions from these nodes in a block in r.[4]
   - There are edges from {f} to {g} and {empty?}, so we place g and empty? in a block in f.
   - There is an edge from {g} to {h}, so we place h in a block in g.

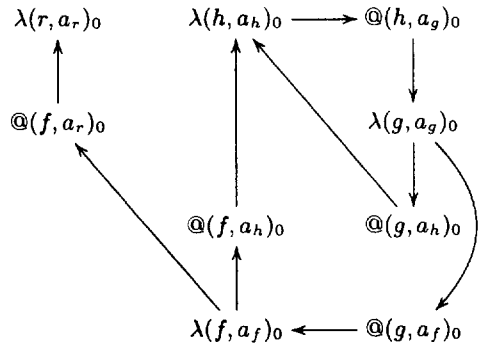
The block structure of the resulting program can be outlined as:

```
(define (r ...)
  (letrec ([f (lambda (...)
                (letrec ([g (lambda (...)
                              (letrec ([h ...])
                                ...))]
                            [empty? ...])
                  ...))]
           [fresh0 ...]
           [err ...])
    ...))
```

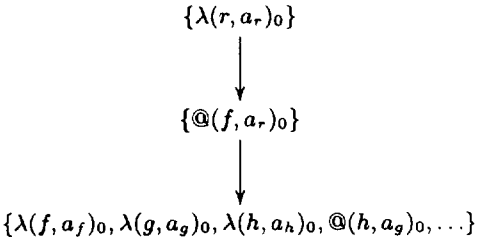
4.3.2 Dropping parameters

1. There are only function names and formal parameters in the program. For each function we let the formal parameters occurring in the body point to the definition of the function.
2. Constructing the flow graph:
   - We create appropriate nodes for all functions and all applications, with the following exceptions:
     - The function fresh0 is passed as an argument, so we create no nodes for it.
     - The function err is passed as an argument, so we create no nodes for it.
     - The function empty? receives an argument that is not a formal parameter. Thus we discard all nodes associated with this first parameter.

   In the rest of this section, we index uncurried parameters starting from zero. The resulting graph consists of four components, one for each of a, b, c and d. These components are all identical except for variable positions and parameter names. The graph for variable position zero and the parameter a looks like this:



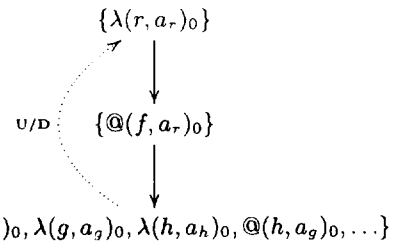
   For clarity, the formal parameter a has been annotated with the name of the function defining it, in each node. The three other components of the graph detail parameter positions one to three, with the variables b, c and d. In the general case, the graph can have connections between nodes with different variable positions.
   - All nodes for the parameter a of f, g and h are in the same SCC, and likewise for b, c and d. Thus, in the case of variable position zero and the parameter a, collapsing the SCC of each component of the flow graph yields:



     The three other components also have the same structure.
   - Reversing the graph yields the flow graph.
3. The propagation of identities through the flow graph is trivial: The roots are $\lambda(r, v)_i$, where v is a, b, c or d, and i is 0, 1, 2 or 3, respectively. Each root is assigned a unique ID. All nodes for a receive the same ID, and similarly for b, c and d.

   This yields a flow graph annotated with U/D chains for variables:

[4] If r had not been a singleton node, the structure of the tree would have been locally collapsed in the previous step.



4. Finally, we traverse the program:

   `(define (r ...) ...)` : The parameters of a top-level definition cannot be dropped.

   `(letrec (...[f (lambda (a b c d) ...)]...) ...)` :
   The U/D chain of `a` points to the variable `a` in the definition of `r`. This variable is visible in the current scope, so `a` can be parameter-dropped. Similarly for the parameters `b`, `c` and `d`.

   Dropping these four parameters leaves none behind. The body of `f` is a letrec block, the body of which is a lambda form (with the formal parameters `reject` and `xs`). We replace the declaration of `f` with this lambda form, and place the letrec block inside.

   ```
   [f (lambda (a b c d)
        (letrec (...)
          (lambda (reject xs) ...)))]
   ```

   is replaced by

   ```
   [f (lambda (reject xs) (letrec (...) ...))]
   ```

   The definitions of `g` and `h` are handled similarly.

   The functions `fresh0`, `err` and `empty?` have no U/D chains, so they cannot be dropped.

5. Every application is processed. If it is an application of `f`, `g` or `h`, the application is replaced by the function itself.

The resulting program is the same as the program we lambda-dropped by hand in Section 3. It can be found in Figure 5.

5 Further Explanation

Several aspects of the lambda-dropping transformation deserve to be explained in greater detail. They include the block-structure transformations, the flow graph and properties of lambda-dropping as a whole.

5.1 Restoring blocks

The block-structure transformations are realized through a series of graph transformations. In this section we discuss some of the intermediate graph representations, and how each of these relate to the programs they represent.

5.1.1 Scope graphs

We now compare the scope graph used by the lambda-dropping algorithm with the more familiar call graph. A graph is used as an intermediate representation during function sinking. At all times, the graph structure expresses constraints on the block structure of the program. We start out with the scope graph of the program: the scope graph has nodes for each function definition, and each edge expresses that a function needs to be able to refer to another function.

In a first-order program, the scope graph is equivalent to the call graph. In this restricted case, the only way to refer to another function is by calling it. Thus, each reference is equivalent to a call, and the graphs coincide. However, if functions are passed as arguments, all references to function names must be included in the graph. In this case, the call graph is a sub-graph of the scope graph. Note that references to a function through an alias (e.g., a variable bound to the function) do not generate an edge in neither the scope graph nor the call graph. Such a reference does not impose restrictions on how the function can be localized.

5.1.2 Refined strongly connected components

The transformation from a graph to a DAG employs an adapted version of the algorithm detecting strongly connected components in the Dragon book [1]. In Step 2 of the block-sinking stage of Figure 6, the cycles of the scope graph (the strongly connected components) are collapsed into nodes, thus isolating recursive dependencies and yielding a DAG. It turns out that the usual algorithm is too coarse for our purposes: localizable functions may not be localized if this algorithm is used in Step 2 of the block-sinking stage.

Consider the DFA program of Figure 4, for example. The scope graph, which is the first diagram of Section 4.3.1, contains two cycles, both of which belong to the same strongly connected component. The nodes `g` and `h` form a sub-cycle of the cycle containing the nodes $\{\mathtt{f}, \mathtt{g}, \mathtt{h}\}$. Performing the function localization using the usual SCC-detection algorithm would yield a program where all local definitions are declared in a single block within the body of the function `r`. The block structure would become identical to that of the source program. (The source program is shown in Figure 3).

This coincidence reflects the fact that lambda-lifting two different programs can yield the same result. Had we lambda-lifted our lambda-dropped program (in Figure 5), we would also have ended up with a program identical to the lambda-lifted program of Figure 4. This example illustrates that we cannot hope to provide a unique inverse for lambda-lifting. Block structure can be reconstructed from recursive equations in several different ways.

We have taken the approach of generating maximally localized block structure. For this reason, iterating the SCC-detection algorithm (as in the lambda-dropping algorithm) yields the desired result. Our current implementation of lambda-dropping supports both algorithms.

5.1.3 Peyton Jones's dependency analysis

In his textbook [28], Peyton Jones uses an analysis to generate block structure. In several ways, the algorithm for this dependency analysis is similar to the algorithm we employ for function localization. Its goal, however, differs. Assuming the lambda-lifted version of the DFA program (see

Figure 4) was extended with a main expression calling the function r with appropriate arguments, Peyton Jones's dependency analysis would yield the following skeleton:

```
let empty? = ...
in let f = ...
       g = ...
       h = ...
   in let err = ...
          fresh0 = ...
      in let r = ...
         in (...call to r...)
```

It is important to note that in this skeleton, all function names are visible to the other functions that need it. Their formal parameters, however, are not visible to these other functions. Peyton Jones's analysis places no function in the scope of any formal parameters, which is incompatible with parameter dropping.

More detail on Peyton Jones's dependency analysis, its purpose and properties, and its relation to our transformation can be found in the second author's MS thesis [32].

5.1.4 Isomorphism of trees and block structure

This section describes an isomorphism between the block structure of certain programs and trees. A lambda-lifted program consists of a set of equations, none of which contain block-structuring constructs. There is no inherent structure among the equations. Lambda-dropping such a program attempts to establish block structure, which by its very nature must conform to a tree structure. Transforming the dependencies between the equations into a tree structure accomplishes the task. The nodes of the tree are functions. The edges define the block structure. An edge from a function f to a function g indicates that g should be declared in a block local to f, along with any other direct successor of f.

Block-structured programs generated from a tree in this manner have letrec blocks and function declarations as their top-level constructs. The bodies of the letrec blocks and functions consist of other types of constructs. Lambda-dropped programs always have this form. We refer to it as Block-Structured Normal Form (BSNF) in the rest of this section.

Any program can be transformed to BSNF. Incrementally lambda-lifting the program to move definitions towards the global level, and creating letrec blocks as appropriate, performs the transformation. Programs in BSNF can be mapped to a tree by reversing the transformation employed by the lambda-dropper. Mapping the tree back to program form yields the same program, modulo the order of function declarations in a block.

Any tree of functions represents a block-structured program, and the block structure of any program in BSNF represents a tree. Transforming a program to a tree and back again yields the same program. Thus, the block structure of a program in BSNF is isomorphic to a tree.

5.2 Dropping formal parameters

A flow graph is used to build Use/Def chains for the formal parameters of each function. Given the U/D chains, parameter dropping is accomplished by detecting whether the formal parameter at the end of each chain is visible in the lexical scope. When building the flow graph, we must eliminate recursive dependencies between parameters. If a group of parameters depend on each other, they can either all be dropped or none of them can be dropped. The original strongly connected component algorithm isolates these recursive dependencies, ensuring that such groups of parameters are processed uniformly.

5.3 Properties of lambda-dropping

We designed lambda-dropping to provide an inverse for Johnsson's lambda-lifting algorithm [19]. Before we can state any inverseness properties, we need an appropriate notion of equality of programs. In the context of this paper, equality of programs is syntactic, modulo renaming of variables and modulo the ordering of mutually recursive declarations.

Consider several versions of a program that have been partially lambda-lifted or lambda-dropped. These versions do not have the same block structure but lambda-lifting maps them to the same set of mutually recursive equations. Consequently, we cannot hope to provide a unique inverse for lambda-lifting. However, it is our conjecture that the two following properties hold:

Property 1 *Lambda-dropping is the inverse of lambda-lifting on all programs that have been lambda-dropped.*

Property 2 *Lambda-lifting is the inverse of lambda-dropping on all programs that have been lambda-lifted.*

Property 1 is arguably the more complex of the two. Lambda-dropping a program requires re-construction of the block structure that was flattened by lambda-lifting. Formal parameters lifted by the lambda-lifter must be omitted.

Examining the lambda-dropping algorithm reveals that a function that is passed as argument never has its parameters dropped. Dropping the parameters of such functions is certainly possible, but is non-trivial since it requires a control-flow analysis to determine the set of variables being passed as arguments (see Section 6.3 for an example).

Restricting ourselves to providing an inverse for lambda-lifting eliminates the need for this analysis. If a function has no free variables before lambda-lifting, no additional parameters are added, and we need not drop any parameters to provide a proper inverse. If the function did have free variables, these variables are applied as arguments to the function *at the point where it is passed as an argument.* Thus, the extra parameters are easily dropped, since they are unambiguously associated with the function.

In languages such as Scheme where currying is explicit, a lambda-lifter may need to construct a function as a curried, higher-order function when lifting parameters. A lambda-dropper can easily detect such declarations (the currying performed by the lambda-lifter is redundant after lambda-dropping), and remove them.

Johnsson's lambda-lifting algorithm explicitly names anonymous lambda forms with let expressions, and eliminates let expressions by converting them into applications. A lambda-dropper can recognize the resulting constructs and reverse the transformations, thereby satisfying the inverseness properties.

6 Applications and Synergy

6.1 Partial evaluation

Our compelling motivation to sort out lambda-lifting and lambda-dropping is partial evaluation [12, 21]. As men-

tioned in Section 1, recursive equations offer a convenient format for a partial evaluator. Similix and Schism, for example [8, 10], lambda-lift source programs before specialization and produce residual programs in the form of recursive equations. Very often, because of arity raising, these recursive equations are afflicted with a huge number of parameters, which increases their compilation time enormously, sometimes to the point of making the whole process of partial evaluation impractical [14].

Our lambda-dropper handles the output language of Schism. As this article is going to press, we have integrated lambda-dropping into Schism, but have not yet had feedback from other Schism users.

6.1.1 Example: a fold function

Figure 8 displays a source program, which uses a standard fold function over a binary tree. Without any static input, Schism propagates the two static abstractions from the main function into the fold function. The raw residual program appears in Figure 9. It is composed of two recursive equations. The static abstractions have been propagated and statically reduced. The dynamic parameters `x` and `y` have been retained and occur as residual parameters.[5] They make the traversal function an obvious candidate for lambda-dropping. Figure 10 displays the corresponding lambda-dropped program, which was obtained automatically.

As partial-evaluation users, we find it more clear to compare Figure 8 and Figure 10 rather than Figure 8 and Figure 9. (N.B. A monovariant specializer would have directly produced the program of Figure 10, using mere unfolding and no memoization).

6.1.2 Example: the first Futamura projection

Let us consider a while-loop language as is traditional in partial evaluation and semantics-based compiling [11]. Figure 11 displays a source program with several while loops. Specializing the corresponding definitional interpreter (not shown here) using Schism with respect to this source program yields the residual program of Figure 12. Each source while loop has given rise to a recursive equation. Figure 13 displays the corresponding lambda-dropped program, which was obtained automatically.

Again, we find it more clear to compare Figure 11 and Figure 13 rather than Figure 11 and Figure 12. The relative positions of the residual recursive functions now match the relative positions of the source while loops. (N.B. Again, a monovariant specializer would have directly produced the program of Figure 13).

6.2 Programming environment

It is our programming experience that lambda-lifting and lambda-dropping go beyond a mere phase in a compiler for functional programs. They can offer truly useful (and often unexpected) views of one's programs. For example, lambda-dropping tells us that in Figure 8, the fold functional could have been defined locally to the main function.

In the context of teaching, these unexpected views often help students to improve their understanding of lexical scope and block structure, and to use them more effectively in programming. Sections 6.3 to 6.5 present more examples.

[5] That is how partially static values and higher-order functions inflate (raise) the arity of recursive equations.

```
(define-type binary-tree
  (leaf alpha)
  (node left right))

(define binary-tree-fold
  (lambda (process-leaf process-node init)
    (letrec ([traverse
              (lambda (t)
                (case-type t
                  [(leaf n)
                   (process-leaf n)]
                  [(node left right)
                   (process-node
                     (traverse left)
                     (traverse right))]))])
      (lambda (t) (init (traverse t))))))

(define main
  (lambda (t x y)
    ((binary-tree-fold
       (lambda (n) (leaf (* (+ x n) y)))
       (lambda (r1 r2) (node r1 r2))
       (lambda (x) x)) t)))
```

Figure 8: Source program.

```
(define (main-1 t x y)
  (traverse:1-1 t y x))

(define (traverse:1-1 t y x)
  (casetype t
    [(leaf n)
     (leaf (* (+ x n) y))]
    [(node left right)
     (node (traverse:1-1 left y x)
           (traverse:1-1 right y x))]))
```

Figure 9: Specialized (lambda-lifted) version of Figure 8.

```
(define (main-1 t x y)
  (letrec ([traverse:1-1
            (lambda (t)
              (case-type t
                [(leaf n)
                 (leaf (* (+ x n) y))]
                [(node left right)
                 (node (traverse:1-1 left)
                       (traverse:1-1 right))]))])
    (traverse:1-1 t)))
```

Figure 10: Lambda-dropped version of Figure 9.

```
{
  int res=1; int n=4; int cnt=1;
  while (cnt > 0) {
    res = 1; n = 4;
    while (n > 0) { res = n * res; n = n - 1; }
    cnt = cnt - 1;
  }
}
```

Figure 11: Example imperative program.

```
(define (evprogram-1 s)
  (evwhile-1
    (intupdate 2 1 (intupdate 1 4 (intupdate 0 1 s)))))

(define (evwhile-1 s)
  (if (gtint (fetchint 2 s) 0)
      (evwhile-2 (intupdate 1 4 (intupdate 0 1 s)))
      s))

(define (evwhile-2 s)
  (if (gtint (fetchint 1 s) 0)
      (let ([s-1 (intupdate 0 (mulint (fetchint 1 s) (fetchint 0 s)) s)])
        (evwhile-2 (intupdate 1 (subint (fetchint 1 s-1) 1) s-1)))
      (evwhile-1 (intupdate 2 (subint (fetchint 2 s) 1) s))))
```

Figure 12: Specialized (lambda-lifted) version of the definitional interpreter with respect to Figure 11.

```
(define (evprogram-1 s)
  (letrec ([evwhile-1
            (lambda (s)
              (letrec ([evwhile-2
                        (lambda (s)
                          (if (gtint (fetchint 1 s) 0)
                              (let ([s-1 (intupdate 0 (mulint (fetchint 1 s) (fetchint 0 s)) s)])
                                (evwhile-2 (intupdate 1 (subint (fetchint 1 s-1) 1) s-1)))
                              (evwhile-1 (intupdate 2 (subint (fetchint 2 s) 1) s))))])
                (if (gtint (fetchint 2 s) 0)
                    (evwhile-2 (intupdate 1 4 (intupdate 0 1 s)))
                    s)))])
    (evwhile-1 (intupdate 2 1 (intupdate 1 4 (intupdate 0 1 s))))))
```

Figure 13: Lambda-dropped version of Figure 12.

6.3 From Curry to Turing

Here is Curry's fixpoint operator [5]:

$$\lambda f.\text{let } g = \lambda x.f\ (x\ x)\\ \text{in } g\ g$$

f occurs free in g. Lambda-lifting this block yields the following λ-term:

$$\lambda f.\text{let } g = \lambda f.\lambda x.f\ (x\ x)\\ \text{in } g\ f\ (g\ f)$$

Control-flow analysis tells us that x can only denote $g\ f$ and that all the occurrences of f denote the same value. Thus we can safely relocate the second occurrence of f, in the let body, into the let header:

$$\lambda f.\text{let } g = \lambda f.\lambda x.f\ (x\ f\ x)\\ \text{in } g\ f\ g$$

Again, control-flow analysis informs us of the only application sites of the λ-abstraction denoted by g. Thus we can safely swap its two parameters:

$$\lambda f.\text{let } g = \lambda x.\lambda f.f\ (x\ x\ f)\\ \text{in } g\ g\ f$$

Eta-reducing this term yields Turing's fixpoint operator [5].

6.4 Detecting global variables

Following Schmidt's initial impetus on single-threading [31], Sestoft has investigated the detection of global variables in recursive equations [33], and Fradet, the detection of single-threaded variables using continuations [16]. Such variables come in two flavors: global, read-only variables, and updatable, single-threaded variables.

Lambda-dropping reveals read-only global variables by *localizing* blocks. (N.B. Many of these global variables are not global to a whole program, only for parts of it. These parts are localized.) Conversely, transforming a program into continuation-passing style (CPS) reveals single-threaded variables: their value is passed to the continuation. This last point of course suggests to lambda-drop after CPS transformation.

6.5 Continuation-based programming

Shivers optimizes a tail-recursive function by "promoting" its CPS counterpart from being a function to being a continuation [35]. For example, consider the function returning the last element of a non-empty list.

$$\begin{array}{l}\text{letrec last} = \lambda x.\text{let } t = \text{tl } x\\ \qquad\qquad\qquad\qquad\text{in if } t = \text{nil}\\ \qquad\qquad\qquad\qquad\quad\text{then hd } x\\ \qquad\qquad\qquad\qquad\quad\text{else last } t\\ \text{in last } l\end{array}$$

Its (call-by-name)[6] CPS counterpart can be written as fol-

[6] For example.

lows.

$$\begin{aligned}\lambda k.\text{letrec last}' = \lambda x.\lambda k.\text{tl}'\ x\ \lambda t.\text{if } t = \text{nil}\\ &\text{then hd}'\ x\ k\\ &\text{else last}'\ t\ k\\ \text{in last}'\ l\ k\end{aligned}$$

where hd′ and tl′ are the CPS versions of hd and tl, respectively. The type of last′ reads:

$$\text{Value} \to (\text{Value} \to \text{Answer}) \to \text{Answer}.$$

Shivers promotes last′ from the status of function to the status of continuation as follows:

$$\begin{aligned}\lambda k.\text{letrec last}' = \lambda x.\text{tl}'\ x\ \lambda t.\text{if } t = \text{nil}\\ &\text{then hd}'\ x\ k\\ &\text{else last}'\ t\\ \text{in last}'\ l\end{aligned}$$

The type of last′ now reads:

$$\text{Value} \to \text{Answer}.$$

It coincides with the type of a continuation, since last′ does not pass continuations anymore. Promoting a function into a continuation amounts to parameter-dropping its continuation.

Lambda-dropping the CPS counterpart of programs that use call/cc also offers a convenient alternative to dragging around escape functions at each function call.

6.6 An empirical study

Lambda-dropping a program removes formal parameters from functions, making locally bound variables free to the function. An implementation must handle these free variables. In languages where functions are first-class citizens, it is usually necessary to store the bindings of these free variables when passing a function as a value. Most implementations use closures for this purpose.

6.6.1 Considerations

Creating a closure usually incurs extra overhead. Values must be copied into the closure, the closure takes up either stack or heap space and it is manipulated by the memory manager. Looking up values in a closure often takes more instructions than referencing a formal parameter. The closure of a function is created upon entry into its definitional block. Thus, the function can be called many times with the same closure. This is relevant in the case of recursive functions, since the surrounding block has already been entered when the function calls itself.

A recursive function in a program written without block structure must explicitly manipulate everything it needs from the environment at every recursive call. A contrario, if the function has many free variables, e.g., after lambda-dropping, the performance of the program may be improved:

- Fewer values need to be pushed onto the stack at each recursive call. This reduces the number of machine instructions spent on each function invocation.
- If a free variable is used in special cases only, it might not be manipulated during the execution of the body of the function. This reduces the amount of data processing, potentially reducing register spilling and improving cache performance.

A compiler unfolding recursive functions up to a threshold could lambda-drop locally before unfolding, thereby globalizing constant parameters, for example.

6.6.2 Experiments

Initial experiments suggest that lambda-dropping can improve the performance of recursive functions, most typically for programs performing recursive descents. The improvement depends on the implementation, the number of parameters removed, the resulting number of parameters, and the depth of the recursion. It is our experience that lambda-dropping increases performance for the following implementations of block-structured languages:

- Scheme: SCM and Scheme 48;
- ML: Standard ML of New Jersey and Moscow ML;
- Haskell: the Glasgow Haskell Compiler and Gofer;

and also for implementations of imperative languages such as Delphi Pascal, Gnu C and Java 1.1.

In some implementations, such as Standard ML of New Jersey, dropping one out of two formal parameters decreases performance. But dropping five out of seven can (in slightly contrived cases) result in a program 7 times faster than the original. In Chez Scheme, however, lambda-dropping entails a slight slowdown in all cases. Limiting our tests to a few programs stressing recursion and parameter passing gave speedups ranging from 1.05 to 2.0 in most cases. This was observed on all implementations but Chez Scheme.

We are currently developing an abstract model describing the costs of procedure invocation and closure creation. The model is parameterized by the costs of the basic operations of a low-level abstract machine. Perhaps it will prove useful to explain the results and, if possible, predict the usefulness of lambda-dropping in given situations.

More detailed information on the experiments, the abstract model, and the results can be found in the second author's MS thesis [32].

6.7 Time complexity

The lambda-lifting algorithm has a time complexity of $\mathcal{O}(n^3 + m \log m)$, where n is the maximal number of functions declared in a letrec block and m is the size of the program. The n^3 component is derived from solving the set equations during the parameter lifting stage [19].

The lambda-dropping algorithm has a time complexity of $\mathcal{O}(n^2 + m \log m)$, where n is the number of definitions and m is the size of the program. The n^2 component is derived from Step 3 of the parameter-dropping stage. In a worst-case situation, we may need to make n traversals of the graph (which has size n) during this step.

7 Related Work

Aside from Peyton Jones's localization of blocks (Section 5.1.3), Sestoft's detection of read-only variables (Section 6.4) and Meijer's unpublished note "Down with Lambda-Lifting" (April 1992) — none of which directly addresses lambda-dropping — we do not know of any work about lambda-dropping. There is, however, plenty of work related to lambda-lifting.

7.1 Enabling principles

The enabling principles of lambda-lifting are worth pointing out: Peter Landin's correspondence principle [23], which has been formalized as categorical exponentiation [4], lets us remove let statements.

$$\text{let } x = a \text{ in } e \quad \equiv \quad (\lambda x.e)\, a$$

Eta-expansion lets us removes free variables. Let associativity enables let-floating, which lets us globalize function definitions that have no free variables.

7.2 Curried and lazy vs. uncurried and eager programs

Johnsson concentrated on lambda-lifting towards mutually recursive equations [19], but alternative approaches exist. The first one seems to be Hughes's supercombinator abstraction, where recursion is handled through self-application and full laziness is a point of concern [18]. Peyton Jones provides a broad overview about fully lazy supercombinators [27, 28, 29]. Essentially, instead of lifting only free variables, one lifts maximally free expressions. Fully lazy lambda-dropping would amount to keep maximally free expressions instead of identifiers in the initial calls to local functions.

In their Scheme compiler Twobit, Clinger and Hansen also use lambda-lifting [9]. They, however, modify the flow equations to reduce the arity of lambda-lifted procedures. Lambda-lifting is also stopped when its cost outweighs its benefits, regarding tail-recursion and allocation of closures in the heap. Lambda-lifting helps register allocation by indicating unchanging arguments across procedure calls.

7.3 Closure conversion

To compile Standard ML programs, Appel represents a closure as a vector [2]. The first element of the vector points to a code address. The rest of the vector contains the values of the free variables. Applying a closure to actual parameters is done by passing to the code address the closure itself and the actual parameters. Thus calls are compiled independently of the number of free variables of the called function. This situation is obtained by "closure conversion." Once a program is closure-converted, it is insensitive to lexical scope and thus it can be turned into recursive equations.

Closure conversion, however, is different from lambda-lifting, for the two following reasons:

- In both closure-converted and lambda-lifted programs, lambda abstractions are named. In a closure-converted program, free variables are passed only when the name is defined. In a lambda-lifted program, free variables are passed each time the name is used.
- Closure conversion only considers the free variables of a lambda-abstraction. Lambda-lifting also considers those of the callees of this lambda-abstraction.

In the latter sense, lambda-lifting can be seen as the transitive closure of closure conversion.

Steckler and Wand consider a mix between lambda-dropping and closure conversion: so-called "lightweight closures" [39]. Such closures do not hold the free variables that are in scope at the application sites of this closure. A similar concern leads Shao and Appel to decide whether to implement closures in a deep or in a flat way [34].

7.4 Analogy with the CPS transformation

An analogy springs to the mind: continuation-based compilation. As observed by Sabry, Felleisen, *et al.* [15], CPS compilers proceed in two steps: first, source programs are transformed into continuation-passing style, but eventually they are mapped back to direct style.

One is left with the conjecture that both transformations (lambda-dropping and CPS transformation) expose, in a simpler way, more information about the structure of a program during its journey through a compiler. The CPS transformation reveals control-flow information, while lambda-dropping reveals scope information. As pointed out in Section 7.2, this information is useful for lambda-lifting proper. We believe that it is also useful for stackability detection by region inference.

7.5 Stackability

Recently, Tofte and Talpin have suggested to implement the λ-calculus with a stack of regions and no garbage collector [37]. Their basic idea is to associate a region for each lexical block, and to collect the region on block exit. While this scheme is very much allergic to CPS (which "never returns"), it may very well benefit from preliminary lambda-dropping, since the more lexical blocks, the better for the region inferencer. We leave this issue for future work.

7.6 Partial evaluation

Instead of lambda-lifting source programs and lambda-dropping residual programs, a partial evaluator could process block-structured programs directly. In the diagram of the abstract, we have depicted such a partial evaluator with a dashed arrow. To the best of our knowledge, however, except for Malmkjær and Ørbæk's case study presented at PEPM'95 [25], no partial evaluator for procedural programs handles block structure today.

As analyzed by Malmkjær and Ørbæk, polyvariant specialization of higher-order, block-structured programs faces a problem similar to Lisp's "upward funarg." An upward funarg is a closure that is returned beyond the point of definition of its free variables, thus defeating stackability. The partial-evaluation analogue of an upward funarg is a higher-order function that refers to a specialization point but is returned past the scope of this specialization point. What should the specializer do? Ideally it should move the specialization point outwards to its closest common ancestor together with the point of use for the higher-order function. Lambda-dropping residual recursive equations achieves precisely that, merely by sinking blocks as low as possible.

The problem only occurs for polyvariant specializers for higher-order, block-structured programs where source programs are not lambda-lifted and program points are specialized with respect to higher-order values. Of these, there are few: Lambda-Mix [17] and type-directed partial evaluation [13] are monovariant; Schism [10] and Similix [8] lambda-lift before binding-time analysis; Pell-Mell [24] lambda-lifts after binding-time analysis; ML-Mix [6] does not specialize with respect to higher-order values; and Fuse [30] does not allow upwards funargs.

7.7 Other program transformations

Other program transformations can also benefit from lambda-dropping: in Wadler's work on deforestation [38],

for example, the "macro" style amounts to lambda-dropping by hand.

8 Conclusion and Issues

In the mid 80's, Hughes, Johnsson and Peyton Jones presented lambda-lifting: the transformation of functional programs into recursive equations. Since then, lambda-lifting seems to have been mostly considered as an intermediate phase in compilers. It is our contention that lambda-lifting is also interesting as a source-to-source program transformation, together with its inverse: lambda-dropping.

In this article, we have introduced lambda-dropping, outlined some of its properties, and mentioned some other applications than our main one: as a back end in a partial evaluator. We have implemented a lambda-dropper in Scheme for the target language of Schism and we plan to port it in ML for Pell-Mell. As this article is going to press, we do not have (yet) any significant statistics on the success rate of lambda-dropping over typical residual programs. We are also formalizing lambda-lifting and lambda-dropping in order to prove their correctness (it should be mentioned that we are not aware of any such correctness proofs for lambda-lifting). Finally, we are investigating faster algorithms using set constraints.

Acknowledgements

Thanks are due to Anindya Banerjee, Charles Consel, Andrzej Filinski, Daniel P. Friedman, Nevin C. Heintze, Tue Jakobsen, Kristoffer Rose, and Peter Sestoft for their kind interest on the general topic of lambda-lifting and lambda-dropping. Special thanks to John Hatcliff, Julia L. Lawall, Karoline Malmkjær, and the anonymous referees for their perceptive and timely comments.

The diagrams were drawn with Kristoffer Rose's XY-pic package. The Scheme programs were pretty-printed with R. Kent Dybvig's Chez Scheme system.

References

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

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

[3] Lennart Augustsson. *Compiling Lazy Functional Languages, part II.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1988.

[4] Anindya Banerjee and David A. Schmidt. A categorical interpretation of Landin's correspondence principle. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 587–602, New Orleans, Louisiana, April 1993.

[5] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, 1984.

[6] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, August 1993. DIKU Rapport 93/22.

[7] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[8] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, 1993.

[9] William Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In Talcott [36], pages 128–139.

[10] 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.

[11] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[12] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[13] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[14] Olivier Danvy, Nevin C. Heintze, and Karoline Malmkjær. Resource-bounded partial evaluation. *ACM Computing Surveys*, 28(2):329–332, June 1996.

[15] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[16] Pascal Fradet. Syntactic detection of single-threading using continuations. In John Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 241–258, Cambridge, Massachusetts, August 1991. Springer-Verlag.

[17] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[18] 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.

[19] 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.

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

[21] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.

[22] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[23] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[24] 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. Also appears as Technical report CMU-CS-94-129.

[25] Karoline Malmkjær and Peter Ørbæk. Polyvariant specialization for higher-order, block-structured languages. In William L. Scherlis, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–76, La Jolla, California, June 1995.

[26] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. In R. Kent Dybvig, editor, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 1–12, Philadelphia, Pennsylvania, May 1996. ACM Press.

[27] 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.

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

[29] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1992.

[30] Erik Ruf. *Topics in Online Partial Evaluation.* PhD thesis, Stanford University, Stanford, California, February 1993. Technical report CSL-TR-93-563.

[31] David A. Schmidt. Detecting global variables in denotational definitions. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[32] Ulrik P. Schultz. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1997. Forthcoming.

[33] Peter Sestoft. Replacing function parameters by global variables. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 39–53, London, England, September 1989. ACM Press.

[34] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In Talcott [36], pages 150–161.

[35] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[36] Carolyn L. Talcott, editor. *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

[37] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In Boehm [7], pages 188–201.

[38] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1989. Special issue on ESOP'88, the Second European Symposium on Programming, Nancy, France, March 21-24, 1988.

[39] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In Boehm [7], pages 435–445.