# Bytecode Closures

Marco T. Morazán

Seton Hall University, South Orange, NJ, USA
`morazanm@shu.edu`

**Abstract.** This article describes a new project to study the memory performance of three new implementation strategies for closures coined bytecode closures. The project proposes to compare the new implementation strategies to the classical strategy that dynamically allocates flat closures as heap data structures. The new closure representations are based on dynamically creating specialized bytecode instead of allocating a data structure. The first new strategy creates specialized functions by inlining the bindings of free variables. The second uses memoization to reduce the number of dynamically created functions. The third dynamically creates memoized specialized functions that treat free variables like parameters at runtime. Empirical results from a preliminary case-study using three small benchmarks are presented as a proof-of-concept. The data suggests that dynamically created bytecode closures in conjunction with memoization can allocate significantly less memory, as much as three orders of magnitude less memory in the presented benchmarks, than a flat closure implementation. Furthermore, the empirical data suggests that the size of a dynamically generated function ought to be proportional to the size of a flat closure. The article ends with other, longer term, interesting lines of research that are also pursued by this new project.

## 1 Introduction

In functional languages functions are first-class. This means that functions can be passed as arguments to functions and can be returned as the result of evaluating a function. One of the consequences of first-class functions that programming language implementors must resolve is how to represent functions that may be applied outside of their lexical scope. Care must be taken to represent functions, because they may contain references to *free* variables[1]. For example, consider the function in Figure 1. The function mk-mapper declares the variable f which is free in the returned function. Notice that the returned function can only be applied outside the lexical scope of f. Therefore, f must be "remembered" by the returned function.

In the $\lambda$-calculus [2], $\beta$-reduction is used as the mechanism for remembering the bindings of free variables. The $\beta$-rule

$$(\lambda x.e)x_0 \rightarrow e\{x_0/x\}$$

---

[1] A variable, $x$, is free in a function, $f$, if $f$ references $x$, $f$ does not declare $x$, and $x$ is declared by an ancestor of $f$ in the program's parse tree.

```
(define (mk-mapper f)
    (define (mapper L)
        (cond [(null? L) L]
              [else (cons (f (car L)) (mapper (rest L)))]))
    mapper)
```

**Fig. 1.** A function that returns a function.

states that all free occurrences of $x$ in $e$ are replaced by $x_0$. Typical implementations of functional languages, however, do not perform actual substitutions in $e$ and, instead, use an *environment* to track what should have been substituted [3]. Thus, to represent a function, with references to free variables, that may be applied outside its lexical scope, the creation of a closed package, called a *closure* [20], is required. The closure captures the bindings of the free variables by storing (a pointer to) an environment. For example, in Figure 1 a closure is created to retain the binding of f for the returned function mapper.

Closures, in this context, are functions that are represented using a data structure in order to avoid actual substitutions. Part of the data structure represents the function itself (i.e., the code to be evaluated) and part of the data structure represents the environment that gives meaning to the function. This representation facilitates the compilation of functions given that the structure of the function remains constant at runtime (i.e., the bindings of the free variables do not change the compiled function). In contrast, substitution would change the structure of a compiled function every time the bindings of the free variables changed requiring the creation of a new function specialized for the bindings of its free variables.

An alternative to using a data-structure closure to represent a function, of course, is to create a specialized function by performing actual substitutions. Such an approach has been investigated in the past, but implementations to date have led to excessive memory allocation [7, 11]. The project described in this article aims to study three strategies for implementing dynamically created byetcode closures instead of dynamically allocating data-structure closures. The memory-efficient strategies are expected to come from a controlled form of actual substitutions employing memoization [18]. The description introduces the implementation strategies using a small pure functional language and compares the strategies using small benchmarks–as a preliminary proof-of-concept. The presented empirical data suggests that memory-wise memoized dynamically allocated bytecode closures can be a viable alternative to data-structure closures. The article ends with other, longer term, interesting lines of research that are also pursued by this new project.

## 2   Closures: Representation and Issues

Typically, closures are heap-allocated data structures that are created every time a function with free variables needs to be represented. Historically, closures

have been implemented in a number of ways. Early implementations of functional languages, like Henderson's Lisp [12] using a SECD machine [20], used *linked* closures (a.k.a deep closures). In a linked closure, a list of frames (i.e., the existing environment) is used to store the bindings of the free variables. The attractive feature of this approach is that closure creation is done in constant time. Accessing the binding of a free variable, however, requires an $O(n)$ traversal of the list of frames, where $n$ is the lexical offset of the free-variable reference. The space required to store a closure is proportional to the size of the environment–amortized over all the closures that share the environment. This closure representation makes closure creation fast at the expense of making resolving variable references slower [23]. In addition, bindings that are no longer relevant to a computation are unnecessarily kept alive (i.e., not garbage collected) by storing (a pointer to) the entire existing environment as part of the closure.

An alternative to linked closures, used by the Functional Abstract Machine (FAM) [4, 14], are flat closures (a.k.a. display closures [23]). A flat closure employs an array to store the bindings of free variables. Free variables are accessed by a fixed displacement within the array in constant time. Closure creation requires copying the bindings of free variables into the closure. Therefore, flat-closure creation is $O(v_f)$, where $v_f$ is the number of free variables the function depends on. The space required to store a closure is proportional to the number of free variables a function depends on which is always less than or equal to the size of the environment. This closure representation makes the resolution of references to free variables faster at the expense of closure creation time. In addition, this representation only stores the part of the environment that is relevant to the remaining computation and, thus, allows a garbage collector to be more effective by allowing the recycling of memory space used by bindings that are known to no longer be relevant to the computation.

Shao and Appel observed that flat-closure creation may require many values to be copied repeatedly from closure to closure [25]. To avoid this copying, they developed *safely linked* closures that allow for bindings to be shared between closures. Free variables referenced by more than one function are grouped together into a shareable record. Their representation strategy guarantees that the nesting of safely linked closures never exceeds two. The space required to store a closure is proportional to the number of free variables a function depends on, but when multiple functions have free variables in common the space required is reduced by $(f-1) * n$, where $f$ is the number of functions that share free variables and $n$ is the number of free variables the functions share. This closure representation makes closure creation faster than flat-closure creation at the expense of adding overhead to the resolution of references to free variables. In addition, bindings are only kept alive while they may still be relevant to the computation allowing the space they occupy to be recycled as soon as possible by a garbage collector.

Always allocating a closure data structure to represent a function with free variables can lead to excessive memory allocation. This is why many modern
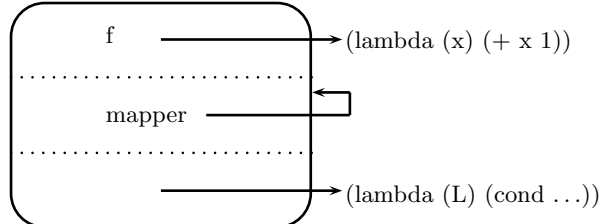
**Fig. 2.** Conceptual View of the Flat Closure for (mk-mapper (lambda (x) (+ x 1))).

implementations of functional languages attempt to eliminate closure allocations whenever possible. For instance, MzScheme [9], which uses flat closures [8], inlines functions and adds free variables as arguments to functions whenever all applications of a function are visible [1]. In addition to reducing memory allocation, providing fast access to free variables is another goal of modern implementations. This has led to a variety of methods to access the bindings of free variables. In MzScheme, for example, the bindings of free variables are not accessed directly from the closure. Instead, the bindings are unpacked onto the stack whenever the closure is applied [8]. Some implementations of functional languages make free variables explicit by performing program transformations such as lambda lifting [13, 16] and closure conversion [15]. Lambda lifting explicitly adds free variables as parameters to functions. Accesses to free variables in the source program are turned into parameter accesses. Closure conversion explicitly adds an environment parameter to functions. The bindings of free variables in the source program are accessed through the environment parameter.

## 3  Intuitive Data-Structure Closure Elimination

When closures are implemented as data structures, heap memory is allocated every time a function with free variables needs to be represented. For example, consider the code Figure 1 and the evaluation of:

```
(mk-mapper (lambda (x) (+ x 1))).
```

This expression returns the closure displayed in Figure 2. This conceptual view of the flat closure has f bound to the representation of the combinator that adds 1 to its input. In addition, it has mapper bound to the closure itself, thus, enabling the self-reference (i.e., recursive application) in the body of the function the closure represents.

Instead of allocating and returning a data structure closure, a specialized version of the returned function, based on the binding of f, can be dynamically created. Specifically, if substitutions were performed the function returned would be semantically equivalent to this new function:

```
(define (mapper-f-x-x+1 L)
  (cond [(null? L) L]
        [else (cons ((lambda (x) (+ x 1)) (car L))
                    (mapper-f-x-x+1 (rest L)))])).
```

In this example, the returned function, mapper-f-x-x+1, has the same structure as, mapper, the function specialized. In general, however, the returned specialized

function does not require the same structure[2]. The important point is that the returned function is a combinator. That is, it lacks references to free variables and, as such, does not require a data-structure closure to store the bindings of free variables.

Studying variations of three basic strategies to dynamically create such a combinator as a bytecode function, coined a bytecode closure, is the focus of the project. The first strategy inlines the bindings of the free variables into the returned function as suggested by $\beta$-reduction. The advantage of this implementation strategy is that the resolution of free variables is transformed to accessing constants in specialized functions. A potential disadvantage is that inlining may lead to code explosion and require more memory allocation than closures when the functions being specialized are large relative to the number of free variables referenced. The second attempts to reduce memory consumption by memoizing inlined functions. That is, dynamically-created specialized functions are reused. The third strategy breaks away from inlining functions in the source code. Instead, references to free variables in the source code are transformed to parameter references. Specialized bytecode closures inlined with the bindings of free variables that push these bindings onto the stack are memoized. When compared to using flat closures or inlined source functions, the advantages of this implementation strategy are that the resolution of free variables is made faster than using flat closures by treating free variables as parameters, that the memory dynamically allocated for specialized functions is proportional to the number of free variables (not the size of the specialized function) as it is for flat closures, and that the structure of compiled $\lambda$-terms does not change. A potential disadvantage, unlike inlining, is that a jump is required to transfer control from the function that pushes the bindings of the free variables to the function that utilizes these bindings. Finally, comparing the performance on bytecode closures with memoized flat closures is part of the project. Memoized flat closures eliminate the need for the jump required by the third strategy at the expense of increasing the access time to free variables. Empirical data will be collected to determine when, if ever, one implementation strategy is superior to the others.

## 4    Illustrating the Compilation Process

The BNF grammar for a small core language is displayed in Figure 3. This core language is used for the preliminary results presented in this article. A program consists of zero or more definitions. A definition consists of a header which contains the function name and the parameters, zero or more local definitions, and a body which is an expression. An expression is a number, a symbol, a boolean, an *if* expression, an *application* expression, or a *lambda* expression.

The architecture of the proof-of-concept compiler is displayed in Figure 4. A source program is first parsed. The parse tree is given as input to a $\delta$-*reducer*. The

---

[2] This can be the result, for example, of performing $\delta$-reductions.

$$program \rightarrow def^*$$
$$def \rightarrow (\textbf{define}\ (symbol^+)\ def^*\ expr)$$
$$expr \rightarrow number$$
$$\rightarrow symbol$$
$$\rightarrow boolean$$
$$\rightarrow (\textbf{if}\ expr\ expr\ expr)$$
$$\rightarrow (expr^+)$$
$$\rightarrow (\textbf{lambda}\ (symbol^*)\ expr)$$

**Fig. 3.** The BNF Grammar of the Source Core Language

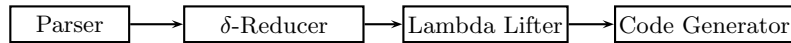| Parser | → | $\delta$-Reducer | → | Lambda Lifter | → | Code Generator |

**Fig. 4.** The General Architecture of the Proof-of-Concept Compiler.

$\delta$-*reducer* replaces a primitive function applied to its required known arguments by a result. This transformation reduces the size of the resulting program by evaluating primitive application expressions and by eliminating dead code (e.g., when the condition of an *if*-expression can be evaluated at compile time). The $\delta$-reduced parse tree is given as input to a lambda lifting function (e.g., [16]). Lambda lifting makes the free variables of a function explicit and, thus, the variables by which to specialize functions at runtime. The parameters of a lifted function are the original function's free variables in the source program and the parameters of the nested lambda expression are the parameters of the original source function. The resulting lambda lifted parse tree is passed to the code generator to produce bytecode.

To illustrate the process, consider a function common in environment-passing interpreters to evaluate the arguments of an application expression. The function takes as input a list of expressions to be evaluated and the environment (implemented as a list of frames) in which to evaluate the expressions. It returns a list containing the results of evaluating each expression. Using the syntax of Figure 3, the function can be implemented as follows:

```
(define (eval-operands rands env)
  (map (lambda (e) (eval-expr e env)) rands))
```

After parsing, the $\delta$-reducer discovers that there are no primitive application expressions that can be evaluated and produces as output the original parse tree. Lambda lifting hoists the lambda expression to the global level. Since env is the only free variable in this function, env is the only parameter in the lifted function. The body of the lifted function is itself the original lambda expression. In the body of eval-operands, the lambda-expression is substituted with an application

```
eval-rands              lifted1                 FN19
   FCALL                   ACLOSURE FN19 1          FCALL
   FCALL                   COPY2CLOSURE 1 1         PACC 1
   PACC 2                  FRETURN 1               FVACC 1
   <update registers>                             <update registers>
   GOTO lifted1                                   GOTO eval-expr
   PACC 1
   <update registers>
   GOTO map
```

**Fig. 5.** Compiled Code for the MT Virtual Machine

expression that applies the lifted function to its free variable. After lambda
lifting, the parse tree represents the following program:

```
(define (eval-rands rands env) (map (lifted1 env) rands))
(define (lifted1 env) (lambda (e) (eval-expr e env))).
```

The lambda lifted parse tree is passed to the code generator to produce the
bytecode displayed in Figure 5. The displayed code is generated assuming the use
of flat closures and, to aid readability, Figure 5 omits the proper handling of tail
calls (i.e., control context is accumulated on a stack). Bytecode is generated for
3 functions: eval-rands, lifted1, and the nested lambda expression in lifted1 (i.e.,
FN19 in the bytecode). The compiled code for eval-rands sets up an activation
records on the stack for the call to map and another for the call to lifted1. For the
latter it pushes, env, the second parameter (i.e., PACC 2) onto the stack, updates
control-flow registers, and calls the function. After returning from lifted1, the
bytecode pushes rands, the first parameter, onto the stack (i.e., PACC 1), updates
control-flow registers, and calls map. The bytecode generated for lifted1 allocates
a closure of size 1 for FN19 (i.e., the nested lambda expression), populates the
closure with the binding of the first parameter, and returns this closure after
popping off its activation record with 1 parameter (i.e. FRETURN 1). The code
for FN19 sets up an activation record for the call to eval-expr, pushes e, its
parameter, and the free variable env onto the stack (i.e., FVACC 1), updates
control-flow registers, and makes the call to eval-expr.

## 5 Bytecode Closures Implementation Strategies

This section describes the three strategies to dynamically create bytecode clo-
sures. It is important to remember that the code generator expects lambda lifted
programs in which a lambda expression only exists as the body of a global func-
tion and in which lambda expressions contain at least one reference to each of
the parameters of its enclosing global function. It is these anonymous functions
that are specialized at runtime.

```
(define (eval-expr expr env)
   (if (literal? expr)
       expr
       ...
       (if (app-expr? expr)
           (apply-proc (eval-expr (proc-expr expr) env)
                       (cons (eval-operands (ops-expr expr) env) env))
       ...)))
(define (eval-operands rands env)
   (map (lambda (e) (eval-expr e env)) rands))
```

**Fig. 6.** Program Fragment of an Environment-Passing Interpreter.

### 5.1   Strategy I: Inlined Functions

In strategy I, anonymous functions (i.e., compiled $\lambda$-terms) are treated as templates with holes. These templates are never executed at runtime and are only used to generate specialized versions of the anonymous function. The holes are the instructions to access free variables (i.e., FVACC instructions in the bytecode).

To generate a specialized function from a template, the bytecode of the template is copied. The holes of the template, however, are filled with instructions to push a constant onto the stack based on the binding of the free variable referenced. That is, specialized functions are inlined with the bindings of the free variables wherever free variables are referenced. Care must be taken to handle jumps to labels correctly (e.g., in the compiled code of an if-expression). Branch instructions that refer to labels can not simply be copied, because that would mean branching into the template instead of a location in the specialized function. The fact that the template and the specialized function have the same number of instructions means that simple address arithmetic solves the problem at runtime.

This strategy uses the *blind* policy of always generating a specialized function whenever a flat closure would be generated. Dynamic function creation using this implementation strategy is $O(n)$, where $n$ is the size of the function being specialized. That is, the time it takes to create a specialized function is proportional to the number of bytecode instructions in the function and not the number of free variables the function references. In general when the size of specialized functions is large relative to the number of free variables referenced, it is expected for such an implementation to be inefficient when compared to using flat closures for two reasons. The first is that programs allocate more memory. The second is that specialized function creation takes longer than closure creation.

To illustrate this strategy, consider the program fragment in Figure 6 for an environment-passing interpreter and the evaluation of

(eval-expr '(h (g (f x)) (g (f y))) '(((x 2) (y 2)))),

```
(define (lifted1-1 e) (eval-expr e '(((x 2) (y 2)))))
(define (lifted1-2 e) (eval-expr e '(((x 2) (y 2)))))
(define (lifted1-3 e) (eval-expr e '((b (f 2)) ((x 2) (y 2)))))
(define (lifted1-4 e) (eval-expr e '((b (f 2)) ((x 2) (y 2)))))
(define (lifted1-5 e)
  (eval-expr e '(((i (g (f 2))) (j (g (f 2)))) ((x 2) (y 2)))))
```

**Fig. 7.** Five Dynamically Created Inlined Functions.

where f, g, and h are user-defined functions, and the environment binds x and y to 2. The function eval-operands[3] is called 5 times: once for h, twice for g, and twice for f. The evaluation of both applications of f is done with the same environment (i.e., the displayed environment). Likewise, the evaluation of both applications of g is done with the same environment (i.e., value-wise). The result at runtime is the generation of the 5 functions[4] displayed in Figure 7. Notice that the generated functions for f, lifted-1 and lifted-2, and the generated functions for g, lifted-3 and lifted-4, are, respectively, semantically equivalent. This means that 3 specialized functions can be generated, instead of 5, to evaluate the expression. Generated functions that are semantically equivalent to needed functions can be re-used to reduce memory allocation.

### 5.2  Strategy II: Memoized Inlined Functions

In strategy II, anonymous functions are also treated as templates with holes. As in strategy I, the instructions of the template are copied and inlined with the binding of the referenced free variables.

The second implementation strategy does not blindly create specialized functions. Instead of always generating a function when a closure would be allocated, specialized functions are memoized and functions are only dynamically created when needed. Specialized function memoization requires a cache of specialized functions to be maintained. If a specialized function is needed and is found in this cache, then the previously generated specialized function is re-used. Otherwise, a new specialized function is generated and this new function is added to the cache of specialized functions. Determining function equality is achieved by exploiting the naming convention used for specialized functions. Instead of simply generating a fresh identifier, the fresh identifier is a linear combination of the name of the function being specialized and of the types and the bindings of the free variables.

To illustrate how memoized function specialization works, once again, consider the program fragment in Figure 6 for an environment-passing interpreter and the evaluation of:

(eval-expr '(h (g (f x)) (g (f y))) '(((x 2) (y 2)))).

---

[3] This function is lambda lifted as described in Section 4.

[4] In the interest of readability, source syntax is used in this example.

As before, the function eval-operands is called 5 times, but only 3 specialized functions are generated[5]:

```
(define (lifted1-list-100-2400 e) (eval-expr e '(((x 2) (y 2)))))
(define (lifted1-list-3000-5000 e)
  (eval-expr e '(((b (f 2))) ((x 2) (y 2)))))
(define (lifted1-list-7500-8150 e)
  (eval-expr e '(((i ((g (f 2)))) (j ((g (f 2)))))
                ((x 2) (y 2))))).
```

Notice that in this example we have a 40% reduction in the number of dynamically generated inlined functions when compared to using strategy I.

### 5.3   Strategy III: Memoized Auxiliary Inlined Functions

In strategy III, $\lambda$-terms are not treated as templates with holes. Instead, these anonymous functions are executable and are converted to combinators. Free-variable references become parameter references. In this context, a specialized byetcode function has two roles. The first is to push the bindings of free variables needed by an anonymous function onto the stack. The second is to transfer control to the anonymous function for which it was created. A specialized function, in other words, completes the construction of the activation record for an anonymous function.

The third implementation strategy makes dynamic function creation memory efficient by making the size of specialized functions proportional to the size of the flat closures they substitute. A specialized function is a collection of instructions to push constants onto the stack followed by a jump instruction. At compile time, the order in which constants are to be pushed onto the stack by a specialized function is determined. This order corresponds to the order of the parameters of a lambda-lifted function that has an anonymous function in its body. Every parameter of such a function must be referenced by the $\lambda$-expression in its body. Therefore, to create a specialized function, the runtime system only needs to examine the current activation record to assemble the instructions to push constants onto the stack in the right order and to add a jump to the anonymous function.

To illustrate how function specialization works using strategy III, once again, consider the program fragment in Figure 6 and the evaluation of

(eval-expr '(h (g (f x)) (g (f y))) '(((x 2) (y 2)))).

The compiled code for lifted1 and its nested anonymous function (i.e., FN19) are displayed in Figure 8. Observe that the compiled code for FN19 is almost the same the compiled code in Figure 5. The only difference is that the reference to the first free variable (i.e., FVACC 1) is now a reference to the second parameter (i.e., PACC 2). The compiled code for lifted1 generates a specialized function

---

[5] The function names are based on the linear combination convention mentioned above.

```
lifted1                            lifted1-list-100-2400
  GENF FN19 1                            PUSHLIST 100 2400
  FRETURN 1                              GOTO FN19
FN19                               lifted1-list-3000-5000
  FCALL                                  PUSHLIST 3000 5000
  PACC 1                                 GOTO FN19
  PACC 2                           lifted1-list-7500-8150
  <update registers>                     PUSHLIST 7500 8150
  GOTO eval-expr                         GOTO FN19
```

**Fig. 8.** Strategy III Lambda Expression.   **Fig. 9.** Strategy III Specialized Functions.

for FN19 that adds one parameter to FN19's activation record (i.e., GENF FN19 1). As for strategy II, the function eval-operands is called 5 times and only 3 specialized functions are generated which are displayed using bytecode in Figure 9. One function is generated for each of the different bindings for env. Each generated function pushes the binding of env (which is a list) onto the stack and transfers control to FN19. It is straightforward to see that the specialized versions of lifted1 are smaller than their counterparts using strategies I or II.

## 6 Preliminary Empirical Results

This section presents preliminary memory allocation empirical results obtained from three small benchmarks. These results are intended solely as an indication that there is fertile ground for exploration using larger benchmarks. First, the benchmarks are briefly described. Second, the performance measurements are presented. The benchmarks naively use higher-order functions to test extreme ends of the memory allocation spectrum. The lambda lifted versions of the benchmarks are found in the appendix in section 10.

### 6.1 Benchmarks

**AP** This benchmark traverses a list of pairs of integers to produce a list that contains the sums of each pair. The presented measurements are for a list of 9,999 pairs with each pair containing two randomly generated integers in [0..9999].

**TK** This is the triply recursive integer function related to the Takeuchi function, one of Gabriel's benchmarks [10], that has been modified to maximize the use of anonymous functions. The presented measurements are for (tak 18 12 6).

**ST** This benchmark traverses a binary tree of integers and scales each integer in the tree by its depth in the tree. The presented measurements are for the scaling of a full binary tree of depth 15.

Each benchmark was executed 4 times for a total of 12 experiments on a non-distributed version of the MT virtual machine [17]. The benchmarks were

|  | Relative Difference |
|---|---|
| AP Strategy I | -0.6190 |
| AP Strategy II | 0.3325 |
| AP Strategy III | 0.3332 |
| TK Strategy I | -0.8889 |
| TK Strategy II | 79.13 |
| TK Strategy III | 794.3 |
| ST Strategy I | -0.1998 |
| ST Strategy II | 0.2497 |
| ST Strategy III | 0.2900 |

**Fig. 10.** Relative Difference with Flat Closures.

executed using flat closures and each of the three strategies described in the previous section, denoted by *Strategy I*, *Strategy II*, and *Strategy III*.

## 6.2 Measurements and Analysis

For each benchmark, Figure 10 displays the relative difference in memory allocation between flat closures and each of the strategies[6]. A negative relative difference means that the flat-closure-based implementation allocated less memory.

For each benchmark, strategy I incurs the maximum number of allocations. The total excess memory allocation ranges from about 20% to about 90% when compared to the flat-closure-based implementation. This occurs, as expected, because the number of dynamically created functions is the same as the number of closures allocated and the size of a specialized function is larger than the size of a flat closure. These numbers clearly suggest that such a naive implementation of dynamically allocated functions is neither efficient nor feasible for industrial-strength implementations.

Strategies II and III significantly outperform the flat-closure-based implementation (as well as Strategy I). For the AP benchmark, the flat closure based implementation allocates about 33% more memory than either of these strategies. The savings in memory allocation are due to memoization exploiting the modest amount of repetition in the generation of random numbers in [0..9999]. Strategies II and III virtually exhibit the same performance with strategy III displaying slightly less memory allocation. The observed performance is so close, because the function being specialized is small and a specialized inlined function generated with strategy II is only one bytecode instruction larger than a specialized function generated with strategy III. This benchmark clearly suggests that memoization of dynamic functions can significantly reduce memory allocation when compared to flat closures.

---

[6] Relative difference is defined as $\frac{Flat\ Closure\ Allocations - Bytecode\ Strategy\ Allocations}{Bytecode\ Strategy\ Allocations}$.

For the TK benchmark, we observe the largest gain in performance over the flat-closure-based implementation. For strategy II the flat-closure-based implementation allocates about 7,913% more memory (i.e., two orders of magnitude more memory) while for strategy III the excess allocation by the flat-closure-based implementation reaches 79,430% (i.e., three orders of magnitude more memory). The difference is quite significant and occurs because the Takeuchi triply recursive function makes many recursive calls with the same arguments. This benchmark presents the ideal conditions under which memoization is most effective. Strategy III significantly outperforms strategy II by about 1 order of magnitude. This difference occurs, because the inlined specialized functions generated using strategy II are significantly larger than the specialized functions generated by strategy III. The TK benchmark clearly suggests that memoized dynamically generated functions can lead to significantly less memory allocation than flat closures.

For the ST benchmark, the flat closure base implementation allocates about 25% more memory than strategy II and 29% more memory than strategy III. For this benchmark, memory allocation is dominated by allocation to build a list-based structure (i.e., a full binary tree). For strategies II and III, only a small number of functions, 16, are dynamically created. In essence, only one specialized function is created per tree level due to memoization. The savings observed are attributed to the large number of flat closures allocated by the the classical implementation (one for each node in the full binary tree). The difference between strategy II and strategy III is due to the smaller functions generated by the latter. This benchmark clearly suggests that even for programs in which first-class functions only play a small role, memoized bytecode closures can significantly reduce memory allocation.

The preliminary empirical data clearly suggests that the thesis that memoized bytecode closures can exhibit significantly better memory performance than flat closures and deserve further study. Furthermore, the data also suggests that keeping the size of dynamically created bytecode closures proportional to the number of free variables is important.

## 7  Related Work

Feeley and Lapalme first suggested generating code instead of allocating a data structure closures [7]. In their work, a specialized function only pushes the bindings of free variables onto the stack and these bindings are accessed like parameters by a compiled lambda expression akin to strategy III described in this article. Their performance measurements indicate that for their implementation strategy memory allocated for specialized functions increases by up to 25% when compared to a closure-based implementation. The major differences between with the approach described in this article and their work is the use of memoization and lambda lifting. Memoization can significantly reduce memory allocation as argued in the previous section. Lambda lifting reduces the complexity of compiling for specialization, but at runtime exactly the same specialized functions

are created by both approaches. Finally, Feeley and Lapalme also address the problems introduced by assignment and propose pushing the address to a mutable box instead of a binding to extend the technique to support assignment. A similar approach would work with the memoization-based strategies described in this article.

More recently, Grabmüller developed a prototype system to implement closures using runtime code generation for a strict (and pure) functional language [11]. Instead of compiled code, this approach uses abstract syntax trees at runtime to create specialized functions. The runtime code generator inlines functions with the bindings of their free variables akin to strategy I described in this article. The use of abstract syntax trees is intended to simplify common optimizations (e.g., reduction to normal form and dead code elimination) that can be performed by the runtime code generator once the bindings of free variables are known. It is unclear, based on the preliminary work done on their prototype, if any runtime analysis of an abstract syntax tree can not be done a priori to indicate to the code generator what optimizations to perform. Grabmüller's performance evaluation indicates that the system runs out of memory space for some benchmarks, but provides no other indication on memory allocation performance.

There have been several approaches, far too many to reference here, to runtime code generation that have not focused on eliminating data-structure closures. Lee and Leone's FABIUS compiler specialize curried functions by inlining the bindings of arguments as they are received [21]. Consel and Noël have used runtime specialization for C programs that uses templates with holes to inline and partially evaluate functions [5]. Poletto et al. have also used dynamic code generation to improve the performance of a superset of C called 'C that requires programmers to annotate their code [22]. The work on 'C has been extended to a dialect of Java called DynJava to generate type safe specialized classes [19].

## 8 Concluding Remarks

This articles describes a new project to study the memory performance of representing closures as dynamically allocated bytecode functions. The preliminary empirical data presented suggests that memoized bytecode closures can significantly reduce memory allocation. The magnitude of the savings increases for programs in which first-class functions play a significant role at runtime reaching up to three orders of magnitude less allocation than flat closures in the presented benchmarks. The inescapable conclusion is that memoized bytecode closures is a technology worthy of future study. In addition, note that the memoization strategy described in this article does not break the high-level of abstraction provided by functional languages. That is, it does not require the programmer to be aware of the memoization process nor to annotate programs for function specialization to occur.

In addition to studying the memory performance of bytecode closures, this work will pursue several other interesting lines of research such as:

**Memoized Flat Closures** What impact do memoized flat closures have on performance? Clearly, the number of flat closures will be the same as for Strategy III bytecode closures. Their memory footprint and their allocation time will also be similar given that both are proportional to the number of free variables. The difference, if any, will be marked by free-variable access time.

**Inflation of Parameters** The bytecode closures presented in this article are based on a compiler that performs lambda lifting. Danvy and Schultz showed that lambda lifting may present efficiency difficulties due to parameter inflation which led them to propose lambda dropping[6]. A fundamental line of research is to determine if bytecode closures overcome the efficiency problems raised by the inflation of parameters.

**Continuations** It is common for functional programs to be transformed to continuation-passing style (CPS) [24, 26, 27]. A continuation can be represented as a function that knows how to complete the rest of the computation. Many implementations, however, transform continuations to a data structure representation. Another fundamental line of research is whether or not bytecode closures eliminate the need for this change.

**Garbage Collection** The performance of memoized bytecode closures hinges on their reuse. Performance, however, also hinges on the recycling of memory (that contains values no longer relevant to the computation) by a garbage collector. How should you garbage collect memoized closures, whether of the bytecode nature or the data structure nature? What rules or heuristics can be used to prevent premature recycling of memoized closures?

## 9 Acknowledgements

## References

1. *Guide: PLT Scheme*, 2008. http://docs.plt-scheme.org/guide/index.html.
2. H. P. Barendregt. *The Lambda Calculus : Its Syntax and Semantics, Revised Edition.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.
3. Malgorzata Biernacka and Olivier Danvy. A Concrete Framework for Environment Machines. *ACM Trans. Comput. Logic*, 9(1), December 2007.
4. 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.
5. Charles Consel and François NoëL. A General Approach for Run-time Specialization and its Application to c. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 145–156. ACM Press, 1996.
6. Olivier Danvy and Ulrik P. Schultz. Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure. *Theoretical Computer Science*, 248(1–2):243–287, 2000.

7. Marc Feeley and Guy Lapalme. Closure Generation Based on Viewing Lambda as Epsilon Plus Compile. *Journal of Computer Languages*, 17:17–4, 1992.
8. Matthew Flatt. Private Communication, May 2007.
9. Matthew Flatt. PLT MzScheme: Language Manual. Technical Report PLT-TR2008-1-v4.1, PLT Scheme Inc., 2008. http://www.plt-scheme.org/techreports/.
10. Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, Cambridge, MA, USA, 1985.
11. Martin Grabmüller. Implementing Closures Using Run-time Code Generation. Research report 2006-02 in *Forschungsberichte Fakultät IV – Elektrotechnik und Informatik*, Technische Universität Berlin, February 2006.
12. Peter Henderson. *Functional Programming: Application and Implementation.* Prentice-Hall International, Englewood, NJ, USA, 1980.
13. Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proc. of a Conf. on Functional Prog. Lang. and Comp. Arch.*, pages 190–203. Springer-Verlag New York, Inc., 1985.
14. Luca Cardelli. The Functional Abstract Machine. Technical Report No.107, Bell Laboratories, April 1983.
15. Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed Closure Conversion. In *Proc. of he 23rd ACM Symp. on Principles of Progr. Lang.*, pages 271–283. ACM Press, 1996.
16. Marco T. Morazán and Ulrik P. Schultz. Optimal Lambda Lifting in Quadratic Time. In Olaf Chitil, editor, *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 37–56. Springer Verlag, 2008.
17. Marco T. Morazán and Douglas R. Troeger. The MT Architecture and Allocation Algorithm. In Greg Michaelson, Phil Trinder, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, volume 1, pages 97–104, Bristol, UK, 2000. Intellect.
18. Peter Norvig. Techniques for Automatic Memoization with Applications to Context-Free Parsing. *Comput. Linguist.*, 17(1):91–98, 1991.
19. Yutaka Oiwa, Hidehiko Masuhara, and Akinori Yonezawa. DynJava: Type Safe Dynamic Code Generation in Java. In *Third JSSST Work. on Progr. and Progr. Lang.*, March 2001.
20. P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 1964.
21. Peter Lee and Mark Leone. Optimizing ML with Run-Time Code Generation. In *Proc. of the ACM SIGPLAN Conf. on Progr. Lang. Design and Impl.*, pages 137–148. ACM Press, May 1996.
22. Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: A Language and Compiler for Dynamic Code Generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, 1999.
23. R. Kent Dybvig. The Development of Chez Scheme. In *Proc. of the Eleventh ACM SIGPLAN Int. Conf. on Funct. Prog.*, pages 1–12, September 2006.
24. John C. Reynolds. The Dicoveries of Continuations. *Lisp and Symbolic Computation*, 6(3/4), 1993.
25. Zhong Shao and Adrew W. Appel. Space Efficient Closure Representations. In *Proc. of the 1994 ACM Conf. on LISP and Funct. Prog.*, pages 150–161, New York, 1994. ACM Press. ISBN 0-89791-643-3.
26. Christopher Strachey and C. P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. *Higher-Order and Symbolic Computation*, 13(1/2), 2000.

27. Gerald Jay Sussman and Guy L Steele Jr. Scheme: An Interpreter for Extended Lambda Calculus. In *MEMO 349, MIT AI LAB*, 1975.

# 10   Appendix

## 10.1   The AP Benchmark

```
(define (g x) (lambda (y) (+ x y)))
(define (f x) ((g (car x)) (cdr x)))
(define (mklist len modus)
    (if (= len 0) '()
        (cons (cons (random modus) (random modus))
              (mklist (- len 1) modus))))
(define (benchmark n modus) (map f (mklist n modus)))
```

## 10.2   The TK Benchmark

```
(define (tak-y x) (lambda (y) (tak-z y x)))
(define (tak-z y x)
  (lambda (z) (if (not (< y x)) z
                  (tak
                    (tak (- x 1) y z)
                    (tak (- y 1) z x)
                    (tak (- z 1) x y)))))
(define (tak x y z) (((tak-y x) y) z))
```

## 10.3   The ST Benchmark

```
(define (scaleT-by-depth T) (scale 0 T))
(define (scale d T) (map (scale-function d) T))
(define (scale-function d)
  (lambda (t) (if (number? t) (* d t) (scale (+ d 1) t))))
(define (mkbt d)
  (if (= d 0) '()
      (cons d (cons (mkbt (- d 1))
                    (cons (mkbt (- d 1)) '())))))
(define (benchmark x) (scaleT-by-depth (mkbt x)))
```