

Lambda-Lifting Scheme Programs

Barbara Mucha and Marco T. Morazán
Seton Hall University
Department of Mathematics and Computer Science
400 South Orange Avenue
South Orange, NJ 07079 USA
E-mail: {muchabar,morazanm}@shu.edu

ABSTRACT

Lambda-lifting is a program transformation technique to elevate nested function definitions, with or without free variables, to a global level. Elevating all functions to a global level means that at runtime closures do not have to be created to store the free variables of a function. Lambda-lifting has been used, for example, in ML to eliminate functions with free variables. The technique used for ML works well, because in ML all functions are curried allowing for a proper subset of the arguments to a function to be passed when it is evaluated. In a language such as Scheme, however, functions are not curried and, therefore, we can not pass in a proper subset of the arguments to a function when it is evaluated. Lambda-lifting, however, can be adapted for Scheme to elevate all functions to the global level. Free variables are not completely eliminated, but their existence is limited to the body of non-nested Scheme lambda expressions. The allocation of closures at runtime to store free variables can then be avoided by using partial evaluation to dynamically create specialized functions when the bindings of free variables are known. That is, lambda-lifting allows for the efficient application of a partial evaluator to a lambda expression with free variables and to the bindings of its free variables. The idea is similar to curried functions in ML that create at runtime new functions when only a subset of their input is passed in. In this article we briefly describe how runtime code generation can eliminate the need for closures and present two tail-recursive implementations of a lambda-lifter for a pure subset of Scheme along with reasons why to choose one implementation over the other.

1. INTRODUCTION

Functional languages support and encourage the use of first class/higher-order functions meaning that functions can be passed in as arguments to functions and can be returned as values from functions. To support first-class functions, many functional languages create closures. A closure is a data structure consisting of a function to be evaluated and

an environment storing the bindings of the free variables in the function. To implement closures, either activation records must be heap-allocated allowing closures to point to those records that hold the bindings of the free variables[3] or heap-allocated closure data structures that directly store the binding of the free variables can be created[2]. In both of these cases, closures are heap-allocated forcing additional heap accesses to determine the binding of a free variable. These additional heap accesses are an unfortunate overhead, because when the closure's function is evaluated the bindings of the free variables are known. To eliminate the need to allocate closures and access the heap for the bindings of free variables in Scheme, partial evaluation can be applied to a function and to the bindings of its free variables to create a specialized function in which references to free variables have been eliminated.

Consider the following example at the Scheme level of abstraction:

```
(define (f y) (lambda (x) (+ x y)))
```

In the scope of the returned function by f , y is a free variable. When f is evaluated, a closure can be allocated to store the binding of y and a reference to the one-input function $(\text{lambda } (x) (+ x y))$. Applying the closure's function to an argument requires accessing the heap for the value of y .

Instead of creating a closure for the function returned by a call to f , partial evaluation can be used to return a specialized function. This implementation strategy has, for example, $(f\ 5)$ return the function Fresult defined as:

```
(define (Fresult x) (+ x 5))
```

The returned function, in this example, does not contain references to free variables. That is, it is a combinator that can be applied to an argument just as any user-defined function that does not contain free variables. In this example, unfortunately, the original function, f , must always be processed to create a specialized function.

To efficiently apply partial evaluation for code generation and closure elimination at runtime, lambda-lifting can be used to raise all anonymous functions (i.e. lambda expressions) and nested functions to the global level. In this manner, the original function does not have to be processed and

only lambda expressions with free variables need to be specialized. For the above example, a lambda-lifter creates a new function at compile time and transforms the original f as follows:

```
(define (f y) (lam1 y))
(define (lam1 y) (lambda (x) (+ x y)))
```

The evaluation of $lam1$ will return a function equivalent to $Fresult$ by substituting the reference to y in the body of the lambda expression with y 's binding. That is, both approaches are semantically equivalent. In this form, however, $lam1$ can be implemented as a specialized code generator for $(lambda (x) (+ x y))$ that does not have to process the original expression for f each time. Instead, references to free variables are limited to the body of a single lambda expression. The partial evaluator can then reduce to normal form subexpressions that depend solely on the values of free variables.

In this article, we describe closures, partial evaluation, lambda lifting for ML, and present two tail-recursive implementations of a lambda-lifter for a pure subset of Scheme along with reasons why to choose one implementation over the other. The tail-recursive lambda lifter we have chosen is being implemented as part of the MT system to support first-class functions and to avoid disrupting the expected access patterns of the MT heap. It is important that the lambda-lifter be tail-recursive in order to efficiently implement it for the MT evaluator machine in C++. A recursive lambda-lifter would allocate too much stack memory in C++ and result in slow execution.

2. RELATED WORK

2.1 Closures

In programming languages, a closure is a special data structure that contains a pointer to a function and a representation of the function's lexical environment at the time when the closure was created[8]. Closures typically appear in languages that allow functions to be first-class like Scheme. Scheme was the first programming language to have fully generated lexically scoped closures. Virtually all functional programming languages, support closures (e.g. Haskell, Lisp, and all variants of ML). In addition, some object-oriented languages, like Smalltalk, enable the programmer to use objects to simulate some features of closures.

There are two standard ways of implementing closures. The first is called *linked closures* in which heap allocated activation records are kept in a linked list to preserve the bindings of free variables[3, 1]. Linked closures are fast to build, but slow to access. The second is called *flat closures* in which the bindings of free variables are copied to one block of memory in the heap[2, 7]. Flat closures are slower to build, but faster to access.

Modern compilers often implement function calls in two steps. First, a closure environment is properly installed to provide access for free variables in the target program fragment. Second, the control is transferred to the target[12]. The crucial compilation decision for functional languages regarding closures is where to store and how to represent closures at runtime.

2.2 Partial Evaluation

Partial evaluation is commonly attempted to improve the efficiency of a program while preserving its meaning. Partial evaluation is a program transformation technique which generates a specialized version of a program given some of the program's input[5]. The specialized program returned by a partial evaluator is called a residual program. If a program, P takes as input a set of values, $x_1 \dots x_n$, a partial evaluator that is given P and some of P 's input, $x_1 \dots x_i$, will return a specialized version of P , P_{res} , that when given the rest of P 's input, $x_{i+1} \dots x_n$, will return the same answer as P . That is, $P(x_1 \dots x_n) = P_{res}(x_i \dots x_n)$.

Partial evaluation in functional languages has been used, for example, in the *Fabius* compiler to implement runtime code generation for a pure subset of ML that does not include higher-order functions[11, 10]. The technique used goes beyond simply substituting values in the sequence of instructions of a program. Instead, specialized code generators are created that do not need to process the original sequence of instructions when code is generated dynamically. More recently, Dynamic Caml has extended Objective Caml with a library to enable dynamic code generation[6]. Dynamic Caml employs specialized code generators and allows, unlike Fabius, the use of higher-order functions. Empirical measurements using Dynamic Caml suggest that real speed-ups are achieved with dynamic code generation. Closures, however, are still created (and required) in Dynamic Caml despite generating specialized code dynamically at runtime.

3. LAMBDA LIFTING

Lambda lifting is a technique for transforming a functional program with local function definitions into a program containing only global function definitions. The resulting program only contains functions without free variables (also known as combinators). Lambda lifting has been used, for example, in the Lazy ML compiler[4].

Much of the work performed by a lambda-lifter is associated with the elimination of free variables from function definitions. The elimination of free variables in ML from a function, f , is achieved by introducing into the formal parameter list of f its free variables and by adding as actual parameters to every call/reference to f the expressions that yield the bindings of the free variables. Introducing new parameters to a function, f , and new arguments to applications of f in a function g , of course, means that new free variables may be introduced into g . Therefore, the parameters added to g are the union of its free variables and the free variables of f . Consider the following expression written using Scheme-like syntax:

```
(lambda (x)
  (lambda(y)
    ((lambda(z) (+ x y z)) (+ 5 y))))
```

We can name each of the functions defined by Scheme's special form $lambda$ ¹ to make the expression more readable. The resulting definitions are:

¹The special form lambda is Scheme's notion of functions.

```
(define (f x)
  (define (g y)
    (define (h z)
      (+ x y z))
    (h (+ 5 y))))
g).
```

The functions f and g are combinators, because their bodies do not contain any free variables. The function h is not a combinator, because x and y are free variables in its body. To lift h to the global level means that its free variables, x and y , must be added to its formal parameters list and that arguments must be added to the call to h in the body of g . Therefore, lambda lifting h yields the following definitions:

```
(define (lifted-h x y z) (+ x y z))
(define (f x)
  (define (g y)
    (lifted-h x y (+ 5 y))))
g).
```

In the above definitions, *lifted-h* is a combinator. However, g is no longer a combinator, because x has been introduced as a free variable in its body. Lifting g now requires adding x to its formal parameter list and, consequently, an actual parameter must be added wherever g is referenced. Lambda lifting g yields to the following definitions which are all combinators:

```
(define (lifted-h x y z) (+ x y z))
(define (lifted-g x y) (lifted-h x y (+ 5 y)))
(define (f x)
  (lifted-g x)).
```

Notice that the call to *lifted-g* in the body of f contains fewer arguments than formal parameters. In the semantics of ML, where all functions are *curried*², it is permissible to pass into *lifted-g* only a subset of the input it requires. ML, therefore, supports partial evaluation of functions. In Scheme, however, functions are not curried. Thus, it is not possible to pass only one argument to *lifted-g*. It is, therefore, necessary to adapt the lambda-lifting transformation for the semantics of Scheme in order to support partial evaluation and eliminate the creation of heap-allocated closures.

4. LAMBDA LIFTING FOR SCHEME-LIKE LANGUAGES

You can think of lambda expressions as anonymous functions. That is, functions that lack a name. Just like known functions (i.e. functions that have a name), a lambda expression has a list of parameters and a block of code specifying the operation to be performed on the parameters. There are two kinds of lambda expressions: lambda expressions without free variables (i.e. combinators) and lambda expressions with free variables. In this section, we illustrate how to lambda lift each of these types of lambda expressions.

4.1 Lambda-Lifting Combinators

We illustrate how to lift combinators, defined by Scheme lambda expressions, to transform them to known functions.

²Curried functions consume their input lazily.

Consider the following expression that contains a lambda subexpression:

```
((lambda (x) (+ x 1)) (f 4))
```

The lifting of a combinator creates a new uniquely named function whose parameters are the parameters of the lambda expression. In the original expression, the lambda expression is substituted by the unique name that is created for it. The example above is transformed to:

```
(define (UniqueName x) (+ x 1))
(UniqueName (f 4))
```

The expression `(lambda (x) (+ x 1))` is transformed to a known function called *UniqueName* that has a parameter x . In the original expression the lambda form is substituted with the new name *UniqueName*.

During program execution, there is no overhead for closure allocation in this form. The argument, `(f 4)`, can be passed into *UniqueName* by using a stack. Semantically, both forms are equivalent since they pass the argument `(f 4)` to a function that adds 1 to its input.

4.2 Lambda-Lifting Expressions with Free Variables

A lambda expression with free variables is a function whose body has references to variables that are not declared in its formal parameter list. To avoid heap allocating closures for a lambda expression with free variables in Scheme, the lambda expression needs to be lifted to create a uniquely named function. The parameters to the new function are the free variables in the body of the lambda expression. The body of the new function is the lambda expression itself.

We illustrate our lambda lifting algorithm with the same example used in the previous section:

```
(lambda (x)
  (lambda(y)
    ((lambda(z) (+ x y z)) (+ 5 y))))
```

In the body of the innermost lambda expression x and y are free variables and they appear in the parameter list of a newly created function, *lam1*, after lifting the innermost lambda expression. The innermost lambda expression is substituted with a call to *lam1* with arguments x and y in the original expression. The result is displayed as follows:

```
(define (lam1 x y) (lambda (z) (+ x y z)))
(lambda (x)
  (lambda(y)
    ((lam1 x y) (+ 5 y))))
```

In this form, x is a free variable in the nested lambda expression. We apply the same transformation to lift the nested lambda expression to yield:

```
(define (lam2 x) (lambda (y) ((lam1 x y) (+ 5 y))))
(define (lam1 x y) (lambda (z) (+ x y z)))
(lambda (x)
  (lam2 x))
```

The original expression has now been transformed to a combinator expressed using a lambda expression. This lambda expression can be transformed using the techniques illustrated in the previous subsection to yield the original expression as *lam3*. The final result is:

```
(define (lam3 x) (lam2 x))
(define (lam2 x) (lambda (y) ((lam1 x y) (+ 5 y))))
(define (lam1 x y) (lambda (z) (+ x y z)))
lam3.
```

In the scope of the functions returned by *lam1* and *lam2*, there are free variables. This is unavoidable in Scheme, because functions are not curried. We can observe, however, that in the above form it is not necessary to heap allocate a closure at runtime when the lambda expressions are evaluated. Instead, specialized code generators created at compile time can dynamically create a new function for each of the lambda expressions by substituting references to free variables with their values and, perhaps, performing some computations that depend solely on the values of free variables. The amount of computation to be done is to be determined empirically. In the above example, there are no subexpressions that depend solely on the values of free variables. Therefore, the partial evaluator would not perform any computations beyond the generation of the new functions at runtime.

The technique illustrated is semantically similar to that of curried functions in ML. In ML, the free variables are passed into a function that consumes them lazily. In our proposed implementation strategy, instead of passing the bindings of free variables to a function that consumes them lazily, we pass them into a code generator that adds at runtime new functions that are specialized for each lambda expression in a program after having been lambda lifted.

5. DESIGN OF LAMBDA LIFTING FOR SCHEME PROGRAMS IN THE MT SYSTEM

In this section, we describe the parameters of the MT lambda lifter for a pure subset of Scheme and present a BNF grammar for the data structure that governs the behavior of the lambda lifter.

5.1 Design Roles of the Parameters

The MT lambda lifter has 6 variables whose design roles can be described as follows:

- **RES:** *RES* is a list which stores the result obtained so far from lambda lifting the current expression. The lambda lifted part of the current expression is stored in *RES* while the part that has not yet been lifted is stored in the variable *STACK*.
- **NEWFS:** *NEWFS* is a list of all the new function definitions created by the lambda lifter.
- **D:** *D* is a counter for the number of new functions that have been created by the lambda lifter. It is used to guarantee that all new functions generated have a unique name.

- **STACK:** *STACK* is a list used to implement a stack and stores expressions that have not yet been lambda lifted as well as information that will be needed to construct a result after an expression has been lambda lifted. The element on the top of the stack dictates the actions taken in the current iteration of the lambda lifter. A BNF grammar describing *STACK* is presented later in this section.
- **ENV:** *ENV* is a list that stores the parameters, if any, of the current function (i.e. lambda expression). If the expression being lifted is not a subexpression of a lambda expression, then *ENV* is empty.
- **KNOWN:** *KNOWN* is a list that stores different information depending on which of the two algorithms presented here we are referring to. In Algorithm I, *KNOWN* a list of all the primitives and known functions at the time an expression is lifted. In Algorithm II, *KNOWN* is a list of all the parameter declarations at the time an expression is lifted.

5.2 STACK Grammar

STACK stores all expressions that need to be lifted, the result of lifting previous subexpressions, and the environment of the current expression being lifted. The actions taken by the lambda-lifter is determined by the top element of the stack and are described in the next section. *STACK* can be described by the BNF grammar in Figure 1. Elements in angled brackets represent nonterminals, * represents Kleene star over a given element (i.e. 0 or more instances), + represents 1 or more instances of a given element, symbols not surrounded by brackets represent literal values, elements in parenthesis represent a list, and a period within a list indicates that the list does not end with the empty list.

During execution, the type of *STACK* is always distinguishable by the topmost element. The meaning of the topmost element is described as follows:

- **empty-list:** *empty-list* means that the stack is empty and the lambda lifter has completed its job.
- **number or symbol:** *number or symbol* means that the next subexpression to be lifted is a number or a symbol respectively.
- **lambda-expr:** *lambda-expr* means the next subexpression to be lifted is a lambda expression. A lambda expression is a list containing the symbol *lambda*, a list of identifiers, and a list representing the body of the lambda expression.
- **define-expr:** *define-expr* means the next subexpression to be lifted is a *define* expression represented as a list containing the symbol *define*, an identifier or a list of identifiers, and a list representing the body of the definition.
- **env-list:** *env-list* means that the top of the stack is a list containing the symbol *env* and a list of declared variables which can be empty.

<i>Stack</i>	
<S>	→ <empty-list>
	→ (<number><S>)
	→ (<symbol><S>)
	→ (<lambda-expr><S>)
	→ (<define-expr><S>)
	→ (<env-list><S>)
	→ (<result-list><S>)
	→ (<empty-list><S>)
	→ (<app-expr><S>)
<number>	→ <digit><digit>*
<digit>	→ 0 1 2 3 4 5 6 7 8 9
<lambda-expr>	→ (lambda (<ids> *) <expr>)
<expr>	→ <ids>
	→ (lambda (<ids>*)<expr>)
	→ (<expr>+)
<ids>	→ <symbol>
<define-expr>	→ (define <ids><expr>)
	→ (define (<ids>*) <expr>)
<env-list>	→ (env <ids>*)
<result-list>	→ (res <res>)
	→ (res . <res>)
<empty-list>	→ ()
<app-expr>	→ (<number><S>)
	→ (<symbol><S>)
	→ (<lambda-expr><S>)
	→ (<define-expr><S>)
	→ (<app-expr><S>)

Figure 1: BNF Grammar for STACK

- **res-list:** *res-list* means that the top of the stack is a list containing the symbol *res* and a saved value of the *RES* parameter. This saved value is the result obtained from lambda lifting previous subexpressions of the current expression.
- **empty-list:** *empty-list* means that an expression representing a function application without arguments is an argument to a function.
- **app-expr:** *app-expr* means that the top of the stack is an application expression that needs to be lambda lifted.

6. ALGORITHM I

6.1 Design Description

The lambda lifter must determine when a variable occurs free within an expression. In order to accomplish this, Algorithm I keeps a list, *KNOWN*, containing all the primitives of the MT Evaluator Virtual Machine and all the known functions. The known functions include the functions that have been user defined and all the new functions that have been created by the lambda lifter at the time when an expression is lifted.

To start lifting an expression, the expression is pushed onto the stack. The lambda lifter then iterates until the STACK is the empty list. At each iteration, the top of the stack is examined and an action is taken changing the state of the parameters. We describe the actions taken based on the type of stack as described in Figure 1:

- **<empty-list>:** The lambda lifter is done and a list containing *RES* and *NEWFS* is returned. This returned list contains the lambda lifted expression (*RES*) and the new functions created by the lambda lifter (*NEWFS*).
- **<number>:** A number is popped off the STACK and added to *RES* as it is the result of lambda lifting a literal number expression. Since a number does not contain free variables the result of lambda lifting it is the number itself.
- **<symbol>:** A symbol is popped off the STACK and added to *RES*. Determining whether or not the variable is free is done later when the result of a lambda-lifted lambda expression is constructed. If the symbol is not part of a lambda expression (e.g. in the case of *define* or *if*), the result obtained from lambda lifting the symbol is the symbol itself.
- **<lambda-expr>:** The lambda expression is popped off the STACK. *ENV* (which contains all declared variables that are lexically valid for the lambda expression), *RES*, and the body of the lambda expression are pushed on the STACK. *ENV* becomes the declarations of the lambda expression.
- **<define-expr>:** The define expression is popped off the stack. If the define expression introduces a new function, then the symbol *define*, the list of parameters, and the body of the function are pushed onto the stack. The list of parameters will be processed by the lambda lifter in the same way as an application expression. The symbol *define* will be lambda lifted as a symbol which is described above. If the define expression introduces a variable (i.e. not a function), then the symbol *define*, the variable, and the expression that binds the variable are pushed onto the stack.
- **<env-list>:** If the top element on the stack is an *<env-list>*, then the result of lambda lifting the body of a lambda or define expression is stored in *RES*. Each symbol in *RES* is checked for membership in the union of *KNOWN* and *ENV*. If all symbols are members of this set, then a combinator is being lifted. Therefore, a new uniquely named function is added to *NEWFS* whose parameters are stored in *ENV* and the body is stored in *RES*. *RES* is set to the name of the newly created function. If *RES* contains free variables, a new uniquely named function is added to *NEWFS* whose parameters are the free variables in *RES*, whose body is a lambda expression whose list of parameters stored in *ENV* and whose body stored in *RES*. *RES* is set to a list representing a call to the newly created function with the free variables as arguments. In both cases, *D* is incremented by 1 and *ENV* is the set to the list of identifiers popped off the stack.

- `<result-list>`: The result-list is popped off the stack. It stores the result, *r*, of previously lambda lifted subexpressions. The lambda lifted expression stored in *RES* is concatenated to the front of *r*. This value is then stored in *RES*.
- `<empty-list>`: The empty-list is popped of the stack and a list is made of *RES*. *RES* is a function application without arguments. A list is made of *RES* to preserve the list structure of the expression being lifted.
- `<app-expr>`: If the length of the top of the stack is 1, the application has 0 parameters. Otherwise, it has at least one parameter. When a function application does not have parameters, *RES*, the empty list, and the function being applied are pushed onto the stack. When the function application has parameters, *RES* and the elements of the application expression are pushed in reversed order.

6.2 Examples

Consider the following definition:

```
(define (f L)
  (map (lambda (y) (inc y)) L))
```

Assuming *map* is a primitive, the only free variable in the body of the lambda expression may be *inc*. If *inc* is lambda lifted before *f*, it is not considered free. On the other hand, if *inc* has not been processed by the lambda lifter when *f* is processed then *inc* is identified as free. If *inc* is free, the lambda lifter yields:

```
(define (UniqueName2 inc) (lambda (y) (inc y)))
(define (f L)
  (map (UniqueName2 inc) L))
```

The result is semantically correct. However, Algorithm I unnecessarily creates new functions. This occurs, because variable or function definitions that have not been lambda lifted are considered free. This is undesirable, because the a partial evaluator and dynamic code generator will perform work for user-defined functions and variables. In addition, when the list *KNOWN* becomes large searching it is inefficient and makes the lambda lifter slow. In the next algorithm presented, globally defined variables are not treated as free. Therefore, fewer new functions are created and the list *KNOWN* does not grow as fast.

7. ALGORITHM II

7.1 Design Algorithm

Algorithm II differs from Algorithm I in the way lambda and define expressions are processed and in what is stored in the list *KNOWN*. We distinguish between globally defined values and free variables in the lexical scope of subexpressions. A global value does not appear in any of the variable declarations that are lexically valid when lifting an expression. A free variable, on the other hand, does appear in a lexically valid declaration. For example, consider the following definitions:

```
(define (f x) (inc x))
(define (g x) (lambda (y) (+ x y))).
```

In the body of *f*, *inc* is assumed to be defined globally and, therefore, is not treated as a free variable by Algorithm II. In the body of the lambda expression in *g*, *x* is considered a free variable, because it is declared in the definition of *g*.

The lambda lifter needs to keep track of all the parameters declarations that are valid when an expression is lifted. To accomplish this, each time a definition or lambda expression is encountered *ENV* is added to the front of *KNOWN*. For instance, lambda lifting the body of *g* causes the variable *x* to be added to the list *KNOWN*. To lambda lift an expression we check each symbol appearing in the expression for membership in *ENV*. If a symbol appears in *ENV*, it is not a free variable. If it does not appear in *ENV*, we check for membership in *KNOWN*. If it appears in *KNOWN* it is considered free. Otherwise, it must be globally defined and the lifter does not treat it as a free variable.

The actions taken by the lambda lifter at each iteration are the same as those of Algorithm I except for the following cases:

- `<lambda-expr>`: The lambda expression is popped off the STACK. *ENV* (which contains all declared variables that are lexically valid for the lambda expression), *RES*, and the body of the lambda expression are pushed on the STACK. *ENV* is added to the front of *KNOWN* and becomes the declarations of the current lambda expression.
- `<define-expr>`: The define expression is popped off the STACK. If the define expression introduces a new function, the same actions as Algorithm I are taken. In addition, if the definition introduces a new function, its parameters are added to the *KNOWN* list.
- `<env-list>`: The env-list is popped off the stack. The symbols in *RES* are checked for membership in *ENV*. If all symbols are in *ENV*, a combinator is being lifted and a new function is created as in Algorithm I. Any symbols that are not in *ENV* are checked for membership in *KNOWN*. If any appear in *KNOWN*, the expression contains free variables and a new function is created whose parameters are only those variables that appear in *KNOWN* and whose body, as in Algorithm I, is a lambda expression. Any symbols not appearing in *KNOWN* must be globally defined and there is no need to consider them free. If there are no symbols that are members of *KNOWN*, the function is a combinator and a new function is created as described before. After the lifting is complete, the members of the parameter list are removed from the front of *KNOWN*. The values of *D*, *RES*, and *ENV* are updated as in Algorithm I.

7.2 Example

Consider the same example used to illustrate Algorithm I:

```
(define (f L)
  (map (lambda (y) (inc y)) L)).
```

During the lifting of the lambda expression, *inc* is not considered a free variable, because it is not declared in the lexical scope of the expression. Thus, it is considered a global value and there is no need to pass it in to a new functions as a free variable has to be passed in. Algorithm II considers the lambda expression a combinator a creates a new known function that does not have a lambda expression in its body. The resulting definitions are:

```
(define (f L) (map UniqueName3 L))
(define (UniqueName3 y) (inc y))
```

The result of this transformation for this example yields a form in which there is no need to call the partial evaluator or code generator at runtime. Algorithm II reduces the number of free variables and, therefore, the number of new functions generated by the lambda lifter at compile time and by the partial evaluator at runtime.

8. CONCLUDING REMARKS AND FUTURE WORK

In this article, we have presented two algorithms to perform lambda lifting on Scheme programs. The lambda lifting algorithms were changed to produce resulting programs that conform to the semantics of Scheme. Instead of producing results that completely eliminate free variables for a language that supports curried functions, our lambda lifting algorithm restricts the existence of free variables to single non-nested lambda expressions. Functions that contain a lambda expression as their body have the free variables of the lambda expression as their parameters. In this form, a partial evaluator can be used to dynamically generate code for functions specialized for the bindings of their free variables. The techniques are similar to currying functions in ML, but input is not consumed lazily. Instead, input is staged in two steps. In the first, the arguments for the free variables are received. In the second, the arguments for the bound variables are received.

The two lambda lifting algorithms presented differ in the definition they use for free variables. In Algorithm I, a free variable is any symbol that is not known at the time an expression is lifted. Despite producing semantically correct code, this algorithm must ultimately be rejected. It unnecessarily creates new functions for globally defined functions that are treated as free variables. Algorithm II avoids the unnecessary creation of new functions by not treating global definitions as free variables. In Algorithm II, only symbols that are declared in the lexical environment of a function, but not in its parameter list are considered free variables.

Our chosen lambda lifting algorithm has been implemented as a tail-recursive function in Scheme. This was done with the purpose of making its translation into a while loop in C++, the implementation language of the MT evaluator virtual machine[8, 9], as easy as possible. Our immediate plans are to port Algorithm II for the MT system and then implement a partial evaluator for a pure subset of Scheme.

9. ACKNOWLEDGMENTS

The authors would like to thank our fellow MTers (Victor Encarnacion, Patrick DeSomma, and Robert Moore) for

their assistance in the development of this project. We also thank the Department of Math and Computer Science and the University Research Council of Seton Hall University for the support they have provided.

10. REFERENCES

- [1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Proceedings of The Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13. Springer-Verlag, Aug 1991.
- [2] Luca Cardelli. Compiling a Functional Language. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 208–217, New York, 1984. ACM Press.
- [3] Chris Hanson. Efficient Stack Allocation for Tail-Recursive Languages. In *ACM Conference on Lisp and Functional Programming*, pages 106–118, New York, June 1990. ACM Press.
- [4] T. Johnsson. Lambda lifting: Transforming Programs to Recursive Equations. *Proceedings of the IFIP Conference on Functional Programming and Computer Architecture*, ed. Jouannaud, LNCS 201, 1995.
- [5] L. Lafave and J.P.Gallagher. Partial Evaluation of Functional Logic Program in Rewriting-based Languages. *ACM Transactions Programming Languages and Systems*, 20:768–844, 1998.
- [6] Dmitry Lomov. Dynamic Caml: a Case Study into Implementation of Dynamic Code Generation Libraries. In Greg Michaelson and Phil Trinder, editors, *Draft Proceedings of the 15th International Workshop on Implementation of Functional Languages*, pages 193–203, Edinburgh, Scotland, 2003.
- [7] Luca Cardelli. The Functional Abstract Machine. Technical Report No.107, Bell Laboratories, April 1983.
- [8] Marco T. Morazaán. Towards DVM Friendly First-Class Functions. In Stephen Gilmore, editor, *Draft Proceedings of the Fourth Symposium on Trends in Functional Programming*, Edinburgh, Scotland, 2003. University of Edinburgh.
- [9] Marco T. Morazaán. The MT Evaluator Virtual Machine. In *The 2004 Proceedings of the Hawaii International Conference on Computer Sciences*, pages 480–497, Honolulu, Hawaii, 2004.
- [10] Mark Leone and Peter Lee. Lightweight Run-Time Code Generation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994.
- [11] Peter Lee and Mark Leone. Optimizing ML with Run-Time Code Generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–148. ACM Press, May 1996.

- [12] Zhong Shao and Andrew W. Appel. Efficient and Aafe-for-Space Closure conversion. *ACM Transactions Programming Languages and Systems*, 22:129–161, January, 2000.