Published in: Journal of Computer Languages, Vol. 17, No. 4, pp. 251-267, Pergamon Press, 1992.

Closure generation based on viewing LAMBDA as EPSILON plus COMPILE

Marc Feeley
Computer Science Dept.
Brandeis University
415 South Street
Waltham, MA 02254

Guy Lapalme
Département d'informatique et de recherche opérationnelle,
Université de Montréal
P.O.B. 6128, Station A,
Montréal, Québec, H3C 3J7 (Canada)

Abstract

This paper describes a way of expressing λ–expressions (which produce closures) in terms of ϵ–expressions (λ–expressions containing only local and global variable references) and calls to an interactive compiler that compiles ϵ–expressions. This point of view is an interesting way of describing the semantics of λ–expressions and closure generation. It also leads to an efficient closure implementation both in time and space. A closure is uniformly represented as a piece of code instead of a compound object containing a code and environment pointer. This method can also be used to simulate closures in conventional dialects of Lisp.

KEY WORDS Closure implementation Compiling Lisp Scheme

1 INTRODUCTION

In many lexically scoped dialects of Lisp, e.g. Scheme[1-3], T[4,5] and Common Lisp[6], procedures are first class objects. They are defined by λ–expressions of the form `(lambda` *formal–argument–list body*`)`. The formal argument list declares the variables that will contain the actual argument values when the procedure is later called and the body indicates the expression that will be evaluated to compute the result. The evaluation of a λ–expression returns a procedure that "remembers" the current state of the environment (i.e. the set of current variable bindings).[1]

This operation is called closure. We speak of the resulting procedure as being a closure. In fact, we shall consider them to be synonymous in this paper.

Closures are a useful programming feature. They can be used to express data abstractions[1], to implement actors[7,8] and also to implement classes and provide data protection[9]. The work of Atkinson and Morrison[10] discusses their usefulness in implementing modules, separate compilation and database views. Closures have also been used to represent code in a Scheme compiler [11,12].

Unfortunately, λ–expressions give little insight into the method used to allocate and reference formal argument variables and into the way closures are represented and generated. This abstraction is fine from the user's point of view, but a more precise definition would be useful for implementing such languages and as a formal definition of the semantics of λ–expressions.

Our proposal is to express λ–expressions in terms of simpler constructs of the source language in an effort to make their evaluation mechanism more explicit and comprehensible. We are mainly concerned in evaluating this method in the context of a compiler based system. For the purpose of this paper, we shall choose Scheme as the source language. We assume the reader is familiar with Lisp or with one of its dialects.

Our approach uses ϵ–expressions (degenerate λ–expressions) and calls to an interactive compilation procedure to express λ–expressions. Compilation of λ–expressions consists of source to source transformations followed by the compilation of the resulting simpler forms. This is similar to the way macros are processed in many Lisp compilers[3,13,14]. Compilation is performed at two levels: statically for the compilation of the λ–expression and dynamically for each evaluation of the λ–expression (i.e. in the generation of each closure). Each closure generated is actually a piece of code. This clearly departs from the conventional representation of closures as data structures. This paper discusses this approach and analyzes its characteristics.

Before explaining our method further, we introduce a few concepts pertaining to closures.

2 DEFINITIONS

2.1 Variables

The possibility of naming computed values in order to ease their manipulation is a fundamental aspect of programming languages. This feature is provided by *variables*. A variable

[1] λ–expression evaluation should not be confused with the invocation of the procedure it produces. In Common Lisp, λ–expression evaluation is written as `(function (lambda ...))`.

designates a location where a value can be stored. A variable's value can be *accessed* through the use of the variable *reference* and *assignment* operations. Variable reference consists of fetching a value from the location designated by the variable and variable assignment consists of storing a value in it. Certain constructs are used to create new variables and give them names. These are known as the *binding constructs*. In Scheme, the most fundamental binding construct is the `lambda` special form (i.e. λ–expression). All other binding constructs, such as the `let`, `letrec`, `define` and `do` special forms, can be described in terms of λ–expressions. Without any loss of generality, we shall consider that the only binding construct available is the λ–expression. λ–expressions are of the form `(lambda` *formal–argument–list body*`)`. The formal argument list indicates the name of the variables that are created. In accordance with lexical scoping rules, the only region of the program where these name–variable associations are effective is the body of the λ–expression that declares them. Any use of a name, in a variable access, refers to the variable associated with this name in the innermost λ–expression that binds the name and contains the use. Distinct variables can have the same name. However, at most one of these variables is accessible at a given place in the program according to the previous rule.

Variables are either *bound* or *free* with respect to each particular λ–expression. A variable is bound with respect to a λ–expression if it is declared in the formal argument list of the λ–expression in question, otherwise, it is free. We shall say that a variable reference is *midway* with respect to a λ–expression if it is declared by an enclosing λ–expression. A variable that is free with respect to all λ–expressions is said to be *global*. In this paper, a variable all of whose accesses are directly surrounded by the λ–expression that binds the variable is called a *local* variable. A variable that has at least one access that is midway with respect to the λ–expression directly surrounding each access is called a *closed* variable. Global variables are usually declared by `define` special forms entered at top–level whereas local and closed variables are declared in the formal argument list of λ–expressions. Global, local and closed variables form distinct classes and every variable is a member of one and only one of these classes. For example, in the expression:

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

`f`, `list` and `+` are global variables, `y` and `z` are local variables and `x` is a closed variable (because it is midway with respect to the inner λ–expression in which `x` is accessed).[2]

2.2 Environments

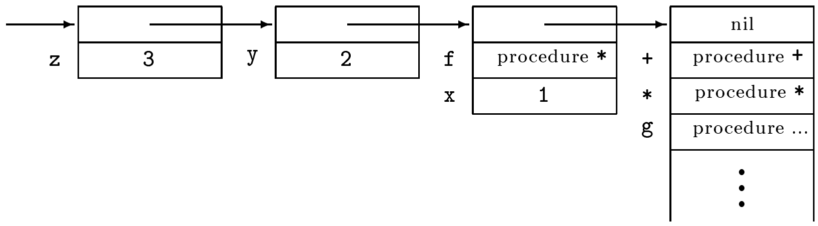
The set of all name–variable associations in effect at some point in the program is known as the *environment* in effect at that point. The environment is used to find the variable that is currently associated with a particular name when variable reference and assignment are performed. However, this does not mean that the structure representing the environment necessarily contains the names of the variables currently accessible. All that matters is that the variable currently associated with a given name can be found at the same place relative to

[2]The form `(define (`*name* ...`)` *body*`)` is just a more elegant way of writing the equivalent form `(define` *name* `(lambda (`...`)` *body*`))`.

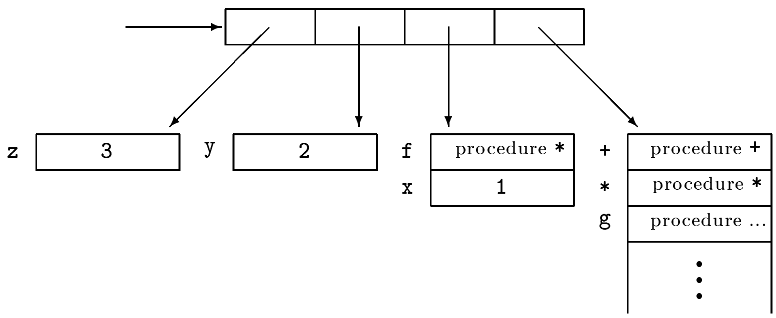
the structure representing the environment. The classical way of implementing environments for lexically scoped languages is with a linked chain of frames[15,16]. At a given time, each frame contained in the chain corresponds to one of the λ–expressions in which the current execution point is lexically nested. The frames are ordered in the chain by decreasing nesting level of the λ–expression to which they correspond. Each frame contains the values of the formal argument variables of the corresponding λ–expression as well as a pointer to the following frame in the chain (known as the static link). The last frame of the chain contains the global variables. When new variables are created (that is when a procedure, the value of a λ–expression, is invoked), a new frame containing the initial values of the variables is added at the front of the environment chain. For example, consider the following expression entered at top–level:

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

When the expression `((g * 1) 2)` is evaluated (resulting in the value 9), the environment in which the expression `(f z z)` is evaluated is:



Formal argument variables are accessed by going through the chain to the appropriate depth and accessing the appropriate slot of the frame obtained. For each particular variable access the depth and slot at which the variable is found does not change from one evaluation to the next. In the previous example, referencing the variable `f` in the body of the innermost λ–expression would involve going through the chain to the third frame and fetching the second slot. Since the time required to get to a frame is proportional to its depth in the chain this method can be expensive if the nesting level of λ–expressions is great. This can be alleviated by using a vector of pointers to the frames (known as a display) instead of linking them with static links[17]. Variable access then consists of an indirect addressing through the appropriate display pointer. Using a display, the environment of the previous example would be:



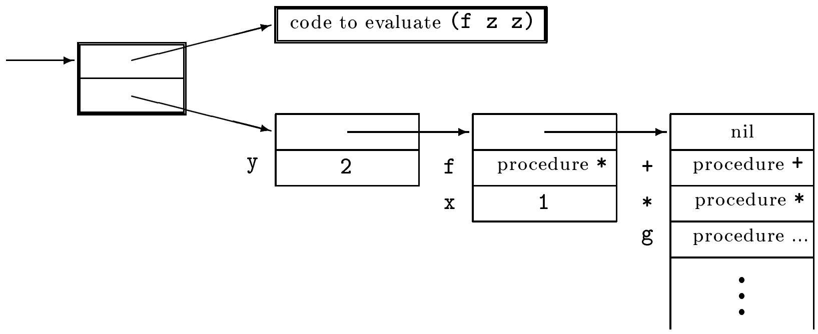
and a reference to the variable `f` would consist of fetching the first slot of the frame to which the third entry of the display points to.

2.3 Closures

The main operations permitted on closures are *creation* and *invocation*. Creation is obtained by the evaluation of a *definition*, which is a λ–expression. The result of the evaluation is a closure. Like other objects, closures can be stored in variables and data structures, passed as arguments to procedures, returned as the result of procedures, etc... Invocation is obtained by the evaluation of a *combination*. A combination is a list of the form `(` *operator operand1...*`)` whose first element is not the keyword of a special form.[3]

The evaluation of the expression *operator* must result in a closure. The result of the invocation is the result of the evaluation of the body of the closure's definition in an environment consisting of the environment which was in effect when the closure was created augmented by new name–variable associations. These new name–variable associations consist of each name in the formal argument list of the closure's definition associated with a newly created variable initialized to the corresponding operand value of the combination. In case of name conflicts, the new association overrides the one which was in effect when the closure was created.

The environment which was in effect when the closure's definition was evaluated is used when the definition's body is evaluated. Consequently, there must be a way to recover it when the closure is invoked. The classical way of implementing closures is with a data structure consisting of two parts. The first part is the code to evaluate the body of the closure, and the second part is an environment. Closure generation consists of the packaging of these two parts into a new object. In order to save space, the "code" part is actually a pointer to the code since it remains the same for all closures created for the same definition. The environment part is the environment which was in effect when the closure was created. For instance, if we assume that environments are implemented as a linked chain of frames, a closure generated by the evaluation of the innermost λ–expression of the previous example would be:



When this closure is invoked, a new environment is constructed by allocating and linking a frame containing a single slot in front of the closure's environment. This slot, corresponding

[3]Invocation can also be performed by using the `apply` procedure or, in Common Lisp, by the form `(funcall` *operator operand1...*`)`

to the variable `z`, is initialized to the value of the operand that is passed to the closure. This new environment is passed to the code evaluating the closure's body in order for it to access the variables `f` and `z`. Frame allocation for the variables of the formal argument list can also be performed by the closure's code, in which case the operand values in addition to the closure's environment must be passed to it. Note that the environment in which the closure's body is evaluated has nothing to do with the environment in effect when the closure is invoked. This is not the case of dynamically scoped dialects of Lisp which, in a sense, allocate the frame containing the formal argument variables in front of the environment in effect when the closure is invoked.

2.4 Efficiency considerations

Closures, like all other objects, have indefinite extent. They exist until it is no longer possible to reference them. This implies that closures, as well as the environments they contain, must generally be allocated from a heap. This often leads to inefficient implementations due to the cost involved in allocating and maintaining variables on the heap. This inefficiency can be avoided in certain situations by allocating environments on the control stack. This can be performed by a clever compiler when it detects that the environment's existence is bounded[3,7]. Consider the expression:

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

When the closure resulting from the evaluation of this expression is called, a closure that uses the closed variable `x` is returned. The frame containing the variable `x` must be allocated from the heap because there is always a possibility that this last closure be invoked after the outer λ–expression's body is evaluated. For example, the closure could be stored in a variable and called much later. Now consider the following expression, obtained by replacing the reference to the variable `z` by `(z 1)` in the previous expression:

```
(lambda (x)
  ((lambda (z) (z 1))
   (lambda (y) (+ x y))))
```

The closure referencing the variable `x`, i.e. the closure created by the evaluation of the λ–expression `(lambda (y) (+ x y))`, can not be accessed outside the λ–expression that declares the variable `x`. This means that the only moment that the variable `x` can be accessed is during the execution of the body of the closure that allocated it. Consequently, it is possible to allocate the variable `x` on the control stack and deallocate it upon exit of the closure. To perform such an optimization, the compiler must have some knowledge of the semantics of λ–expressions, closure invocation and data manipulation primitives. Another possible optimization is to consistently allocate environments on the control stack and to move them to the heap only when necessary[18]. This last method is particularly well suited for an interpreter based system because the decision to move the environment to the heap can be taken at run time.

Certain implementation tricks can be used to represent environments more efficiently. Global variables, for example, can be implemented efficiently by using a slot in the record of the symbol that represents the variable. This is possible because there exists only one global environment (i.e. global variables designate a single location throughout the entire program). Allocation is performed when the symbol is first introduced and access can be as simple as indirect addressing of the symbol pointer. Local variables can also be implemented efficiently by allocating them on the control stack or even in registers. This is possible because, by definition, such a variable is never accessed by another closure than the one that allocated it. Consequently, the existence of a local variable is limited by the execution of the body of the closure that declares it and can be deallocated upon return.

Thus the environment remembered by a closure is used to access its closed variables. If closed variables were not accessed, closure generation would not be needed since no environment would have to be remembered. A closure would simply be a pointer to the code evaluating its body. Invoking closures would be straightforward since no environment would have to be explicitly manipulated. A simple jump would suffice. In fact this idea can be implemented as an optimization performed by the compiler. In general though, closures that access closed variables contain both a code and environment pointer.

Our closure implementation method takes a different point of view unifying the concepts of environment and closure. Instead of being a data structure containing pointers to the code and to the environment, a closure is a piece of code which itself contains the required environment information. Closure generation consists of the generation of this piece of code instead of the creation of the code–environment pair. The following section describes how this can be done by transforming λ–expressions into simpler forms. Later on, we will compare our closure implementation method with the one just described in this section.

3 TRANSLATING λ–EXPRESSIONS

This section describes our closure implementation method which is based on a source to source translation. We will first describe the basic idea and then refine it to improve its efficiency.

3.1 Basic method

For the moment, we do not consider variable assignment to closed variables. This aspect, which brings additional problems, will be considered later on. The target constructs used in the translation of λ–expressions are ϵ–expressions and calls to the `compile` procedure.

As stated above, ϵ–expressions are degenerate λ–expressions. They are denoted by the keyword `epsilon` instead of `lambda` and their body can only contain references to local and global variables. All free variables of an ϵ–expression are treated as global variables (i.e. the concept of midway and closed variable does not exist for ϵ–expressions). In a way, ϵ–expressions simply denote constant pieces of code (i.e. closures with no environment information) and all evaluations of the same ϵ–expression yield the same procedure. The reason we use ϵ–expressions in our transformation is that variable allocation and reference are simple and can be implemented efficiently. As stated in the previous section, the formal argument variables of ϵ–expressions can be allocated on the control stack or in registers.

The `compile` procedure takes a single argument representing an ϵ–expression and returns a new procedure that implements the given ϵ–expression in the global environment. For example, the evaluation of the expression `(compile '(epsilon (x) (+ x 1)))` and `(epsilon (x) (+ x 1))` result in equivalent procedures. The `compile` procedure can be thought of as a procedure constructor. Thus it shares some common points with the evaluation of a λ–expression which also creates procedures. This aspect will be exploited in our closure implementation method.

The transformation process uses the β–conversion principle of the λ–calculus[19]. The β–conversion principle states that when an abstraction (i.e. the λ–calculus equivalent of a procedure) is applied to some argument, every occurrence of the formal argument in the body of the abstraction can be replaced by the actual argument value. This principle permits us to implement closed variables and, consequently, closures. A λ–expression is transformed into a call to the `compile` procedure. The argument is the list representation of the λ–expression with the keyword `lambda` substituted by `epsilon` and with each reference to a closed variable replaced by the quoted current value of the variable. For example, the expression:

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

is transformed into:

```
(define (adder x)
  (compile
    `(epsilon (y) (+ ',x y))))
```

Each time it is invoked, the procedure `adder` constructs the list representation of an ϵ–expression and compiles it. The resulting procedure is perfectly equivalent to the procedure that would have resulted from the evaluation of the corresponding λ–expression. Each time `x` is referenced, the value which `x` had when the closure was generated is obtained. This is exactly what is expected. In a way, we could say that the value of the variable `x` is frozen into the code of the newly created procedure. This is coherent with the β–conversion principle stated above. In general, all the closed variables accessed in a λ–expression are frozen in the code of the corresponding closure.

We might question the efficiency of this transformation. From the stand point of its utilization, the execution time of the resulting closure is in a way “optimal” since references to closed variables are replaced by references to constants and it is not necessary to perform an environment lookup to access their value. On the other hand, closure generation is somewhat time consuming because each time a closure is generated a structure representing an ϵ–expression must be constructed and compiled. Since the structure to compile is arbitrarily complex much time and space may be required to generate a closure in this fashion. The method may be useful in a context where closures are created rarely and invoked frequently. However, in a more realistic context, a better trade–off between closure execution time and closure generation time and space may be preferable.

3.2 Improved closure generation

The problem discussed in the previous section can be alleviated by observing that most of the expression to be compiled remains the same. It can be compiled once and for all and used as a constant in the translated form. The constant part will take the form of a procedure that we shall call the *B-procedure* (i.e. "body procedure"). It is expressed as an ϵ–expression that has the same body as the λ–expression to transform and has an argument list that is the concatenation of the variables contained in the argument list of the λ–expression with the list of the closed variables that are referenced in its body. For example, if the λ–expression `(lambda` $(v_1 \ldots v_p)$ *body*) references the closed variables $c_1 ... c_q$, then the corresponding B-procedure is expressed by the following ϵ–expression:
`(epsilon` $(v_1 \ldots v_p\ c_1 \ldots c_q)$ *body*)
This has the effect of localizing closed variables. What remains to be done is to pass along the correct value for each of these variables when the closure is invoked. This is performed by an interface procedure (*I-procedure*) that is expressed as an ϵ–expression with the same argument list as the λ–expression to transform and that calls the B-procedure. The complete transformation takes the λ–expression `(lambda` $(v_1 \ldots v_p)$ *body*) and translates it into:

```
(compile
  `(epsilon (v1...vp)
     (',(epsilon (v1...vp c1...cq) body) v1...vp ',c1...',cq)))
```

Applying this transformation to our previous example yields:

```
(define (adder x)
  (compile
    `(epsilon (y)
       (',(epsilon (y x) (+ x y)) y ',x))))
```

The ϵ–expression corresponding to the B-procedure (i.e. `(epsilon` $(v_1 \ldots v_p\ c_1 \ldots c_q)$ *body*)) is compiled once and is shared by all the closures generated by the transformed λ–expression. Compilation needs only to be performed on the ϵ–expression corresponding to the I-procedure when a closure is generated. The resulting I-procedure is the closure corresponding to the λ–expression that has been transformed. This transformation corresponds to the "lambda-lifting" transformation which is used in supercombinator graph reduction [20]. This article shows how it can be applied in the environment based approach to programming language implementation.

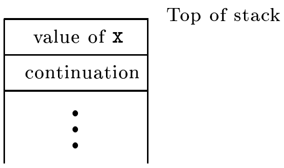
3.3 Internal aspect of closure generation

To fully explain the transformation process, consider the code that needs to be generated when a closure is created. The code produced is for a hypothetical stack based architecture with the following procedure invocation convention:

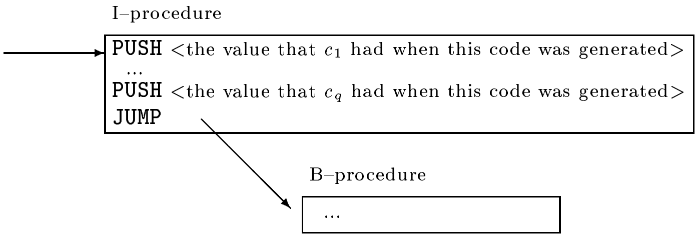
- on entry to a procedure of n arguments, the n topmost stack locations contain the actual argument values (the last is on top of the stack).

- the continuation (i.e. return address) of the procedure is located, on the stack, directly below the actual argument values.

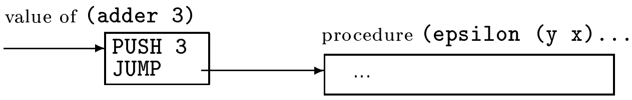
Therefore, on entry to the I-procedure, the $p+1$ topmost stack locations contain the continuation and the values of the variables $v_1 ... v_p$. For instance, on entry to the closure returned by the call `(adder 3)`, the stack has the following format:



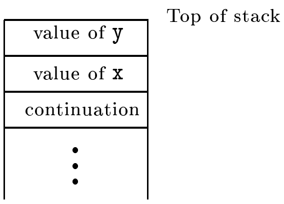
On the other hand, the B-procedure expects the $p+q+1$ topmost stack locations to contain the continuation and the values of the variables $v_1 ... v_p$ and $c_1 ... c_q$. Since the call to the B-procedure is the last in the body of the I-procedure, the B-procedure's continuation is identical to the one passed to the I-procedure. Consequently, the I-procedure needs only to push the values corresponding to the closed variables $c_1 ... c_q$ on the stack and to jump to the B-procedure. The code produced for the I-procedure, and thus for a closure, will be:



For example, the closure generated by the call `(adder 3)` is the piece of code:

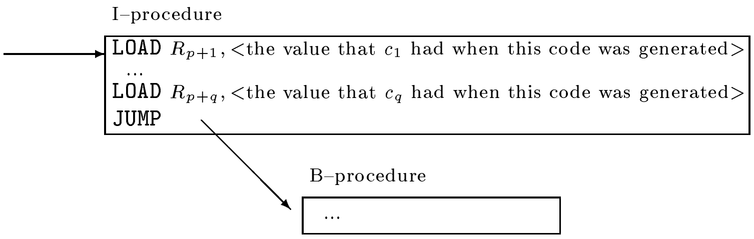


and on entry to the B-procedure the stack has the following format:



When the variable `x` is referenced in the B-procedure, the value 3 is obtained. Once the B-procedure has finished computing the result (or more precisely, once the procedure `+` has finished computing the sum), the two topmost entries of the stack are discarded and execution resumes at the continuation.

The method can easily be extended to systems having other procedure invocation conventions. For a register based architectures that passes the arguments to a procedure of n arguments in the registers $R_1..R_n$ and the continuation in R_0, the following code can be produced for the I-procedure:



One way or another, the code generated for an I-procedure is simple and regular. This comes from the fact that it is defined by an ϵ–expression that has a regular structure. The whole generality offered by the `compile` procedure is therefore not needed and a specialized compilation procedure can be used instead. In order to generate the appropriate code, this procedure, that we shall call `closure`, needs to know the B-procedure and the value of the closed variables $c_1 ... c_q$. The value of p may also be needed by `closure` depending on the procedure invocation convention. Using `closure`, the transformation of the λ–expression `(lambda (`$v_1 \ldots v_p$`)` *body*`)` would produce the form:

`(closure` $c_1 \ldots c_q$ `(epsilon (`$v_1 \ldots v_p$ $c_1 \ldots c_q$`)` *body*`))`

.

4 ASSIGNMENT

Up to now, we have not considered assignment to closed variables. If our method stayed as it is, assignment to ϵ–expression variables would only modify the corresponding local variable. But an assignment to a closed variable must affect all closures in which it participates. This is essential to implement mutable data abstractions with closures[1].

A closed variable that is assigned to, will be called a *mutable variable*. Assignment to such a variable can be handled in the following way. When a procedure that declares a mutable variable is entered, a *cell* (i.e. a frame containing a single value) is allocated from the heap and the corresponding actual argument value is stored in the cell. Assignment consists of storing a value in the cell and reference (i.e. variable evaluation) consists of fetching the cell's contents. Closures that share a same mutable variable will in fact share the cell which is associated to the variable.

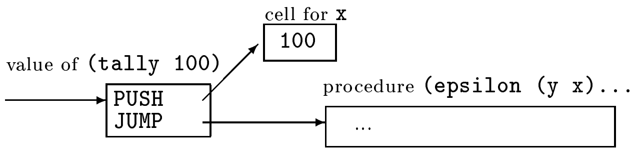
Minor modifications to the previously described transformation process are needed to deal with assignment correctly. Cells are allocated and initialized, through the use of the procedure `cell`, at the beginning of procedures that declare mutable variables. In order to use them, the address of the cells are saved in temporary variables, of the same name, declared through the use of an ϵ–expression. Reference and assignment to a mutable variable are transformed into calls to the `fetch` and `store` procedures respectively.[4] Also, when a closure is generated, the value used in the I-procedure is the mutable variable's cell address, not its contents. For example, the form:

```
(define (tally x)
  (lambda (y)
    (set! x (+ x y))
    x))
```

is transformed into:

```
(define (tally x)
  ((epsilon (x)
     (closure x (epsilon (y x)
                  (store x (+ (fetch x) y))
                  (fetch x))))
   (cell x)))
```

The closure generated by the call `(tally 100)` is the piece of code:



Mutable variables thus occupy slightly more space and are more expensive to access than other variables since an additional indirection must be performed to access them. This is not a major inconvenience especially considering that the programming style advocated in applicative languages is usually without side–effect.

It is worthwhile noting that cells could be allocated for all closed variables regardless of whether they are mutable or not. Not having to allocate cells for closed variables that are not mutable can be regarded as an optimization, possible because the value of such variables never changes.

5 TRANSLATION ALGORITHM

The translation of λ–expressions is a two pass process. The first pass consists of scanning the input expression to construct a graph structure that represents it. Each node of the graph represents a subexpression and contains additional information collected during this pass. In particular, the structure representing variables indicates if the variable is global,

[4] The procedures `cell`, `fetch` and `store` can be implemented with dotted–pairs:

```
(define (cell v)    (list v))
(define (fetch b)   (car b))
(define (store b v) (set-car! b v))
```

local or closed and if it is used in an assignment. During this pass, a structure describing the environment is maintained in order to be able to find the structure that represents a particular variable. The information concerning variables is updated when variable references and assignments are discovered in the expression. A variable declared in a λ–expression is initially local and becomes closed if it is accessed in the body of another λ–expression. Similarly, variables are initially non–assigned and become assigned if an assignment to that variable is found. Also, each node representing a λ–expression contains the list of closed variables that are used in its body. This list is initially empty and is augmented each time an access to a closed variable is found while processing it's body. Note that this analysis is simple because Scheme is lexically scoped and that arguments are always passed by value. A much more complex analysis, possibly involving interprocedural analysis, would be needed if this was not the case.

The second pass scans the graph structure obtained and reconstructs the corresponding Scheme expression based on the accumulated information found in the graph. There are three cases where the portion of expression reconstructed is not identical to that of the original expression. The first occurs for a reference to a closed variable that is also used in an assignment. In this case the expression `(fetch` *var*`)` must be produced instead of *var*. The second case occurs for an assignment to a closed variable, `(store` *var val*`)` must be produced instead of `(set!` *var val*`)`. Finally, for a λ–expression the expression `(compile '(epsilon ...))` or `(closure ... (epsilon ...))` must be produced as previously described. Cells must also be created for λ–expressions that declare mutable variables.

We have implemented the translation process in Scheme. The program is about 150 lines long and only handles the basic special forms of Scheme since they can be used to express the other special forms. The main procedure is `lambda->epsilon+compile`; it takes an expression and translates it into an equivalent expression containing ϵ–expressions and calls to the `compile` procedure. Here is a sample translation as performed by our program:
Call:

```
(lambda->epsilon+compile
  '(define special-cons
     (lambda (x y)
       (lambda (msg)
         (cond
           ((eq? msg 'car) x)
           ((eq? msg 'cdr) y)
           ((eq? msg 'set-car!) (lambda (v) (set! x v))))))))
```

Result:

```
(define special-cons
  (epsilon (x y)
    ((epsilon (x y)
       (compile
         (list 'epsilon '(msg)
```

```
      (list
        (list 'quote
              (epsilon (msg y x)
                (cond
                  ((eq? msg 'car) (fetch x))
                  ((eq? msg 'cdr) y)
                  ((eq? msg 'set-car!)
                   (compile
                     (list 'epsilon '(v)
                       (list
                         (list 'quote
                               (epsilon (v x) (store x v)))
                         'v
                         (list 'quote x))))))))
        'msg
        (list 'quote y)
        (list 'quote x)))))
  (cell x)
  y)))
```

This example, inspired from Reference 1, is an implementation of special dotted pairs using closures (only the `car` part of the pair can be modified). The example contains a mutable variable and a non–mutable closed variable. As can be seen, the closure created for the most internal λ–expression will not retain the closed variable `y` since it is not used in it's body. This is a natural consequence of the translation algorithm.

6 PERFORMANCE

The general performance of a Scheme system depends on many parameters such as the internal representation of objects, the frequency of usage of each Scheme construct, the context in which they are used and many other parameters that are not directly related to the closure implementation method. Our goal is not to study the impact of all these parameters on the performance of a Scheme system; we will limit ourselves to the aspects that are directly related to closures.

Two important aspects to analyze are the amount of memory used and the execution time. In order to appreciate the merits of our method, we proceed to a comparative analysis with the classical closure implementation method. To assist this analysis we have conducted a study of the execution characteristics of Scheme programs which is summarized in Figure 1.

First we define more precisely the classical method used in our analysis. It consists in representing closures as a data structure containing two fields. The first is a pointer to the closure's code and the second is a pointer to the definition environment. The environment is represented as a linked chain of frames that only contains the closed variables. Local variables are allocated on the execution stack and global variables are allocated in the symbol that represents the variable. Hence, there is no frame associated with the global environment in

	average %	fib %	sort %	*sort %	queens %	deriv %	dderiv %	interp %	comp %	exec %
Primitive constructs										
Variable reference	53.8	48.0	59.9	47.1	55.9	51.5	53.4	55.9	49.1	63.8
Procedure application	29.7	28.0	30.4	24.0	29.8	29.6	29.9	29.9	31.1	34.8
Conditionnal expression	8.7	8.0	8.7	11.6	8.9	9.5	8.5	10.4	11.6	1.3
Constant reference	7.0	16.0	0.9	14.4	5.2	9.5	8.3	3.8	4.9	0.1
Procedure creation	0.6			2.8					3.1	
Variable references										
Global	49.6	58.3	50.7	36.7	53.1	54.4	53.0	52.1	63.0	25.7
Local	43.8	41.7	49.3	49.0	46.6	45.6	47.0	46.7	35.3	33.6
Closed in frame 1	6.0			14.3	0.3			1.2	1.4	37.0
in frame > 1	0.4								0.2	3.6
Procedure application type										
Primitive procedure	71.7	71.2	82.7	48.0	83.6	74.7	71.8	80.5	77.9	55.2
Closure def env len $= 0$	19.7	28.8	17.3	23.9	15.8	25.3	28.2	18.7	19.9	0.2
def env len $= 1$	8.0			28.1	0.6			0.8	1.8	41.3
def env len > 1	0.4								0.4	3.3
Procedure application location										
Non-tail application	82.6	85.6	82.9	88.0	91.6	83.3	80.5	82.0	88.2	61.9
Tail application	17.3	14.4	17.1	12.0	8.4	16.7	19.5	18.0	11.8	38.1

`fib`	fibonacci function, `(fib 10)`
`sort`	selection sort of a list
`*sort`	selection sort of a list represented with closures
`queens`	solution to the 'n' queens puzzle using lists
`deriv`	symbolic derivation
`dderiv`	data driven symbolic derivation
`interp`	small Scheme interpreter evaluating `(fib 10)`
`comp`	small Scheme compiler compiling fibonacci function into a closure tree
`exec`	execution of the code generated by `comp` for the call `(fib 10)`

`deriv` and `dderiv` programs are from Reference 23, all others can be found in Reference 11.

Figure 1: Dynamic characteristics of some Scheme programs

the chain. The closure is responsible for the allocation of the frame in front of the environment chain for its parameters that are closed. As additional optimizations, the last frame of the chain does not contain a static link and empty frames are never placed on the environment. Throughout execution, the current environment is pointed to by a global pointer (e.g. a register).

The procedure invocation convention is similar for both methods and consists in pushing the continuation on the execution stack followed by each argument of the procedure application. The major difference resides in how control is passed to the procedure. With our method, a simple jump to the pointer denoting the procedure is needed. No environment pointer, besides the stack pointer, needs to be manipulated. With the classical closure implementation method, each part of the code–environment pair must be extracted, the current environment must be set to the closure's environment and finally a jump to the code must be performed. Thus it is reasonable to believe that the procedure invocation code will be shorter and faster with our method.

Furthermore, when the application is non–terminal (e.g. when evaluating an argument to an application), which according to Figure 1 is a frequent case, the classical method must save the caller's environment on the execution stack and restore it upon return in order to access any closed variables in the remaining calling procedure's body. This operation could be avoided if the environment pointer was always stored on the stack but this would increase the cost of each access to closed variables. This operation is unnecessary for our method since all non–global variables are located on the stack.

Also, if all procedures are represented in the same way with a code–environment pair, the application of procedures with a null definition environment (i.e. primitive procedures and most user defined procedures) will perform the general treatment even if this is not really necessary. On the other hand, if they have a distinct representation, they will have to be distinguished for each application because the type of procedure that will be called is unknown at compile time. This is not the case for our method since primitive procedures as well as true closures are both represented by a piece of code and a simple jump suffices to pass them control. We believe this aspect is of great importance especially since the application of such procedures is by far the most frequent in programs and that it will have a significant impact on overall performance.

Some systems circumvent this problem by treating combinations whose operator is a global variable initially bound to a predefined procedure (such as `+`, `car`, `read`) in a special way. These systems will generate specific code for such combinations assuming that they will always invoke the corresponding predefined procedures. This leads to efficient code but of course violates Scheme semantics since global variable initially bound to a predefined procedure cannot be changed.

As explained earlier, our closure implementation method provides fast access to closed variables. An access to a non–mutable closed variable consists of an indirection with displacement relative to the stack pointer. For a mutable variable, an additional indirection is needed. With the classical method, accessing a closed variable is performed by an indirection with displacement relative to the frame that contains the variable in the chain. This frame is obtained by descending the chain to the right level. If the closed variable is in the n^{th} frame from the beginning of the chain, $n-1$ indirections are needed. These indirections can be reduced to a single indirection with displacement if the environment is represented with a display of frames. Therefore, the only situation where our method involves more instructions and time to access a closed variable is when the variable accessed is mutable and in the first frame, the difference being of an additional indirection. In the other cases, our method is at least as good as the classical method and progressively better as the depth of the frame containing the variable in the chain increases. Given the results of Figure 1, it seems that both methods will yield similar performance for variable access since closed variables accessed are most often in the first frame.

The space occupied by a closure is more difficult to analyze. This difficulty comes in part from the fact that certain portions of the closures may be shared between many closures. Also, the additional space required to represent the objects in memory (e.g. GC information, type tag, length, etc...) should be considered. However, this varies from one system to another. In order to simplify the analysis, we will suppose that this additional space is null and we won't count the space for the code of the closure's body in the space occupied by the closure itself. Consider our method. The space occupied by a closure depends on the number of mutable closed variables (N_M) and non–mutable closed variables (N_N) used by the closure, on the size of the opcodes for the instructions `PUSH` (S_P) and `JUMP` (S_J) and on the size of an address (S_A). The total space occupied by a closure created with our method is thus:

$$(N_M + N_N) * (S_P + S_A) + S_J + S_A + N_M * S_A$$

Note that the last term of this equation corresponds to the cells associated with the mutable variables. These cells can be shared between closures. Now consider the classical method. The space occupied by closure depends on the number of closed variables in the environment chain (N_C), on the number of frames in the chain (N_F) and on S_A. The total space occupied by a closure created with the classical method is thus:

$$2 * S_A + N_C * S_A + (N_F - 1) * S_A$$

Here, the last two terms correspond to the space occupied by the environment chain which can be shared between closures. However, this environment can contain an arbitrary number (X) of closed variables that are not used by the closure (i.e. $N_C = N_M + N_N + X$, $X \geq 0$). This particularity, which happens when many closures that use different closed variables are created from the same environment, makes the comparison between the two methods harder.

The most favorable case for the classical method occurs when a large number of closures share a same environment. In that case, practically no more space is needed per closure than that needed for the code–environment pair (i.e. $2 * S_A$). The worst case happens when X is large, the environment chain is long and the definition environment is not shared.

In order to measure and compare the performance of both methods, we have performed some benchmarks. We translated four Scheme procedure definitions in MC68000 assembler by hand using each closure implementation method. These definitions, the calls performed and the closures created during the execution of the procedures are shown in Figure 2. We measured four characteristics: the space occupied by the MC68000 code corresponding to the definition, the space allocated for the closures during the call of the defined procedure, the time required to execute the procedure call and the time required to execute the resulting closure (which is, in the last three cases, a one argument closure). Figure 3 shows the results when executed on a Macintosh Plus computer.

The space occupied per closure can be computed using these informations: on the MC68000, $S_A = 4$ bytes and $S_P = S_J = 2$ bytes (since the instructions `PUSH` and `JUMP` correspond to `MOVE.L #` *pnt*`,-(SP)` and `JMP` *pnt*`.L`). In the case of our closure implementation method, we have open coded the closure generation code instead of calling the procedures `compile` or `closure`. This lengthened the code used by the definitions and diminished the closure generation time.

The results show that the code for procedures implemented using our method is 10–15% shorter than and, except for a special case discussed later on, as fast as the procedures implemented with the classical method. Since these procedures mainly perform closure creations, this observation is attributable to the differences from the stand point of closure generation.

The results also indicate that, for the benchmarks performed, the space occupied by the closures created with our method is up to 25% more that for the classical method. However, if we only consider the closures that can still be referenced after the procedures are executed (i.e. the last closure generated in each test), then the space occupied is up to 40% less for our method. The closures created by our method have the advantage of only retaining the closed variables that are necessary for their own execution. On the other hand, closed variables are not shared between closures. Therefore, the space occupied by the closures depend on the context and from this stand point, none is clearly superior to the other in all contexts.

Finally, the results show that the closures created with our method execute 15–30% faster than the closures created with the classical method. This figure measures the time for referencing closed variables and to invoke closures.

The results are therefore favorable for our closure implementation method. However, our method has hidden weaknesses as demonstrated by the execution of the procedure `test3` which is 25% slower than when implemented with the classical method. This procedure contains deeply nested λ–expressions and the innermost λ–expression uses variables that are closed with respect to the enclosing λ–expressions. In this case, the closures created for enclosing λ–expressions must retain these variables in order to be able to create the closures for the inner λ–expressions. A certain effort is required to create these closures and to push the values on the stack when the closures are invoked. In the procedure `test3`, only the last closure generated uses the closed variables `a`, `b` and `c`. Thus the effort expended in pushing the values on the stack in order to speed up closed variable access is not really useful since these closed variables are only referenced when the closure is invoked later on. According to Figure 1, this case occurs infrequently.

Closer examination of our method suggests a similarity with a cache. On entry to a closure, the value of the closed variables it references are copied on the control stack (or in registers). This puts the values in a place which can be accessed efficiently. Though copying costs time, it is expected that these values will be accessed frequently and thus, that the overall execution time will be lowered. If many values are copied and they are seldom referenced, the copying time may overwhelm the access time saved. In such a case, a hybrid closure implementation can be preferable. It consists in representing environments in the same way as the classical method and representing closures as a piece of code of the form:

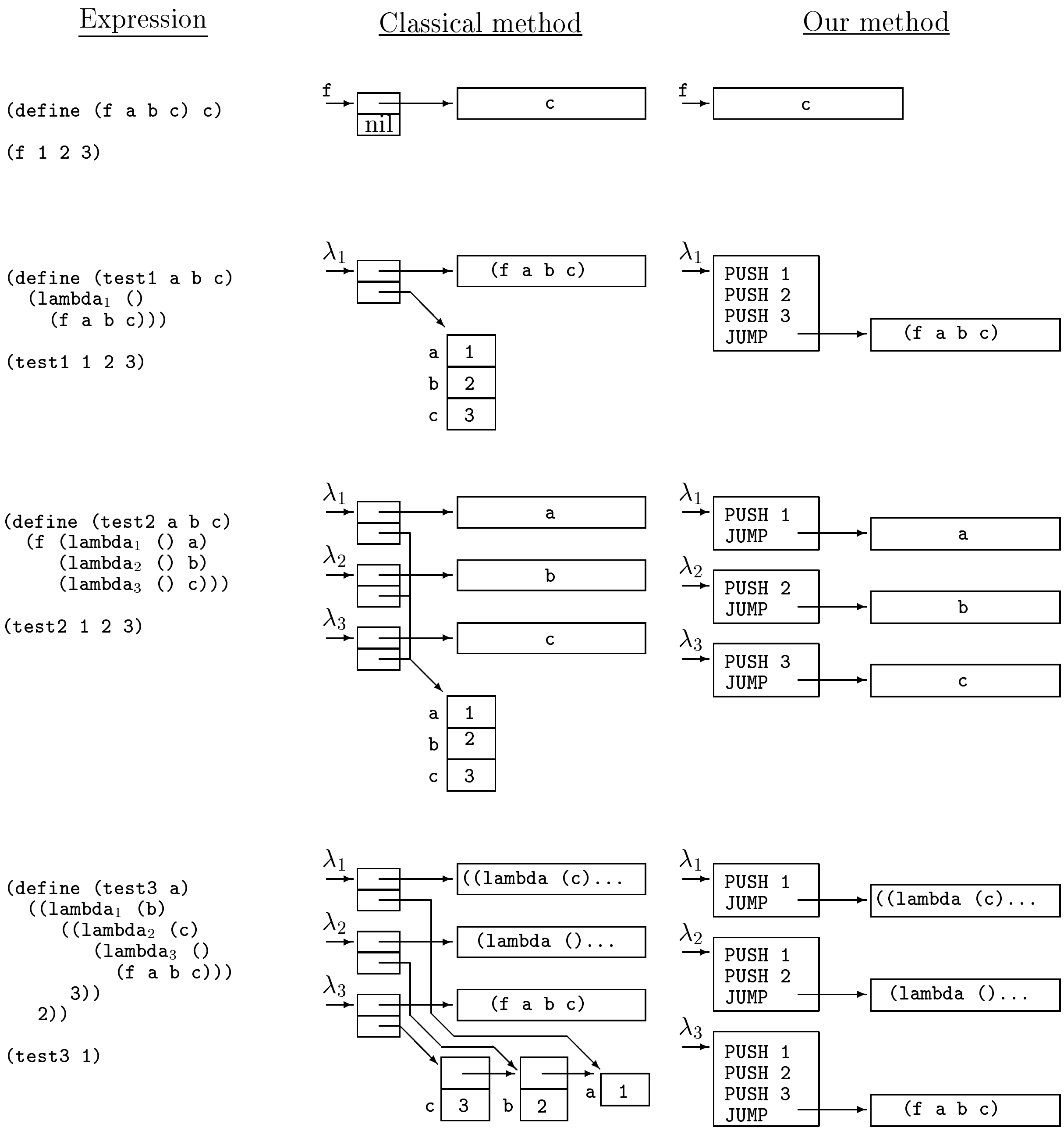
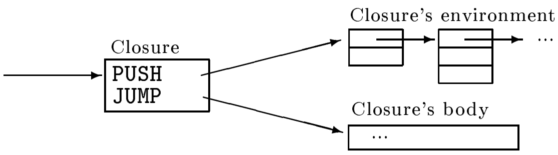


Figure 2: Layout of closures

Call	space for closure(s) (bytes)	space for call (bytes)	time for call (μsec)	time to call result (μsec)
Our method				
`(f 1 2 3)`	6		27	
`(test1 1 2 3)`	128	24	50	40
`(test2 1 2 3)`	164	36	80	21
`(test3 1)`	170	54	101	40
Classical method				
`(f 1 2 3)`	6		34	
`(test1 1 2 3)`	152	20	51	51
`(test2 1 2 3)`	180	36	81	25
`(test3 1)`	192	44	80	56

Figure 3: Test results



This method has the same characteristics as the classical method except that the closures occupy slightly more space (i.e. $S_P + S_J$) and are invoked by jumping to them.

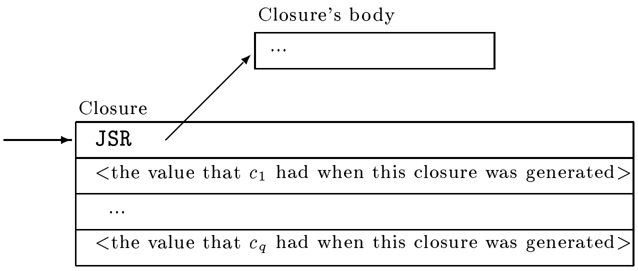
An advantage of lexically scoped languages is that all accesses to a variable occur in the body of the λ–expression that declares the variable. Consequently, the compiler can choose, with a local analysis of a program, the representation it judges more efficient for a particular environment when it compiles a λ–expression. Our standard method could be used under certain circumstances and the hybrid method in others. The prerequisite being that the procedure invocation convention is the same for each representation, which is the case of our standard and hybrid method.

7 RELATED METHODS

We think the central idea of our method is that closures should be a true piece of code instead of a data structure which represents a piece of code for an abstract machine as is usually done. In this way procedure application is a simple jump and the caller need not know if the procedure has closed variables or not in order to call it efficiently. Other closure representations can be adapted to include this idea.

In his Functional Abstract Machine, Cardelli[21,22] uses a single frame representation for closures. The first slot contains a pointer to the code of the closure's body and the remaining slots correspond to the closed variables of the closure. When the closure is invoked a jump to the closure's body is performed and a pointer to the frame itself is passed (in order for the closure's body to access the closed variables). This is very similar to our method except that we place `PUSH` opcodes between the slots and a `JUMP` opcode before the pointer which is at the end of the frame.

One could modify this representation is several ways to turn it into a true piece of code. One simple way is to add a single "branch-and-link" opcode as a prefix to the frame:



When this closure is "jumped to", the `JSR` instruction transfers control to the closure's body and the address of the closed variables is automatically pushed on top of the stack. This address can then be used to access the closed variables of the closure. This approach was used in the Gambit compiler with good results[23].

Another interesting use of our method is to implement closures in dialects of Lisp which do not have them. This is possible because in many dialects of Lisp which do not have closures, the form `(lambda ...)` is the equivalent of an ϵ–expression and `eval` (when it is applied to the list representation of a procedure definition) is equivalent to the `compile` procedure. In other words, the S–expression `(lambda ...)` is a "true piece of code" for the evaluator (i.e. the `eval` procedure and/or the underlying interpreter). For example, in Franz–Lisp[24], the form:
`(eval '(function (lambda ...)))`
is equivalent to our:
`(compile '(epsilon ...))`

Using these equivalences, one can translate the procedure `adder`, described earlier, in Franz–Lisp as follows:

```
(defun adder (x)
  (eval
    `(function (lambda (y)
        (',(function (lambda (y) (+ x y)))
         y
         ',x)))))
```

Of course, the user should not be compelled to enter this form and the actual transformation would probably be best performed by a macro. For instance, `lambda` could be redefined as a macro that would perform the transformation described. λ–expressions would then have lexical scoping semantics just like Scheme.

8 CONCLUSION

We have shown that λ–expressions can be translated into simpler constructs, namely: ϵ–expressions and calls to a compilation procedure. This seemingly inefficient method has been refined. It yields a closure implementation efficient both in time and space. A closure is implemented as a piece of code instead of a compound object containing a code and environment pointer. Closure invocation is as simple as a jump and variable access does not imply a costly environment lookup. The transformation process has been extended to deal with assignment. It can be used in a compiler and also to simulate closures in dialects of Lisp which do not have them. It can also be used in conjunction with other closure representation methods to improve all procedure application times (be they to true closures or not). Since Scheme's basic constructs are variable reference and assignment, closure generation, procedure application, constant reference and conditional evaluation and that the performance of the first four of these constructs is influenced by our closure implementation method, it is reasonable to believe that a Scheme system using our method will perform well.

9 REFERENCES

1. Abelson, H., Sussman, G. J., Sussman, J., *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

2. Rees, J. A., Clinger, W., (editors) *The Revised^3 Report on the Algorithmic Language Scheme.* ACM SIGPLAN Notices 21, 12, pages 37–79, December 1986.

3. Steele, G. L., *Rabbit: a compiler for Scheme.* MIT Artificial Intelligence Memo 474, Cambridge, Massachusetts, May 1978.

4. Rees, J. A., Adams, N. I., *T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool.* In Conference record of the 1982 ACM Symposium on Lisp and Functional Programming, pages 114–122, Pittsburgh, Pennsylvania, August 1982.

5. Rees, J. A., Adams, N. I., Meehan, J. R., *The T Manual.* Computer Science Department, Yale University, New Haven, Connecticut, January 1984.

6. Steele, G. L., *Common Lisp: the Language.* Digital Press, 1984.

7. Steele, G. L., *Lambda: the ultimate declarative.* MIT Artificial Intelligence Memo 379, Cambridge, Massachusetts, November 1976.

8. Sussman, G. J., Steele, G. L., *Scheme: An Interpreter for extended Lambda Calculus.* MIT Artificial Intelligence Memo 349, Cambridge, Massachusetts, December 1975.

9. Wand, M., *Continuation–Based Multiprocessing.* In Conference record of the 1984 ACM Symposium on Lisp and Functional Programming, pages 19–28, Stanford, California, August 1980.

10. Atkinson, M. P., Morrison, R., *Procedures as Persistent Data Objects.* In ACM Transactions on Programming Languages and Systems, vol. 7, no. 4, pages 539–559, October 1985.

11. Feeley, M., *Deux approches à l'implantation du langage Scheme.* Document de travail no 183, Département d'informatique et de recherche opérationnelle, Université de Montréal, May 1986.

12. Feeley, M., Lapalme, G., *Using Closures for Code Generation.* Computer Languages, vol. 12, no. 1, pages 47–66, 1987.

13. Brooks, R. A., Gabriel, R. P., Steele, G. L., *An Optimizing Compiler for lexically Scoped Lisp.* In Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, pages 261–275, June 1982.

14. Griss, M. L., Hearn, A. C., *A Portable Lisp Compiler.* In Software–Practice and Experience, vol. 11, pages 541–605, 1981.

15. Aho, A. V., Ullman, J. D., *Principles of Compiler Design.* Addison–Wesley, Reading, Massachusetts, 1977.

16. Randell, B., Russell, L. J., *Algol 60 implementation.* Academic Press, New York, New York, 1964.

17. Dijkstra, E. W., *Recursive Programming.* In Programming Systems and Languages, McGraw–Hill, New York, New York, 1967.

18. McDermott, D., *An efficient Environment Allocation Scheme in an Interpreter for a Lexically–scoped Lisp.* In Conference record of the 1980 ACM Symposium on Lisp and Functional Programming, pages 154–162, Stanford, California, August 1980.

19. Church, A., *The Calculi of Lambda–Conversion.* Annals of Mathematics Studies Number 6, Princeton University Press, Princeton, New Jersey, 1941.

20. Peyton Jones, S.L., *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

21. Cardelli, L., *The Functional Abstract Machine.* Bell Labs Tech. Report TR-107, 1983.

22. Cardelli, L., *Compiling a Functional Language.* In Conference record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, August 1984.

23. Feeley, M., Miller, J. S., *A Parallel Virtual Machine for Efficient Scheme Compilation.* In Conference record of the 1990 ACM Symposium on Lisp and Functional Programming, Nice, France, June 1990.

24. Foderaro, J. K., Sklower, K. L., *The FRANZ LISP Manual.* University of California, Berkeley, California, April 1982.

Biographical sketches

Marc Feely received a Bachelor's degree (1983) and a Master's degree (1986) in Computer Science from the Université de Montréal. He is now working on a Ph.D. in Computer Science at Brandeis University. His research interests include programming language design and implementation, parallel processing and symbolic processing.

Guy Lapalme is a professor of Computer Science at the Université de Montréal. His research interests in computer languages are functional programming and object oriented logic programming. He is also interested in natural language generation and in the use of artificial intelligence techniques in the domains of operations research and tridimensional molecular structure determination.