Optimal Lambda Lifting in Quadratic Time

Marco T. Morazán and Ulrik P. Schultz

Seton Hall University, South Orange, NJ, USA
morazanm@shu.edu
University of Southern Denmark, Odense, Denmark
ups@mmmi.sdu.dk

Abstract. The process of lambda lifting flattens a program by lifting all local function definitions to the global level. Optimal lambda lifting computes the minimal set of extraneous parameters needed by each function as is done by the $O(n^3)$ equation-based algorithm proposed by Johnsson. In contrast, modern lambda lifting algorithms have used a graph-based approach to compute the set of extraneous parameters needed by each function. Danvy and Schultz proposed an algorithm that reduced the complexity of lambda lifting from $O(n^3)$ to $O(n^2)$. Their algorithm, however, is an approximation of optimal lambda lifting. Morazán and Mucha proposed an optimal graph-based algorithm at the expense of raising the complexity to $O(n^3)$. Their algorithm, however, suggested that dominator trees might be used to develop an $O(n^2)$ algorithm. This article explores the relationship between the call graph of a program, its dominator tree, and lambda lifting by developing algorithms for successively richer sets of programs. The result of this exploration is an $O(n^2)$ optimal lambda lifting algorithm.

1 Introduction

The process of lambda lifting flattens a program by lifting all local function definitions to the global level. In order to perform this program transformation the free variables of a function, f, and a subset of the free variables transitively needed by its callees, must be added as formal parameters to f before it can be lifted to the global level. That is, f must be made scope insensitive before it can be moved to the global level. Free variables must be explicitly passed to f, because at runtime the lifted version of f does not have the benefit of a closure to store the bindings of the free variables. This program transformation technique is valid for programs using higher-order functions, because the extraneous parameters are passed to function references (e.g. the site where functions are passed as arguments) rather than to function calls (e.g. the site where a function is applied to a set of arguments).

Lambda lifting is important for restructuring functional programs written for the web [7], for partial evaluators [1], and for efficient compilation [15]. Furthermore, many abstract machines for functional languages only handle lambda-lifted programs [10,12] making this transformation an important step in several compilers for functional languages [9,11]. Lambda lifting and its inverse lambda

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 37–56, 2008.
© Springer-Verlag Berlin Heidelberg 2008

dropping [2] are also important for improving the performance of compiled programs by providing a mechanism through which the number of parameters of a function can be optimized for the target machine. For example, functions with a large number of parameters (which are handled poorly by most compilers) can be transformed to have fewer parameters [2]. Danvy and Schultz also point out that in the context of teaching, lambda lifting and lambda dropping are useful by offering different views of programs that help students understand lexical scoping and block structure [2].

The computation of the set of free variables needed by a lifted function makes lambda lifting difficult. Modern graph-based approaches [3,14] tackle the problem by transforming the call graph of a program into a directed acyclic graph that is used to propagate free variables. The algorithm developed by Danvy and Schultz [3] improves the complexity of Johnsson's [8] lambda lifting algorithm from $O(n^3)$ to $O(n^2)$. Their algorithm, however, is not optimal because it may unnecessarily increase the arity of lifted functions. The algorithm developed by Morazán and Mucha [14] makes graph-based lambda lifting optimal at the cost of increasing its complexity to $O(n^3)$.

In this article, we first review Johnsson's (J), Danvy's and Schultz's (DS), and Morazán's and Mucha's (MM) lambda lifting algorithms. After this review, we present a new insight that simplifies the presentation and the implementation of graph-based lambda lifting by using a depth-first traversal instead of a reversed breadth-first traversal to propagate free variables. The article then explores the relationship between call graphs, dominator trees, and lambda lifting. The result of this exploration is an optimal $O(n^2)$ lambda lifting algorithm. Although the discussion is technically intricate at some points, the resulting algorithm is simple and elegant. The presentation assumes that all variable names are unique. Programs for which this does not hold can easily be transformed by generating a fresh identifier for repeated identifiers [4]. Moreover, since lambda-lifting (as pointed out earlier) is indifferent to higher-order functions, our presentation only uses first-order programs as examples. The article ends with some concluding remarks and directions of future work. The appendix includes a brief glossary of the graph terminology used in the article (i.e. tree, dominator tree, and strongly connected component).

2 Lambda Lifting Algorithms

2.1 Johnsson's Algorithm

In the J-algorithm, the source program is traversed top-down to compute the required (i.e. minimal) set of extraneous parameters needed by each function. For any given function, f, the equation for the required set of free variables of f, R_f, is given by:

$$R_f = FV_f \cup ((\cup_{g \in FF_f} R_g) \cap SV_f), \tag{1}$$

where FV_f is the set of free variables directly referenced by f, FF_f is the set of functions referenced by f, and SV_f is the set of variables defined in f's enclosing

```
(define (f x y)
  (define (g. . .) (. . .x. . .i. . .))
  (define (h. . .) (. . .y. . .f. . .))
  (define (i. . .) (. . .f. . .))
 (. . .g. . .h. . .))
```

Fig. 1. First Scheme Pseudo-code

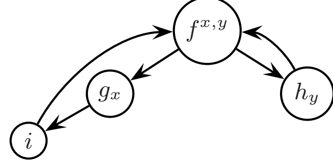

Fig. 2. Call Graph

lexical scope. Mutually recursive functions give rise to a system of mutually recursive equations which is solved by traversing down the parse tree. Once R_f is known it is used to compute the minimal set of free variables for functions declared further down the program's parse tree.

To illustrate how the J-algorithm works using equation (1) consider the pseudo-code in Figure 1. At the topmost level of the parse tree the free variables of f are computed by solving the following equation:

$$R_f = FV_f \cup \left((\cup_{g \epsilon FF_f} R_g) \bigcap SV_f \right) .$$

Since $FV_f = SV_f = \emptyset$, we may conclude that $R_f = \emptyset$.

At the next level of the parse tree, the free variables equations to solve are:

$$\begin{aligned}
R_g &= FV_g \cup \left((\cup_{j \epsilon FF_g} R_j) \cap SV_g \right) \\
&= \{x\} \cup \{R_i \cap \{x, y\}\} \\
&= \{x\} \cup \{\{FV_i \cup ((\cup_{j \epsilon FF_i} R_j) \cap SV_i)\} \cap \{x, y\}\} \\
&= \{x\} \cup \{\{\emptyset \cup \{R_f \cap \{x, y\}\}\} \cap \{x, y\}\} \\
&= \{x\} \cup \{\emptyset \cup \{\emptyset \cap \{x, y\}\} \cap \{x, y\}\} \\
&= \{x\} \\
R_h &= FV_h \cup ((\cup_{j \epsilon FF_h} R_j) \cap SV_H) \\
&= \{y\} \cup \{R_f \cap \{x, y\}\} \\
&= \{y\} \cup \{\emptyset \cap \{x, y\}\} \\
&= \{y\} \cup \emptyset \\
&= \{y\} \\
R_i &= FV_i \cup ((\cup_{j \epsilon FF_i} R_j) \cap SV_i) \\
&= \emptyset \cup \{FV_f \cap \{x, y\}\} \\
&= \emptyset \cup \{\emptyset \cap \{x, y\} \\
&= \emptyset \cup \emptyset \\
&= \emptyset
\end{aligned}$$

Notice that x is not identified as an extraneous parameter needed by h and that y is not identified as an extraneous parameter needed by g nor i. Furthermore, x is not identified as an extraneous parameter for i. This occurs, because the set of extraneous parameters needed by f, an ancestor of g, h, and i in the program's parse tree, are computed before the set of extraneous parameters needed by g, h, and i. Thus, the members of FF_f are not explored during the computation of R_g, R_h, and R_i and do not contribute extraneous parameters to g, h, and i.

The time complexity of the J-algorithm is $O(n^3)$, where n is the size of the program. Briefly, globally there are $O(n)$ equations to solve the transitive closure problem, which requires $O(n)$ steps of set union operations each taking $O(n)$ per equation.

2.2 Danvy's and Schultz's Graph-Based Lambda Lifting

To perform lambda lifting in quadratic time, a program is represented as a call graph. Each node in this graph represents a function. An edge from f to g means that there is a reference to g in the body of f. Mutually recursive functions give rise to strongly connected components (akin to Johnsson's mutually recursive equations). Danvy and Schultz observed that a function, f, in a strongly connected component can be given as extraneous parameters the set of free variables lexically visible to f found in the union of the free variables of the functions that constitute the component. Therefore, strongly connected components can be coalesced in the call graph of a program to yield a directed acyclic graph that is traversed in a reversed breadth-first order to propagate free variables between nodes.

To illustrate the DS-algorithm consider the call graph in Figure 2 for the pseudo-code in Figure 1. In the call graph each node is labeled with the name of a function. The superscript at the right of each function name is the set of variables declared by the function that appear in the pseudo-code and the subscript at the right of each function name is the set of free variables referenced by the function. The nodes in the call graph form a strongly connected component and are coalesced yielding a graph with a single node. The union of all the free variables of the functions in the node (i.e. f, g, h, and i) is taken. For each function, the lexically visible variables in this union become parameters to the lifted functions. That is, $\{x, y\}$ are identified as extraneous parameters for g, h, and i.

The time complexity of the DS-algorithm is $O(n^2)$, where n is the size of the program. Briefly, the coalesced call graph of size $O(n)$ defines a global order in which free variables can be linearly propagated through the graph, with each propagation step performing a set unification of size $O(n)$.

2.3 Morazán's and Mucha's Graph-Based Algorithm

Morazán and Mucha observed that using strongly connected components to propagate free variables may result in an approximation of the required set of extraneous parameters needed by lifted functions as exemplified by the results obtained by the J-algorithm and the DS-algorithm for the pseudo-code in Figure 1. Unnecessary extraneous parameters may be added to lifted functions for two reasons. The first reason is that functions can be members of a strongly connected component that contains nested strongly connected components and that also contains functions defined at different levels in the program's parse tree. Suppose r is a function defined at level n in the parse tree of a program and that there are m disjoint sets of functions (modulo r), $D_1 \ldots D_m$, defined

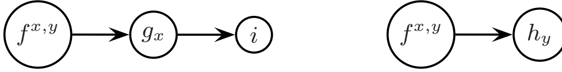

Fig. 3. MM-Algorithm Components for the Call Graph in Figure 2

at any level greater than n (i.e. in the parse tree of the program r is an ancestor of these functions) such that r dominates all paths from functions in D_i to functions in D_j, $i \neq j$. In such a scenario, r may declare variables that are free[1] for functions in D_i that are not needed as extraneous parameters by functions in D_j and viceversa. This may occur, for example, when r is contained in two independent loops (modulo r).

The second reason is that a variable, x, declared by r that is free in D_i may not be needed as an extraneous parameter by all the functions in D_i. For example, let r and s be members of the same loop such that x is known to be free in s and is declared by r. The variable x only needs to be carried by successors of s in the call graph if there is a path, that does not contain r, from s to another function where x is directly referenced. This follows from the observation that the successors of s do not need to make x available to any other function if such a path does not exist. Thus, these successors do not require x as an extraneous parameter.

The MM-algorithm is an improvement of the DS-algorithm that reduces the arity of lifted functions by computing the minimal set of extraneous parameters needed by each lifted function, as is done by the J-algorithm, based on the observations above. Extraneous parameters are reduced by splitting the strongly connected components of a call graph that contain functions defined at different levels in the program's parse tree into multiple components based on its nested strongly connected components and by ignoring edges into a dominating function that are internal to any such component after the split. Splitting strongly connected components into multiple components guarantees that free variables local to a component (e.g. declared by the dominating function) do not propagate between nested strongly connected components. Ignoring internal edges into the dominating function of a nested strongly connected component guarantees that a free variable local to a component is not propagated beyond the last function that references it in a loop. This occurs, because the removal of such edges eliminates the loop and, therefore, these functions no longer constitute a strongly connected component and do not receive the same set of extraneous parameters.

To illustrate the MM-algorithm once again consider the call graph in Figure 2. The graph is split into two components displayed in Figure 3. This disconnected graph is used to propagate free variables between nodes in a reversed breath-first order. Notice that the dominating ancestor function, f, is a member of two components which prevents its descendants from unnecessarily contributing free variables to each other. By ignoring the edges into f in Figure 2, the nested

[1] We call such free variables *local* to the strongly connected component.

```
(define (f x)
  (define (g...) (...i...))
  (define (h...) (...x...))
  (define (i...) (...h...))
 (...g...h...))
```

Fig. 4. Second Scheme Pseudo-code

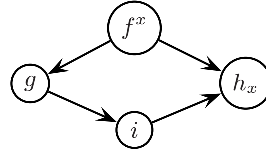

Fig. 5. Call Graph for Figure 4

strongly connected components cease themselves to be strongly connected. During the propagation of free variables, y is not unnecessarily propagated to g and i, and x is not unnecessarily propagated to h and i. Notice that within the loop formed by $\{f, g, i\}$ in Figure 2 only g requires and receives x as an extraneous parameter. This algorithm yields the same results as the J-algorithm.

The time complexity of the MM-algorithm is $O(n^3)$, where n is the size of the program. Briefly, strongly connected components must be split $O(n)$ times. For each split the resulting coalesced call graph is of size $O(n)$ and each propagation step performs a set unification of size $O(n)$.

3 A Simplifying Insight

The graph-based lambda lifting algorithms developed to date use the reversed breadth-first ordering of the nodes of an acyclic graph to ensure that a node is only processed once all of its successors in the call graph have been processed. Successor nodes must be processed first, because the required set of free variables of predecessor nodes depends on them. The use of this ordering, however, requires that special attention be paid to calls from functions appearing late in the reversed breadth-first ordering to functions appearing early in the reversed breadth-first ordering.

To illustrate the problem consider the Scheme pseudo-code in Figure 4 and its diamond-shaped call graph in Figure 5. In this graph the function f declares x (noted as right superscript) and x is free in h (noted as a right subscript). The breadth-first ordering of the nodes is: $\{f, g, h, i\}$[2]. There are no strongly connected components and, thus, nothing to coalesce. Having an acyclic graph means that free variables ought to be propagated from callees to callers in a reversed breadth-first order. For our example that order is: $\{i, h, g, f\}$. If free variables are simply propagated from callees to callers nothing propagates from i to g, from h the free variable x propagates to i and nothing propagates to f, and nothing propagates from g to f. The end result is that x is identified as a free variable for h and i, but not for g which also needs x as a free variable. To avoid this pitfall, the DS-algorithm unifies the set of local free variables with the set of free variables of the immediate successors in the call graph. Thus, x propagates from i to g when g is processed.

[2] The breadth-first ordering could also be $\{f, h, g, i\}$, but this is irrelevant for our purposes.

We observe that if the graph is acyclic, as is always the case after coalescing strongly connected components, then a depth-first traversal of the graph can be used to propagate free variables: every time the process pops back from a node to its antecessor free variables are propagated. This ensures that all successors are processed before a caller is processed. For the call graph in Figure 5, a depth-first traversal follows the path $f \rightarrow g \rightarrow i \rightarrow h$. The free variable x is propagated back through this path from h to i and finally to g. The depth-first traversal then proceeds down the path $f \rightarrow h$ and nothing additional is propagated from h to f before terminating. Although the result is the same as using the reversed breadth-first ordering, this process is more elegant and simplifies the implementation of lambda lifting.

Propagation using a depth-first traversal instead of a reversed breadth-first ordering is still proportional to the number of function calls and the number of declared variables in the program. This type of traversal does not change the time complexity of neither the DS-algorithm nor the MM-algorithm.

4 Call Graphs and Dominator Trees

The key lessons that must be highlighted from the previous sections are:

1. **J-algorithm:** The set of extraneous parameters for an ancestor function in a parse tree must be known before finalizing the computation of the set of extraneous parameters for any descendant of this function.
2. **DS-algorithm:** Lambda lifting can be done using a graph-based approach. Furthermore, functions in a strongly connected component of a call graph that do not have references to any free variables local to the component can be coalesced. These functions all require the same set of extraneous parameters.
3. **MM-algorithm:** Dominating functions must not be coalesced with their dominated functions in order to avoid dominated functions from unnecessarily contributing free variables to each other. Furthermore, simple loops on a dominating function must be dissolved in order to avoid unnecessary propagation of free variables.
4. **New Observation:** Once an acyclic graph is obtained for a graph-based approach a depth-first traversal can be used to simplify the process of propagating free variables.

The MM-algorithm repeatedly computes strongly connected components in order to avoid the unnecessary propagation of local free variables. The splitting of a strongly connected component is always done around an ancestor function that dominates all paths between disjoint sets of functions within the strongly connected component when the strongly connected component contains functions defined at different levels in the program's parse tree. This observation suggests that dominator trees can be used to perform lambda lifting.

A defining property of a dominator tree is that an ancestor function always appears before its descendants. Thus, a dominator tree tells us for which functions the complete set of free variables must be computed first. For our purposes,

```
(define (f x y z)
  (define (g a b)
    (define (h c d)
      (define (i e) (...h...g...))
     (...i...))
   (...h...))
  (...g...))
```

Fig. 6. Third Scheme Pseudo-code

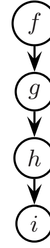

Fig. 7. Dominator Tree

an interesting feature of the dominator tree of a call graph is that independent loops dominated by a function are represented as different branches out of the dominating function which precludes the need to dissolve simple loops. Dominator trees, therefore, can be used as the basis of a graph used to propagate free variables. Since dominator trees can be computed in linear time [16], the need to repeatedly compute strongly connected subcomponents, which makes the MM-algorithm cubic, can be eliminated to reduce the complexity of lambda lifting.

The dominator tree, however, does not capture all dependencies between functions needed for lambda lifting. We classify these missing dependencies as *vertical* and *horizontal* dependencies, described in Sections 5 and 6 respectively. Vertical dependencies capture dependencies arising due to recursion between ancestors and descendants in the dominator tree. Horizontal dependencies capture dependencies arising between functions that do not have a vertical dependence in the dominator tree. Vertical dependencies are annotated on the dominator tree and are used to drive the propagation of free variables throughout the tree. Horizontal dependencies are added to the dominator tree, which necessitates coalescing the strongly connected components to obtain a directed acyclic graph. The resulting coalesced graph is used to propagate free variables.

5 Vertical Function Dependencies

We define a *downward* vertical dependence as the dependence that exists between a function and a descendant in the parse tree. At runtime, a call to any local function, g, must be preceded by calls to g's ancestors in the parse tree which are also ancestors of g in the dominator tree of the call graph of the program. Any extraneous parameters that g contributes to its ancestors can be propagated up the dominator tree.

We define an *upward* vertical dependence as the dependence that exists between a function, g, and a function, f, which is an ancestor of g. The function g may depend on several of its ancestors in the dominator tree of which we are interested in the one that has the maximum depth. We define the lowest upward vertical dependence of g, LD_g, as the function with the maximum depth in the

```
(define (f x y z)
  (define (g ...) (...x...i...))
  (define (h ...) (...y...f...))
  (define (i ... (...z...f...g...))
 (...g...h...))
```

Fig. 8. Fourth Sample Scheme Pseudo-code

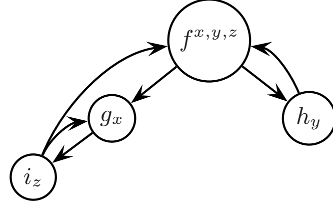

Fig. 9. Call Graph and Relevant Variables

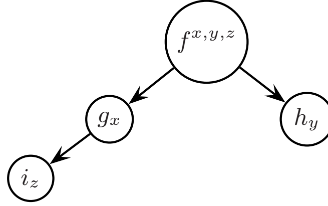

Fig. 10. Dominator Tree (DT)

dominator tree that g depends on. LD_g, if it exists, is the ancestor of g with the maximum depth that is either called by g or is called from any of g's descendants in the dominator tree. For example, consider the pseudo-code in Figure 6. The dominator tree for this code is displayed in Figure 7. Observe that i calls g and h which are ancestors of i in the dominator tree. Since h has the maximum depth, we have that $LD_i = h$. The function h does not call any of its ancestors, but it depends on its ancestor g which is called from i. Since g is the only ancestor of h that is referenced by any function in the subtree rooted at h, we have that $LD_h = g$. Finally, $LD_g = LD_f = \emptyset$ because none of the ancestors of g or f are referenced by functions in the subtrees of the dominator tree rooted at these functions.

To start exploring lambda lifting algorithms let us restrict our observations to the class of programs in which all dependencies are vertical (this restriction will be removed in the next section). Upward vertical dependence is not captured by a dominator tree, but can be computed as free variables are propagated up the dominator tree to identify the extraneous parameters contributed by downward vertical dependencies. Along with free variables, the set of referenced ancestor functions is propagated up the dominator tree.

Clearly, all extraneous parameters for g contributed by its descendants will reach g during an upward propagation. The following theorem establishes that LD_g, if it exists, contains all the extraneous parameters needed by g that are contributed by its ancestors in the dominator tree.

Theorem 1. *The set of extraneous parameters needed by LD_g contains all the extraneous parameters needed by g from its ancestors.*

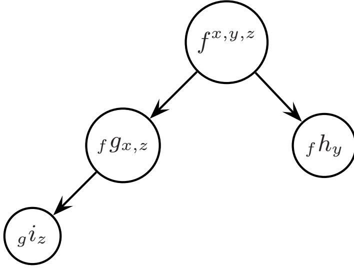

Fig. 11. DT After Upward Propagation

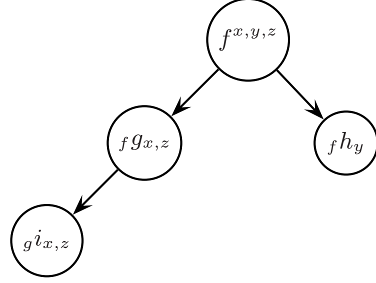

Fig. 12. DT After Downward Propagation

Proof. Let DT be the dominator tree for a call graph, CG, and let h be LD_g. Assume x is an ancestor-contributed extraneous parameter needed by g that is not a member of the set of extraneous parameters needed by h. If x is defined by an ancestor of h, then x must be a member of the set of extraneous parameters needed by h which contradicts our assumption. This follows from observing that h must carry x in order to make it available to g. If x is defined by h or a descendant of h, then there must exist a path in CG from g to a function where x is a known free variable that does not contain the function that declares x. All the functions on this path must be descendants of h in the dominator tree which means that $LD_g \neq h$. This contradicts our assumption and completes the proof that x must be a member of the set of extraneous parameters needed by h. Q.E.D

To illustrate how vertical dependencies are used in lambda lifting consider the pseudo-code in Figure 8 whose call graph is displayed in Figure 9. Figure 10 displays its dominator tree. Free variables needed by functions due to downward vertical dependence can be propagated up the dominator tree using a depth-first traversal. After this is done, the variable z has been propagated from i to g. In addition during this propagation step, the LD_i of each function i is computed by also propagating relevant upward vertical dependencies. LD_i is g and LD_h is f, because for leaves the LD function is the lowest ancestor in the dominator tree that they directly reference. Nodes pass the set of referenced ancestors back up the tree along with their free variables. In this manner, LD_g becomes f as it is the ancestor of g with the largest depth that is referenced from a function in the subtree rooted at g. The result of this step is displayed in Figure 11 in which the subscript to the left of each function name is its lowest dependence function. Finally, free variables need to be propagated down the dominator tree to satisfy upward vertical dependencies. This propagation proceeds in a breadth-first order propagating to function i the free variables needed by LD_i. A breadth-first order propagation is required to guarantee that the extraneous parameters of ancestor functions are known before the extraneous parameters of any descendant function are computed (which satisfies the key lesson highlighted from the J-algorithm). During this step the variable x is propagated from g to i. The result of this propagation step is displayed in Figure 12.

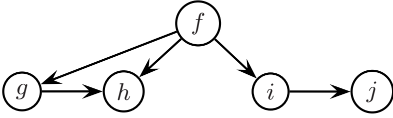

Fig. 13. Call Graph with Calls Among Siblings

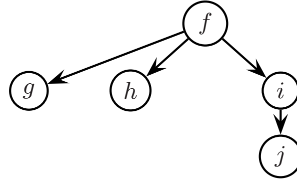

Fig. 14. Dominator Tree Lacks Some Function Dependencies

6 Horizontal Function Dependencies

Lexical scoping restricts the set of functions that may be directly referenced by any given function to itself, its children, its ancestors, its siblings, and its uncles in the parse tree. References to itself do not contribute new free variables to the lifted version of the function. References to ancestors and children are all captured as vertical dependencies annotated in the dominator tree as described in the previous section. References to siblings and uncles are references to functions with which there may be no dominance relation. For example, consider the call graph in Figure 13. Assume that f is the parent of g, h, i, and j in the parse tree. The dominator tree is displayed in Figure 14. Notice that i dominates its sibling j while there is no dominance relation between g and h despite g having a reference to h. The dependence of g on h, in fact, is not captured by the dominator tree.

We define a *horizontal* dependence as a reference to a function that is not an ancestor or a descendant in the dominator tree (i.e. a reference to a sibling or an uncle in the parse tree). The free variables of a horizontal dependence must also be propagated from the callee to the caller. Since horizontal dependencies are not captured by the dominator tree of a call graph, a dominator tree must be augmented into a graph to capture horizontal dependencies.

To convert a dominator tree into a graph that captures horizontal dependencies, the dominator tree is augmented with the edges between functions in the call graph that do **not** have a vertical dependence. We call this graph an EDT (**E**xtended **D**ominator **T**ree) graph and the new edges are called lateral edges. If the resulting EDT graph does not contain any cycles then it only has *simple* horizontal dependencies. Otherwise, it has *complex* horizontal dependencies. Clearly, the EDT graph for a program that only has functions with vertical dependencies is its annotated dominator tree.

First, we highlight some important properties of EDT graphs. Second, we extend our lambda lifting algorithm to handle the class of programs that have simple horizontal dependencies. Finally, we extend our lambda lifting algorithm to handle arbitrary programs that may contain complex horizontal dependencies.

6.1 Important Properties of EDT Graphs

Formally, the set of lateral edges, E_l, in an EDT graph formed from the dominator tree, DT, of a call graph, CG, is defined as:

```
(define (f x)
  (define (g ...) (...x...a...b...c...d...))
  (define (a ...) (...b...))
  (define (b ... (...c...d...))
  (define (c ...) (...g...))
  (define (d ...) (...))
 (...g...))
```

Fig. 15. Fifth Scheme Pseudo-code

$$E_l = \{(f, g) \in CG \mid f \text{ and } g \text{ do not have a dominance relation}\}.$$

The set E_l endows the EDT graph with important properties outlined by the following theorems. After establishing the validity of these properties we will point out their significance for lambda lifting.

Theorem 2. *If (f,g) $\in E_l$, then the parent of g, p_g, in the dominator tree, DT, dominates f.*

Proof. *Let G be the EDT graph obtained by only extending DT with the lateral edge from f to g and let r be the root function of DT. If there is a path in G from r to g that contains f and that does not contain p_g, then p_g does not dominate all paths from r to g. This means that DT can not be the dominator tree which contradicts our assumption. Q.E.D.*

Having established that the parent of the called function for a lateral edge in the EDT graph dominates the caller, we can now establish that all the ancestors of the called function dominate the caller. The proof simply exploits the fact that domination is a transitive property.

Theorem 3. *If (f,g) $\in E_l$, then all ancestors of g in the dominator tree, DT, dominate f.*

Proof. *Theorem 2 establishes that the parent of g dominates f. All other ancestors of g dominate its parent. Therefore, all of g's ancestors dominate f. Q.E.D.*

The significance of Theorems 2 and 3 for lambda lifting is that the existence of a lateral edge from f to g in an EDT graph means that LD_g, if it exists, dominates f. Therefore, LD_g may also be LD_f. This occurs when none of the nodes in the dominator tree path from the parent of g to f are LD_f. In addition to free variables, LD information must be propagated from callees to callers across lateral edges.

6.2 Simple Horizontal Dependencies

When an EDT graph only has simple horizontal dependencies (i.e. there are no strongly connected compinents in the EDT graph) it suffices to first propagate free variables and lowest dependence information between functions using

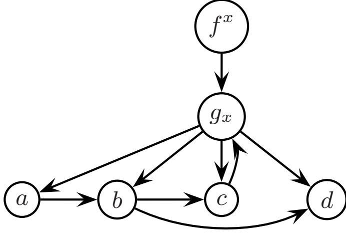

Fig. 16. Call Graph

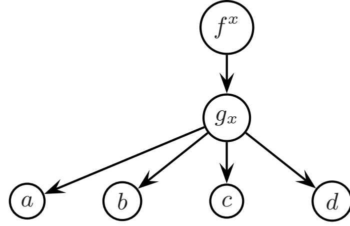

Fig. 17. Dominator Tree

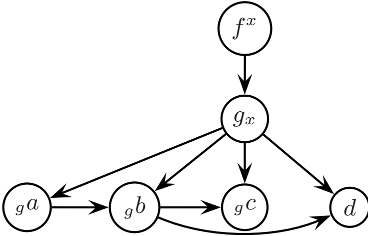

Fig. 18. After Depth-First Propagation

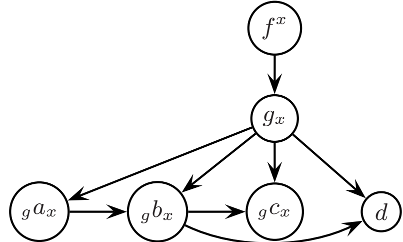

Fig. 19. After Breadth-First Propagation

a depth-first traversal (akin to propagating up the dominator tree) and then to propagate free variables in breadth-first order exploiting lowest dependence information (akin to propagating down the dominator tree). The correctness of the second propagation follows from observing that free variables are propagated from callees to callers and from Theorem 3 that guarantees lowest dependence information can safely be propagated across lateral edges.

To illustrate the use of horizontal dependence information in the absence of strongly connected components consider the pseudo-code in Figure 15 and its call graph in Figure 16. The dominator tree for this graph is displayed in Figure 17. Extending the dominator tree with edges between functions that do not have a vertical dependence results in the original call graph without the edge from c to g. Figure 18 displays the results of propagating free variables and LD information after a depth-first traversal. The node representing c has no successors and, therefore, LD_c is g (the lowest ancestor it references). No free variables propagate between the functions in this step, but LD_c, g, propagates to become LD_b and LD_a. Figure 19 displays the results of propagating free variables in a breadth-first order by exploiting LD information. Each function receives the free variables of its LD function. That is, a, b, and c receive x.

6.3 Complex Horizontal Dependencies

The augmentation of the dominator tree, however, may lead to an EDT graph that is no longer acyclic. That is, the resulting graph may contain strongly connected components. This occurs, for example, when two siblings in the dominator

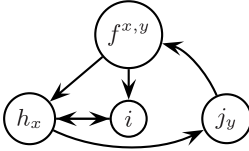

Fig. 20. Call Graph

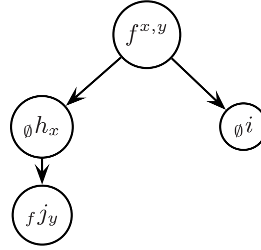

Fig. 21. Dominator Tree

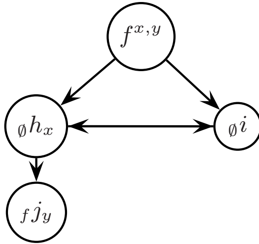

Fig. 22. EDT Graph

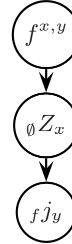

Fig. 23. Coalesced EDT Graph

tree are mutually recursive. In the presence of strongly connected components, it no longer suffices to simply propagate free variables and LD information using a depth-first traversal. The problem is that such a traversal does not guarantee that all successors of a node are processed first.

Strongly connected components must be coalesced, but as learned from the MM-algorithm sets of functions that include a dominating function and the functions it dominates should not be coalesced. That is, functions that have a vertical dependence should not be coalesced. This observation suggests that within a strongly connected component only functions at the same level in the dominator tree can be coalesced together. Notice that a function at level n in the dominator tree can not declare any variables that are free in other functions at level n. This means that they do not have a dominance relation and it is safe to coalesce these functions together, because none of these functions will unnecessarily contribute free variables to each other.

The goal, therefore, is to coalesce strongly connected components in an EDT graph without loosing vertical dependence information. To achieve this it is helpful to distinguish between two types of edges in an EDT graph. The first kind of edge is a *simple lateral edge* which occurs between functions at the same level of the dominator tree (i.e. edges between siblings in the dominator tree). Any strongly connected components formed solely by simple lateral edges can be coalesced in the EDT graph, because among the siblings in each component there is no dominating function. If an EDT graph is created by solely adding simple lateral edges to the dominator tree, then after coalescing strongly connected

components the EDT graph is acyclic. Thus, lambda lifting can proceed as described in section 6.2 by making a coalesced node's free variables the union of the free variables of the functions in the strongly connected component and by making the node's LD function be the $max_f(LD_g)$, where g is a function in the strongly connected component. To illustrate this concept consider the call graph in Figure 20. Its dominator tree, displayed in Figure 21, reflects the known facts after its creation: i and h have no known upward vertical dependencies, LD_j is f, x is free in h, and y is free in j. The EDT graph, displayed in Figure 22, is created by adding the two lateral edges between i and j in the dominator tree. The strongly connected component formed by $\{h, i\}$ is coalesced into a node, say, Z. The set of free variables of Z is $\{x\}$ and LD_Z is $\emptyset$. The result of this transformation is displayed in Figure 23. After the depth-first propagation the set of free variables of Z is $\{x, y\}$ and $LD_Z = f$. Nothing propagates during the breadth-first propagation (because f has no free variables). After the propagation steps, we have that $\{x, y\}$ are the required free variables for h and i which is precisely what is needed.

The second kind of edge is an *upward lateral edge* which exists between functions at different levels of the dominator tree. These edges always occur from a node for a function, g, at level n to a node for function, f, at level $n - i$, where $i \geq 1$, such that f is not an ancestor of g in the dominator tree[3]. The existence of such an edge, means that g needs the free variables of f. Notice, however, that f may not need all of g's free variables despite being in the same strongly connected component. The free variables of g not needed by f are those that are local to the strongly connected component and that are not lexically visible nor declared by f. All of these variables must be declared by a function with a depth greater than or equal to the depth of f in the dominator tree.

Notice that the set of functions in the strongly connected component may include siblings of f in the dominator tree. The incoming upward lateral edge to f means that these siblings need the same free variables. This follows from observing that they all need as free variables the variables declared by common ancestors in the dominator tree that are free in the strongly connected component. Therefore, we have that the siblings of a function in the dominator tree, like f that has an incoming upward lateral edge, that are in the same strongly connected component can be coalesced with f without local free variables being unnecessarily propagated during lambda lifting. Coalescing only siblings in a strongly connected component preserves vertical dependence information and provides a directed acyclic graph that can be used to compute the free variables needed by each function in an arbitrary program.

To illustrate the use of horizontal dependence information in the presence of strongly connected components created by upward lateral edges consider the pseudo-code in Figure 24 and its call graph in Figure 25. Its dominator tree is displayed in Figure 26. The first step is to extend the dominator tree with simple lateral edges. The resulting graph is displayed in Figure 27. Five simple

[3] There can not exist any edges in the other (i.e. downward) direction from f to g in a dominator tree.

```
(define (f x)
  (define (g ...) (...h...))
  (define (h c) (...j...k...)))
    (define (j ...) (k...c...l...m...)))
    (define (k a b) (...j...n...o...)))
      (define (n ...) (...a...o...)))
      (define (o ...) (...b...i...)))
  (define (i ...) (...h...))
  (define (l ...) (...g...))
  (define (m ...) (...x...))
 (...g...h...i...)))
```

Fig. 24. Sixth Scheme Pseudo-code.

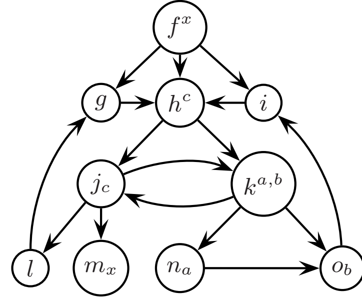

Fig. 25. Call Graph

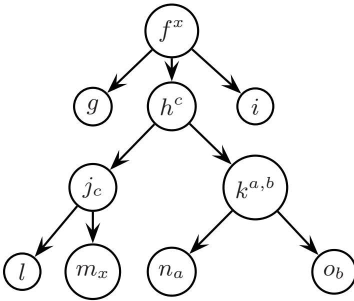

Fig. 26. Dominator Tree

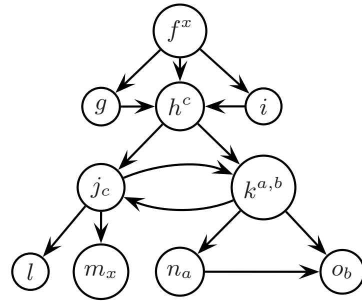

Fig. 27. DT with Simple Lateral Edges

lateral edges have been added to extend the dominator tree. These additions have formed a strongly connected component that contains the functions j and k. These functions are coalesced to form a new node S. The set of free variables for S is obtained from the union of the free variables of j and k. The resulting graph is displayed in Figure 28. The graph in Figure 28 is now extended with upward lateral edges. If a function on either side of an edge has been coalesced then the coalesced node replaces the function. The result of this extension adds edges from l to g and from o to i. The result is displayed in Figure 29. This graph now has a strongly connected component formed by $\{g, h, i, S, l, n, o\}$. Function g has an incoming upward lateral edge and, therefore, it is coalesced with its siblings h and i that are also members of the strongly connected component. Given that i, a function with an incoming lateral edge, has been coalesced there is no need for further action with it. No other functions have an incoming upward lateral edge which means the graph is now acyclic. The finalized EDT graph is displayed in Figure 30 in which Q represents the coalesced functions $\{g, h, i\}$. This graph can now be used to propagate free variables and LD information as done in section 6.2.

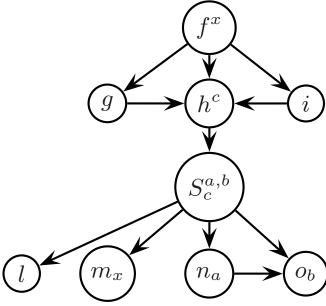

Fig. 28. Graph After First Coalescing Step

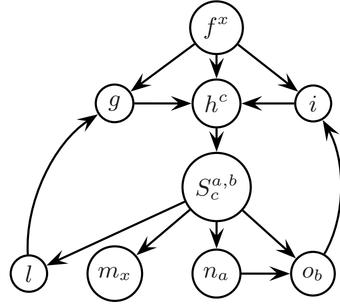

Fig. 29. Added Upward Lateral Edges

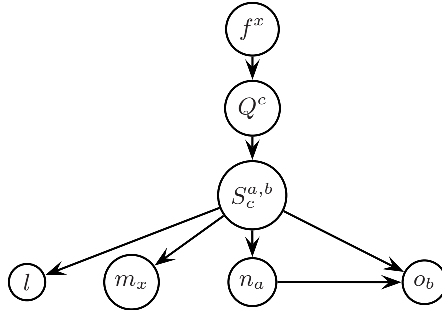

Fig. 30. Completed EDT Graph After Second Coalescing Step

7 The Algorithm, Complexity, and Correctness

7.1 The New Lambda Lifting Algorithm

The new lambda lifting algorithm builds a directed acyclic EDT graph from the call graph of a program. Propagation of free variables then proceeds in two steps: the first using a depth-first traversal and the second using a breadth-first traversal. The steps in the algorithm can be outlined as follows:

1. Build the call graph, CG, of the program from its parse tree.
2. Build the dominator tree, DT, for CG.
3. Extend DT with simple lateral edges and coalesce strongly connected components to obtain an acyclic graph EDT'.
4. Extend EDT' with upward lateral edges and compute strongly connected components. Coalesce functions that have an incoming upward lateral edge with their dominator tree siblings that are members of the same component. The resulting graph is the directed acyclic EDT'' graph.
5. Use EDT'' to propagate free variables and LD information using a depth-first traversal.
6. Use EDT'' to propagate free variables using LD information using a breadth-first traversal.

7. For each function, f, make f scope insensitive by adding its complete set of free variables as parameters to f and as arguments to each reference to f.
8. Remove block structure by floating each function to the global level.

7.2 Complexity and Correctness

For a program P, let i be the number of functions, let e be the number of function calls, let v be the number of variables declared, and let n be the size of the program (i.e. $i + e + v$). Step 1 is proportional to $O(e + i)$ or simply $O(n)$. Step 2 is $O(n)$ [16]. For steps 3 and 4 extending a graph with edges is $O(e+i)$ or simply $O(n)$. The computation of strongly connected components is $O(n)$ [5,6] and their coalescing is $O(n^2)$. For step 5, the propagation of free variables and LD information is $O(e * (i+v))$, or simply $O(n^2)$, assuming the union operation is done in linear time. A similar line of reasoning holds for step 6. Step 7 is $O(v + i + e)$ or simply $O(n)$. Finally, step 8 is $O(n)$. This means that optimal lambda lifting is done in $O(n^2)$. Since lambda lifting can generate an output program of size $O(n^2)$, the time complexity of this algorithm is optimal [3].

We have not formally proven the correctness of the presented algorithm and we only argue informally for its correctness. The correctness of the algorithm hinges on correctly computing the set of required variables for each function. The required set of free variables for a function, f, depends on the free variables f directly references and on a subset of the free variables transitively needed by the functions f calls. The computation of the latter subset is achieved by never coalescing a dominating function with any functions it dominates. This leads to a graph in which the breadth-first propagation in Step 6 completes the computation of the required free variables of any ancestor function in the parse tree before any successor function as done in the J-algorithm and prevents free variables local to a strongly connected component to be unnecessarily propagated. The required set of free variables computed for each function is complete, because all functional dependencies are captured by the EDT graph. Lateral and downward vertical dependencies are captured by edges and upward vertical dependencies are captured by LD information.

8 Concluding Remarks

This article presents an optimal graph-based $O(n^2)$ lambda lifting algorithm. The algorithm is optimal in the sense that it computes the minimal set of free variables required by each function to make them scope insensitive. The algorithm is also asymptotically optimal, because a lambda lifted program of size $O(n^2)$ is computed in $O(n^2)$ steps. The new algorithm is superior to Johnsson's and to Morazán's and Mucha's algorithms by reducing the complexity of optimal lambda lifting from $O(n^3)$ to $O(n^2)$ and it is superior to Danvy's and Schultz's algorithm by being optimal. Nonetheless, this new algorithm owes a great deal of its creation to these predecessors. Considering that Johnsson's original algorithm first appeared in 1985, this newest algorithm has been over 20 years in the

making. It is, indeed, a tribute to all these algorithms and to the work of the cited authors from whom we borrowed ideas and inspiration.

Free variables arise in programs due to the nesting of function definitions. This suggests that identifying free variables may rely heavily on lexical analysis. The algorithm presented, however, only relies on lexical analysis to identify the free variables directly referenced by each function and the function dependencies of each function. Once this lexical information is known, the algorithm builds and relies on non-lexical elements such as the call graph, the dominator tree, and sets of free variables. In essence, lambda lifting is not solely an exercise in lexical analysis.

As part of our future work we are interested in testing the runtime efficiency of the different algorithms to determine their impact on compilation time. Although compilers only spend a small amount of time on lambda lifting, such testing will determine the practical impact of this work. Future work also includes the implementation of a closureless functional language that uses applicative-order evaluation. The main idea behind the design of this new langauge is to dynamically generate functions that are specialized based on the bindings of its free variables instead of allocating closures [13]. Lambda lifting identifies for us the variables that are used to specialize functions.

Acknowledgements

The authors thank Olivier Danvy, Sven-Bodo Scholz, and Barbara Mucha for the discussions during and after IFL 2005 that initiated us down the path that lead to the new algorithm presented in this article. Marco T. Morazán also thanks TLTC at Seton Hall University for the support received through a Faculty Innovation Grant.

References

1. Consel, C.: A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages. In: Proc. of the Symp. on Partial Evaluation and Semantics-Based Program Manipulation, June 1993, pp. 145–154. ACM Press, New York (1993)
2. Danvy, O., Schultz, U.P.: Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure. Theoretical Computer Science 248(1–2), 243–287 (2000)
3. Danvy, O., Schultz, U.P.: Lambda-Lifting in Quadratic Time. Journal of Functional and Logic Programming 2004(1) (July 2004)
4. Friedman, D.P., Wand, M., Haynes, C.T.: Essentials of Programming Languages. The MIT Press, Cambridge (2001)
5. Gibbons, A.: Algorithmic Graph Theory. Cambridge University Press, Cambridge (1985)
6. Gould, R.: Graph Theory. The Benjamin/Cummings Publishing Company, Inc. (1988)
7. Matthews, J., Findler, R., Graunke, P., Krishnamurthi, S., Felleisen, M.: Automatically Restructuring Programs for the Web. Automated Software Engineering 11(4), 337–364 (2004)

8. Johnsson, T.: Lambda Lifting: Transforming Programs to Recursive Equations. In: Proc. of a Conf. on Functional Prog. Lang. and Comp. Arch., pp. 190–203. Springer, New York (1985)
9. Johnsson, T.: Target Code Generation from G-Machine Code. In: Fasel, J.H., Keller, R.M. (eds.) Graph Reduction: Proceedings of a Workshop at Santa Fé, New Mexico, New York, NY, pp. 119–159. Springer, Heidelberg (1987)
10. Jones, S.L.P.: The Implementation of Functional Programming Languages. Prentice-Hall International Series in Computer Science. Prentice-Hall, Upper Saddle River (1987)
11. Jones, S.L.P., Lester, D.: A Modular Fully-lazy Lambda Lifter in HASKELL. Software - Practice and Experience 21(5), 479–506 (1991)
12. Jones, S.L.P., Lester, D.: Implementing Functional Languages: A Tutorial. Prentice Hall International Series in Computer Science (1992)
13. Morazán, M.T.: Towards Closureless Functional Languages. In: Arabnia, H. (ed.) Proc. of the Int. Conf. on Prog. Lang. and Compilers, pp. 57–63. CSREA Press (2005)
14. Morazán, M.T., Mucha, B.: Improved Graph-Based Lambda Lifting. In: Arabnia, H. (ed.) Proc. of the Int. Conf. on Prog. Lang. and Compilers, June 2006, pp. 896–902. CSREA Press (2006)
15. Oliva, D.P., Ramsdell, J.D., Wand, M.: The VLISP Verified PreScheme Compiler. Lisp and Symbolic Computation 8(1-2), 111–182 (1995)
16. Alstrup, S., Harel, D., Lauridsen, P.W., Thorup, M.: Dominators in Linear Time. SIAM Journal on Computing 28(6), 2117–2132 (1999)

Appendix

Tree. A *tree* is a connected, directed, and acyclic graph in which there is a node, called the *root*, such that there is a path from the root to every other node in the graph. The root node has no incoming edges and all other nodes have only one incoming edge. When an edge from node A to node B exists, we say that A is the parent of B. The *depth* or *level* of a node N in a tree rooted at R is the number of edges in the path from R to N. The *ancestors* of N are all the nodes on the path from R to the parent of N. The *descendants* of N are all the nodes that are reachable from the subtree rooted at N.

Dominator Tree. In a graph G with root node R, a node N *dominates* a node M if every path from R to M must pass through N. The *immediate dominator* of a node M is a node N if $\nexists K \in G$: N dominates K and K dominates M. The *dominator tree* T of G contains the same set of nodes as G and has an edge from a node N to a node M when N is the immediate dominator of M.

Strongly Connected Component. The nodes $N_1 \ldots N_k$ of a directed graph G form a *strongly connected component* if for every pair of distinct nodes, N_i and N_j, there is a path from N_i to N_j and a path from N_j and N_i.