Towards a Portable and Mobile Scheme Interpreter

Adrien Piérard
Université Paris 6
adrien.pierard@etu.upmc.fr

Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Abstract

The transfer of program data between the nodes of a distributed system is a fundamental operation. It usually requires some form of data serialization. For a functional language such as Scheme it is clearly desirable to also allow the unrestricted transfer of functions between nodes. With the goal of developing a portable implementation of the Termite system we have designed the Mobit Scheme interpreter which supports unrestricted serialization of Scheme objects, including procedures and continuations. Mobit is derived from an existing Scheme in Scheme fast interpreter. We demonstrate how macros were valuable in transforming the interpreter while preserving its structure and maintainability. Our performance evaluation shows that the run time speed of Mobit is comparable to existing Scheme interpreters.

1. Introduction

There is an increasing interest in the programming of distributed systems using distributed functional languages (e.g. Erlang [2], Termite [11], *m*Haskell [14]). Termite is a first attempt to adapt the Erlang distributed programming model to Scheme. Because Termite is implemented on top of Gambit and relies on system specific features it is difficult to port to other implementations of Scheme. A portable implementation would have the advantage that the nodes of the distributed system can be running different implementations of Scheme while still allowing unrestricted exchange of data (including procedures and continuations) and process migration.

We have begun the development of such a distributed programming system. Here we report on the design and implementation of Mobit, the interpreter at the core of this system. Mobit is derived from an existing Scheme in Scheme interpreter which achieves reasonably good execution speed by using the fast interpretation method. Because Mobit is a portable Snow [8] package it currently runs on a dozen popular Scheme systems. This approach is more attractive than reimplementing a portable Scheme system from scratch because of the lower implementation cost (i.e. reuse of the host Scheme system's runtime system and libraries) and it has a low acceptance barrier for current Scheme users (i.e. they can more easily integrate Mobit to their code base and practices).

The serialization of functions is a fundamental issue that must be addressed by the implementers of any distributed functional language. Because Mobit implements R^4RS Scheme [6], we must also address the serialization of continuations. Our main contribution is the demonstration of how this can be done while preserving the interpreter's maintainability and with local changes to the original interpreter's structure, mainly through the use of unhygienic macros.

We start by giving an overview of the pertinent features of the Termite dialect of Scheme. In Section 3 we explain the structure of the interpreter on which Mobit is based. Object serialization is discussed in Section 4. Section 5 compares Mobit's performance with other interpreters. We conclude with related and future work.

2. Termite

Termite is a Scheme adaptation of the Erlang concurrency model. A distributed system is composed of a set of *nodes* which host concurrent processes. The nodes are identified by a socket (i.e. an IP address and network port number). Processes may create new processes on the same node or a different node using the `spawn` and `remote-spawn` procedures respectively. These operations return a *pid*, which is a reference to the new process.

Each process has a single mailbox, which is the only source of data from other processes. The procedure call `(! pid obj)`, a.k.a. the *send* operation, adds *obj* to the mailbox of the process which *pid* refers to. A process may retrieve the next message in its mailbox with the procedure call `(?)`. Messages can also be selectively retrieved with the `recv` form using pattern matching.

Like Erlang, Termite disallows the mutation of variables and data structures. This avoids the semantic problems associated with data sharing. An implementation is thus free to copy objects when they are sent to another process. This is how the remote send operation is implemented. For local sends the object is shared by simply adding a reference to the object to the destination mailbox.

The only form of sharing occurs for processes. When a remote send operation encounters a *pid* in the object being sent it is only a copy of the *pid* that is added to the target mailbox. The process *pid* refers to is unchanged. Because a *pid* contains a reference to the node that hosts the process it is easy for the send operation to detect when an internode communication is occurring. Internode communication requires the serialization of the object being sent. All standard data types including procedures can be serialized. Unlike Erlang, Termite allows code and continuations to be sent between processes. This is very useful for implementing remote code update and process migration with `call/cc`. For example, a call to the procedure `goto` given below will cause the current process to continue its execution in a new process on the given node, and all of its messages to be forwarded to the new process:

```
(define (goto node)
  (call/cc (lambda (k)
             (let ((pid (remote-spawn
                         node
                         (lambda () (k #f)))))
               (let loop () (! pid (?)) (loop))))))
```

Proceedings of the 2007 Workshop on Scheme and Functional Programming
Université Laval Technical Report DIUL-RT-0701

```
(define (eval expr env)
  (cond ((or (number? expr) ... (string? expr))
         (cst expr env))
        ((symbol? expr)
         (ref expr env))
        ((pair? expr)
         (case (car expr)
           ((quote)
            (cst (cadr expr) env))
           ((if)
            (if3 (cadr expr)
                 (caddr expr)
                 (and (pair? (cdddr expr))
                      (cadddr expr))
                 env))
           ((lambda)
            (lamb (cadr expr) (caddr expr) env))
           ...
           (else
            (call (car expr) (cdr expr) env))))
        (else
         (error "malformed expression" expr))))

(define (cst val env) val)

(define (ref var env) (cdr (assq var env)))

(define (if3 test yes no env)
  (if (eval test env) (eval yes env) (eval no env)))

(define (lamb params body env)
  (lambda args
    (eval body
          (append (map cons params args) env))))

(define (call proc args env)
  (apply (eval proc env)
         (map (lambda (arg) (eval arg env))
              args)))

(define (prim proc) proc)

(define top-env
  (list (cons '+ (prim +))
        (cons 'car (prim car))
        ...))
```

Figure 1. Traditional implementation of `eval`.

```
(define (eval expr env)
  ((comp expr) env))

(define (comp expr)
  (cond ((or (number? expr) ... (string? expr))
         (cst expr))
        ((symbol? expr)
         (ref expr))
        ((pair? expr)
         (case (car expr)
           ((quote)
            (cst (cadr expr)))
           ((if)
            (if3 (comp (cadr expr))
                 (comp (caddr expr))
                 (comp (and (pair? (cdddr expr))
                            (cadddr expr)))))
           ((lambda)
            (lamb (cadr expr) (comp (caddr expr))))
           ...
           (else
            (call (comp (car expr))
                  (map comp (cdr expr))))))
        (else
         (error "malformed expression" expr))))

(define (cst val) (lambda (env) val))

(define (ref var) (lambda (env) (cdr (assq var env))))

(define (if3 test yes no)
  (lambda (env)
    (if (test env) (yes env) (no env))))

(define (lamb params body)
  (lambda (env)
    (lambda args
      (body (append (map cons params args) env)))))

(define (call proc args)
  (lambda (env)
    (apply (proc env)
           (map (lambda (arg) (arg env))
                args))))
```

Figure 2. "Fast interpretation" implementation of `eval`.

3. Interpreter structure

The Scheme metacircular evaluator is a classic textbook example. Not only does it have pedagogical value, it can serve as the basis of a Scheme interpreter when it is compiled with a Scheme compiler. Indeed many Scheme systems with an efficient compiler use this approach to implement their interpreters, e.g. Bigloo [16], Chicken [18], and Gambit [9].

The traditional formulation of the evaluator is as a two parameter `eval` procedure, for example see [1]. The first parameter is the S-expression representation of the expression to evaluate and the second parameter is the evaluation environment represented as a *variable/value* association list. As shown in Figure 1 the evaluator first determines the type of expression being evaluated and destructures it into its subparts (the procedure `eval`) and then executes the appropriate evaluation rule (the procedures `cst`, `ref`, `if3`, `lamb`, `call`, etc). Note that to simplify our exposition we have omitted most error checking and the handling of rest parameters.

To evaluate an expression at top-level, the environment bound to `top-env` is passed to `eval`. This environment contains the predefined procedures.

3.1 Fast interpretation

The evaluator's run time speed can be improved substantially by using the technique of fast interpretation [10]. Fast interpretation separates the decoding of the expression from the execution of the evaluation rules so that the decoding is done exactly once. As shown in Figure 2, this is done by Currying the `eval` procedure and moving the `(lambda (env) ...)` part to the procedures implementing the evaluation rules. Because the resulting evaluator behaves similarly to a compiler we will use compilation terminology to explain it. Like a compiler there is a compile time phase, implemented by the `comp` procedure which decodes the expression

```
(define (call proc args)
  (case (length args)
    ((0) (lambda (env) ((proc env))))
    ((1) (let ((a (car args)))
           (lambda (env)
             ((proc env) (a env)))))
    ((2) (let ((a (car args)) (b (cadr args)))
           (lambda (env)
             ((proc env) (a env) (b env)))))
    (else
     (lambda (env)
       (apply (proc env)
              (map (lambda (arg) (arg env))
                   args))))))
```

Figure 3. Code specialization of the `call` code generator.

and generates executable code. The executable code is represented with closures whose only parameter is the evaluation environment. These closures are created by the code generation procedures `cst`, `ref`, etc.

Code specialization can be applied to the code generator to further improve execution speed and reduce the size of the closures representing the code (by reducing or eliminating the free variables). Figure 3 shows how the `call` code generation procedure can be modified to avoid using list operations and `apply` at run time for the frequent kinds of calls (with two arguments or less).

Another important improvement consists in separating the compile time and run time parts of the environment. The run time environment is a chain of lexical frames (vectors) containing the values of the variables and the compile time environment maps variable names to the relative location of the value in the run time environment (i.e. position of the frame in the chain and position of the variable in the frame). The procedure `comp` thus takes two parameters, the expression `expr` and the compile time environment `cte`. Because of this separation, the `lamb` code generator no longer needs the list of parameter names.

3.2 Continuation passing style

Such a fast interpreter was developed for portably implementing the unhygienic macro expander of an early version of the Gambit compiler. It was later added to the Gambit benchmark suite to compare the performance of various R^{4}RS Scheme systems.

The preservation of the interpreter's structure was viewed as an important goal to simplify maintenance of the system and to allow the interpreter to be easily adapted to purposes beyond the scope of the Termite project. The most important structural change needed for Termite, which is also useful to increase the flexibility of the interpreter in general, is the adoption of a continuation-passing style (CPS) in the code execution part. Making the run time continuation explicit this way is useful for serializing continuations, as is shown in the next section. Consequently the procedures which represent the code take two parameters: `rtk` is the run time continuation and `rte` is the run time environment. The procedures created by `lamb` and `prim`, which represent source procedures, are also extended to take an extra `rtk` parameter. Note that the compile time phase remains in direct style. Figure 4 illustrates the changes needed for CPS on the procedures `if3`, (unspecialized) `lamb`, and `prim` procedures.

```
(define (if3 test yes no)
  (lambda (rtk rte)
    (test (lambda (r)
            (if r
                (yes rtk rte)
                (no rtk rte)))
          rte)))

(define (lamb body)
  (lambda (rtk rte)
    (rtk (lambda (rtk . args)
           (body rtk
                 (list->vector
                  (cons rte args)))))))

(define (prim proc)
  (lambda (rtk . args)
    (rtk (apply proc args))))
```

Figure 4. CPS transformation applied to the `if3`, `lamb`, and `prim` procedures.

4. Serialization

Serialization is the process of encoding an object as a sequence of symbols (characters, bytes, etc) so that an indistinguishable copy of the object can subsequently be re-created by the process of deserialization. These are fundamental operations in internode data transfer.

Scheme's `write` and `read` procedures implement a kind of serialization/deserialization with a textual representation. These procedures have the advantage of being standard but they cannot be used in the context of Termite for three reasons.

Firstly, data with shared structure and cycles are not adequately handled. The interpreter depends on sharing to minimize memory consumption, in particular the sharing of environment frames by source closures. The sharing of continuations is also a concern when the source program uses `call/cc`, for example to implement exception handling. Although Termite forbids mutation of variables and data structures it is still possible to create cycles with `letrec` forms binding lambda-expressions (or internal definitions or named-`let`s) such as:

```
(define fact
  (letrec ((f (lambda (x)
                (if (= x 0) 1 (* x (f (- x 1)))))))
    f))
```

In this case the cell for variable `f` in the `letrec`'s environment frame contains a reference to the closure created for `(lambda (x) ...)` which contains a reference back to the environment frame. Regardless, to broaden the applications of Mobit, for serialization we assume that the source language supports mutation of strings, pairs and vectors. Consequently cycles are also possible in source data.

The second problem is that some Scheme objects do not have a standard external representation, for example the value returned by `read-char` at end-of-file, characters that don't have a graphical representation, the string returned by `(string #\x #\newline)`, and the symbol returned by `(string->symbol "a B")`. If these objects are serialized with `write` with one implementation of Scheme, they may not be deserialized properly with `read` on another implementation of Scheme.

Finally, `write` and `read` are unable to serialize procedures. This is an important issue because the interpreter uses procedures at run time to represent code, continuations and source procedures.

The first problem can be addressed with a notation that expresses sharing, such as SRFI 38 (External Representation for Data With Shared Structure). A reimplementation of the `write` and `read` procedures can integrate this functionality and also solve the

second problem by adopting a standardized format for objects not having an external representation specified by the Scheme standard. The details of our serialization algorithm are given in Section 4.1. The solution to the third problem is the subject of Sections 4.2 to 4.5.

4.1 Serialization algorithm

The input to the serialization algorithm is the Scheme object to be serialized and the output is a textual encoding. Because of the possibility of cycles, the input is considered to be a *directed graph*. Each *node* of the graph is a sub-object of the input Scheme object. All nodes are reachable from the graph's *root*.

The algorithm performs a depth-first traversal of the graph starting at the root. Each node is visited exactly once. When a node is visited it is given an integer serial number indicating the order in which it was encountered in the traversal (with 0 for the root), and then the node's children are recursively serialized. The serial numbers are allocated by incrementing a serial number counter (snc). The serial number table has a dual purpose. It is used to determine when a node has already been visited. It also indicates the order in which the deserialization algorithm will encounter the node in the object's encoding. When a node contains a reference to a previously visited node i, the serialization algorithm places a *back reference* of value $snc - i$ in the output instead of the recursive serialization of node i. Using a relative back reference helps make its encoding compact.

As it scans the serialized encoding, the deserialization algorithm builds a table mapping serial numbers to reconstructed nodes. Here too serial numbers are allocated by incrementing snc which starts at 0. When a back reference k is encountered during deserialization the serial number $i = snc - k$ is looked up in the table. When the back reference to node i forms a cycle it means that the encoding of node i has not been completely scanned by the deserialization algorithm and that node i has not completely been reconstructed yet. To handle this case when the encoding of an object is encountered the deserialization algorithm immediately allocates the node with dummy content, puts it into the table of nodes, and then proceeds to mutate the fields of the node with the recursive deserialization of the children.

To improve the readability of the encoding we use an external representation similar to the one used by `write`. The following three extensions are used:

- `#`k when not followed by "`(`" – back reference (k is a non-negative integer)
- `#`n`(` *elem*$_0$ `...` `)` – vector of length n (the length must be known before the elements are scanned)
- `#p(` *id* *fv*$_1$ `...` `)` – procedure (the format is explained in Section 4.2)

For example, the object created by this Scheme code:

```
(let* ((s (string #\a)) (v (vector s #f s)))
  (vector-set! v 1 v)
  (cons s (cons v s)))
```

contains 4 nodes and is serialized into:

```
("a" #3(#2 #0 #2) . #1)
```

The performance of the serialization algorithm is highly dependent on the implementation of the serial number table. To preserve the graph's shape (sharing and cycles), the nodes are looked up using an `eq?` test. The use of an association list and the `assq` procedure is thus a correct and portable implementation. In this case serialization has a time complexity of $O(n^2)$ where n is the number of nodes. With the use of "`eq?`" hash tables the time complexity can be lowered to $O(n)$. Such hash tables are specified in SRFI 69 (Basic hash tables) and they exist in many implementations of Scheme including Gambit.

The serialization/deserialization algorithms are optimized to avoid giving a serial number to atomic objects for which `eq?`-ness is either not guaranteed by the Scheme language (such as characters and numbers) or is implicit (because there is only one instance of the object, such as booleans and the the empty list).

4.2 Serializing closures

In the CPS version of the fast interpreter, closures are used to represent three distinct run time entities:

1. **Code** (e.g. the `(lambda (rtk rte) ...)` at the top of `if3`)
2. **Continuations** (e.g. the `(lambda (r) ...)` in `if3`)
3. **Procedure objects** (e.g. the `(lambda (rtk . args) ...)` in `lamb` and `prim`)

To serialize these closures it is necessary to extract their content, that is their associated code and the value of their free variables. Deserialization does the inverse, that is it constructs an indistinguishable closure from the extracted information. Note that there is a finite number of lambda-expressions in the interpreter so we can simply use an integer label to identify the associated source code. In the external representation for procedures given above, *id* represents the integer label and *fv*$_i$ is the value of its i^{th} free variable. For primitive procedures there are no free variables and *id* is a symbol giving the name of the primitive procedure (i.e. `#p(car)` is the serialization of the primitive procedure `car`).

For example, the closure returned by the call `(cst 100)` is a closure whose only free variable `val` has the value 100 and whose integer label is 1 (let's say). It is serialized into `#p(1 100)`. Deserialization will use the integer label to find the appropriate constructor in a table, in this case the procedure `(lambda (val) (lambda (rtk rte) (rtk val)))` which unsurprisingly is the procedure `cst`, and will call it with the value 100 to reconstruct an equivalent closure.

This closure reconstruction process works fine for closures that don't contain cycles. This is the case for all closures created by `lambda-c`, the code closures. The continuation closures created by `lambda-k` can contain cycles, because of the combined presence of `call/cc` and mutation operations (`set!` and data mutators). This is impossible in Termite. Procedure object closures created by `lambda-p` can contain cycles, even in Termite, due to the presence of `letrec`.

The deserialization of closures which might contain cycles (those created by `lambda-p` and possibly those created by `lambda-k`) is more complex because, like pairs and vectors, the closures must be reconstructed in two phases. First the closure is created with dummy values for its free variables. Subsequently the free variables are mutated when the children are deserialized. This issue is discussed in the next section.

4.3 Non-opaque closures

Closures are opaque objects in Scheme and there is no standard way to extract their content. In order to experiment with various calling protocols and non-opaque representations for closures while preserving the structure of the interpreter we opted to use macros to abstract closure creation and calling. The calling protocol and closure representation can easily be changed by redefining these macros.

The macros `lambda-c`, `lambda-k`, and `lambda-p` abstract the creation of the code, continuation and procedure object closures respectively. The parameters for these macros are in order:

1. a symbolic name (used for debugging only),

```
(define (if3 test yes no)
  (lambda-c if3-c (test yes no) (rtk rte)
    (call-c test
            (lambda-k if3-k (yes no rtk rte) (r)
              (if r
                  (call-c yes rtk rte)
                  (call-c no rtk rte)))
            rte)))

(define (lamb body)
  (lambda-c lamb-c (body) (rtk rte)
    (call-k rtk
            (lambda-p lamb-p (rte body) (rtk . args)
              (call-c body
                      rtk
                      (list->vector
                       (cons rte args)))))))
```

Figure 5. Changes for easily experimenting with closure representation applied to the `if3` and `lamb` procedures.

2. the list of free variables,
3. the closure's parameters,
4. the closure's body.

Closure calling is abstracted by the macros `call-c`, `call-k`, `call-p`, and `apply-p`. The changes to the interpreter are fairly straightforward and systematic. The CPS structure is preserved and the `-c`, `-k`, and `-p` suffixes on the macros are helpful to understand the interpreter. Figure 5 shows uses of most of these macros in the procedures `if3` and `lamb`.

We considered two concrete representations for non-opaque closures: a flat closure representation implemented with vectors and closures of the host Scheme implementation.

The flat closure representation using vectors is naturally non-opaque so serialization and deserialization become trivial to implement. However it causes other problems, notably it hinders the debugging and maintenance of the interpreter. Source vectors cannot be encoded directly with host vectors. Some form of tagging is needed to distinguish them from closures (to implement the `procedure?` and `vector?` procedures). Using the host Scheme system's `write` procedure (possibly the one hidden in its REPL) to display source vectors and procedures will give an external representation that is confusing to the user. In fact when the procedure contains a cycle most Scheme systems will enter an infinite recursion. This is a case where procedure opacity is convenient.

Using closures of the host Scheme implementation to represent the interpreter's closures has some distinct advantages. It maps the source types to the same host types (so that the source vector primitives `vector?`, `vector-length`, ... are implemented with the same host vector primitives, and similarly for `procedure?`). This also facilitates the debugging of the interpreter because vectors and procedures are displayed by `write` using the normal external representation (typically procedures are displayed using an opaque external representation such as `#<procedure #10>`).

To work around the host closures' opacity, we impose a special closure calling protocol. In this new calling protocol all closures take the same number of parameters. Our original design used a two parameter protocol. Because of its relative simplicity we will describe our technique using this protocol. For increased efficiency, in the final design we use a different calling protocol which is explained in Section 4.5.

All closures take exactly two parameters. This is already the case for code closures. Continuation closures are extended to take a dummy first parameter whose value is non-`#f` when called by `call-k`. Procedure object closures are changed so that the continuation is passed in the first parameter and the list of arguments is passed in the second parameter.

This calling protocol makes it possible for the closure to implement three distinct operations:

1. **execution** – normal execution of the closure,
2. **extraction** – extraction of the closure's *id* and value of its free variables,
3. **initialization** – mutation of all the free variables.

The last two operations are selected when the first parameter is `#f`, a case that is not possible during the normal execution of the interpreter, i.e. when the closure is called through `call-c`, `call-k`, `call-p`, and `apply-p`. Initialization is selected when the first parameter is `#f` and the second is a vector whose length must be equal to the number of free variables. Each of the closure's free variables is mutated with the corresponding element of the vector. Otherwise, extraction is selected. A vector containing the closure's integer label and the values of its free variables is returned.

The lambda macros implement a transformation similar to defunctionalization [15] which moves lambda-expression bodies to top-level procedure definitions. Each lambda-expression in the interpreter becomes a call to a corresponding top-level closure constructor. Our approach differs from defunctionalization in the closure representation. We will use the `prim` and `cst` procedures to explain the transformation that the lambda macros implement. The interpreter contains the following definitions for those procedures:

```
(define (prim proc)
  (lambda-p prim-p (proc) (rtk . args)
    (call-k rtk (apply proc args))))

(define (cst val)
  (lambda-c cst-c (val) (rtk rte)
    (call-k rtk val)))
```

Let's assume that the closures created by `prim` have *id*=0 and the closures created by `cst` have *id*=1. When the `lambda-p` macro call in `prim` is encountered it is replaced by a call to a closure creation procedure, `make-closure-0`, whose definition must be generated elsewhere at top-level. Similarly the `lambda-c` macro call in `cst` is replaced by a call to `make-closure-1`. The closure creation procedures parameters are the free variables of the lambda expression, i.e. `proc` and `val` respectively. Two tables are also generated. The first table, `closure-constructor-table`, contains all the closure creation procedures. The second table, `closure-size-table`, indicates the size of the closure (number of free variables) and whether the closure can contain cycles or not (encoded in the number's sign). The expanded code along with auxiliary definitions used by the deserialization algorithm are shown in Figure 6.

Note that the initialization operation is implemented for the closures created by `prim`, but not for the closures created by `cst`. Scheme compilers typically use assignment conversion to handle mutable free variables, such as the parameter `proc` of procedure `make-closure-0`. The cell that is introduced causes a space and time overhead for reading the variable's value. It is worthwhile to avoid this overhead for closures that can't contain cycles.

The deserialization algorithm will use the single phase closure reconstruction process when the value *size*=(`vector-ref closure-size-table` *id*) is negative (cycles impossible). The list *fv* of length *-size* containing the deserialized free variables is first built. The closure is then constructed with the call (`closure-construct` *id fv*).

```
(define (prim proc)
  (make-closure-0 proc))

(define (cst val)
  (make-closure-1 val))

(define (make-closure-0 proc)
  (lambda (rtk args)
    (cond (rtk
           (call-k rtk (apply proc args)))
          ((vector? args)
           (set! proc (vector-ref args 0)))
          (else
           (vector 0 proc)))))

(define (make-closure-1 val)
  (lambda (rtk rte)
    (cond (rtk
           (call-k rtk val))
          (else
           (vector 1 val)))))

(define closure-constructor-table
  (vector make-closure-0
          make-closure-1
          ...))

(define closure-size-table
  (vector 1   ;; 1 free variable, cycles possible
          -1  ;; 1 free variable, cycles impossible
          ...))

(define (closure-construct id fv)
  (apply (vector-ref closure-constructor-table id)
         fv))

(define (closure-allocate id size)
  (closure-construct id (iota size)))

(define (closure-initialize clo fv)
  (clo #f fv))

(define (closure-extract clo)
  (clo #f #f))
```

Figure 6. Result of expansion of lambda macros.

The two phase closure reconstruction process is used when the value *size* is positive (cycles possible). It first allocates the closure *clo* with dummy values for the free variables with the call (`closure-allocate` *id size*). It then allocates a vector *fv* of length *size* and fills it with the deserialized free variables. The call (`closure-initialize` *clo fv*) finishes the reconstruction of the closure by assigning the values to the free variables.

The procedure `closure-extract` is used by the serialization algorithm to extract the closure's *id* and the value of its free variables.

4.4 Implementation of lambda macros

The lambda macros perform an unusual non-local transformation of the source code. Each call site is transformed into a call to a top-level closure constructor whose name depends on the number of previously encountered call sites. The closure constructor's definition appears later and it depends on information from the lambda macro's call site. Consequently there is a need for carrying information across macro calls. We do this by maintaining state in the macro-expansion environment. This is not something that is possible with the hygienic `syntax-rules` form. Instead we have used the unhygienic Common-Lisp style `define-macro` form which is supported by the Snow framework and many implementations of Scheme.

To maintain state we have two options: use a file which is incrementally updated or a global variable in the macro expansion environment. We chose the second option which avoids file system access portability issues and messing up the file system.

Three auxiliary macros are used: `lambda*-begin`, `lambda*`, and `lambda*-end`. Two calls to the macros `lambda*-begin`, and `lambda*-end` delimit the section of code where calls to `lambda*` are allowed (i.e. all the code generation procedures). The call (`lambda*-begin`) initializes the state which is maintained between calls to `lambda*` and the call (`lambda*-end`) produces the top-level definitions of the closure constructors, `closure-constructor-table` and `closure-size-table`. The macro `lambda*` performs the local transformation at the lambda macro call sites and stores in the state the information of the lambda macro call for later use by `lambda*-end`.

The definition of the lambda macros and the three auxiliary macros are given in Figures 7 and 8. The state is stored in three global variables in the macro expansion environment: `lambda*-indx`, `lambda*-todo`, and `lambda*-done`. These variables are defined by the macro `lambda*-begin` using a call to `eval`. This is a reasonably portable approach to add new global variables to the macro expansion environment. The other macros also reference these variables using calls to `eval` because there is no guarantee that macro bodies share the evaluation environment with `eval`.

The definition of `lambda*-end` needs to be recursive to handle nested calls to `lambda*`. For example, this happens in the procedure `lamb` which nests a call to `lambda-p` in a call to `lambda-c`. In this case, the closure constuctor generated by `lambda*-end` for the `lambda-c` will contain a call to `lambda-p`. It is only when the generated closure constructor is macro expanded that the call to `lambda-p` is processed. This must cause the generation of a new closure constructor. To handle this properly two lists of `lambda*` call sites is maintained. The variable `lambda*-todo` contains the list of call sites whose closure constructor has not yet been generated, and `lambda*-done` contains the list of call sites whose closure constructor has been generated. When `lambda*-todo` is non-empty when `lambda*-end` is expanded, the closure constructors for the call sites in `lambda*-todo` are generated followed by a call to `lambda*-end`, and the list `lambda*-todo` is transferred to `lambda*-done`. Otherwise, `lambda*-done` is used to generate the definitions for the tables `closure-constructor-table` and `closure-size-table`, and the recursion stops. This approach requires that the host Scheme system expand macros from top to bottom, but this must be the case to properly handle macros that generate macro definitions.

4.5 Improved procedure calling protocol

When Mobit uses the two parameter calling protocol, a source call to a source procedure constructs a list of the parameters and passes this list as the procedure's second parameter. This tends to generate many short lived objects which cause the program to spend considerable time garbage collecting. To reduce the generation of garbage a N parameter protocol is used. The first parameter is the continuation, the second is the number of source parameters, the third is the first source parameter (if there is one), the fourth is the second source parameter (if there is one), and so on. The last parameter is the list of the remaining source parameters. The need to construct

```
(define-macro (lambda-c info fv params . body)
  `(lambda* #f ,fv ,params
     ,@body))

(define-macro (lambda-k info fv params . body)
  `(lambda* #t ,fv (dummy ,(car params))
     ,@body))

(define-macro (lambda-p info fv params . body)
  `(lambda* #t ,fv (,(car params) ,(cdr params))
     ,@body))

(define-macro (lambda* cycles? fv params . body)
  (eval `(let* ((i
                 lambda*-indx)
                (constr
                 (string->symbol
                  (string-append
                   "make-closure-"
                   (number->string i))))
                (size
                 ,((if cycles? + -)
                   (length fv))))
           (set! lambda*-todo
                 (cons (list i
                             constr
                             size
                             ',fv
                             ',(car params)
                             ',(cadr params)
                             ',body)
                       lambda*-todo))
           (set! lambda*-indx
                 (+ lambda*-indx 1))
           (cons constr ',fv))))
```

Figure 7. Definition of lambda macros.

a list of parameters is completely avoided when there are zero to $N-3$ source parameters.

This situation is analogous to a machine code calling convention which assigns some of the registers to hold the parameter count and the $N-3$ first parameters. In our final design we use $N=6$. This works well in practice because no garbage is created for the statistically most frequent procedure calls (which take at most two parameters).

4.6 Serializing ports

Scheme ports present important challenges for serialization. Like closures, ports are opaque objects. Moreover they contain state and are bound to local resources (disks, operating system handles, etc). In most Scheme systems ports can provide access to a variety of data sources: existing data permanently stored on a file system (a file), a communication sink where yet-to-exist data will be sent in the future (a socket), a physical device that accepts data from a user (a terminal), etc. It is unclear what the user's expectations are concerning a deserialized port. In the case of a file, a copy of the file might be acceptable if the file is read-only, but not if the file is mutable or very large. Our view is that in a language like Termite, Scheme ports should be wrappers around processes. In other words, a port is a record which refers to the process bound to the local resources. When a port is serialized it is the underlying process' *pid* that is serialized. The state of the port is thus shared with other processes.

```
(define-macro (lambda*-begin)
  (eval '(begin
           (define lambda*-indx 0)
           (define lambda*-todo '())
           (define lambda*-done '())))
  #f)

(define-macro (lambda*-end)

  (define (expand x)
    (apply
     (lambda (i constr size fv p1 p2 body)
       `(define (,constr ,@fv)
          (lambda (,p1 ,p2)
            (cond (,p1
                   (let ()
                     ,@body))
                  ,@(if (< size 0)
                        '()
                        `(((vector? ,p2)
                           ,@(map (lambda (v j)
                                    `(set! ,v
                                           (vector-ref
                                            ,p2
                                            ,j)))
                                  fv
                                  (iota size)))))
                  (else
                   (vector ,i ,@fv))))))
     x))

  (let ((todo (eval 'lambda*-todo)))
    (if (pair? todo)

        (begin
          (eval '(begin
                   (set! lambda*-done
                         (append lambda*-todo
                                 lambda*-done))
                   (set! lambda*-todo '())))
          `(begin
             ,@(map expand todo)
             (lambda*-end)))

        (let ((done (reverse (eval 'lambda*-done))))
          `(begin
             (define closure-constructor-table
               (vector ,@(map cadr done)))
             (define closure-size-table
               (vector ,@(map caddr done))))))))
```

Figure 8. Definition of auxiliary macros.

4.7 Interfacing to the host Scheme system

Because Mobit uses a special procedure calling protocol the host Scheme system's procedures are incompatible with Mobit's procedures. In other words Mobit procedures can't be called directly from the host Scheme system and vice versa. To solve this problem two conversion procedures are needed: `(mobit-procedure->host` *mobit-proc*`)` and `(host-procedure->mobit` *host-proc*`)`. So although Mobit's implementation does not rely on any host specific feature, with `host->mobit-procedure` it is easy to extend Mobit with host specific features by storing converted host procedures in Mobit's global environment. Values other than procedures have the same representation in Mobit and the host so they do not require a conversion.

5. Performance

In this section we investigate the performance of the Mobit interpreter. Our goal is to show that it is in the same ballpark as the performance of interpreters available in other Scheme systems.

Being mostly written in R^{4}RS Scheme Mobit can be executed by many host Scheme systems, including interpreters and compilers. Obviously the performance of Mobit is highly dependent on the performance of the host Scheme system, and the best performance is expected from optimizing compilers. In our experiments we tried the following host Scheme compilers: Bigloo, Chicken, Gambit and Larceny [5]. These are Scheme to C compilers, except Larceny which generates native code. We encountered some problems with Bigloo and Chicken, and abandoned their use in the final experiments. Bigloo frequently produced segment violation run time errors when executing long-running programs. We suspect this is due to its non-conformant implementation of tail-calls which caused stack overflows. Chicken executed Mobit properly but very slowly. We suspect that we did not supply the compilation options and declarations for best performance.

Most of the Scheme programs we used for measuring performance are taken from the Gambit benchmark suite. Each program is iterated hundreds of times so that the execution time for the fastest case is at least 4 seconds. This avoids losing too much precision (in some cases we could only measure execution time with a resolution of 1 second). The following five Scheme programs were used:

- **Tak**: Takeuchi function, `(tak 18 12 6)`, 500 iterations.
- **Ctak**: Takeuchi function using `call/cc`, 500 iterations.
- **Earley**: Earley parser, 150 iterations.
- **Mazefun**: Maze generator, 500 iterations.
- **Paraffins**: Compute how many paraffins exist with N carbon atoms ($N = 17$), 500 iterations.

Our test machine is a 1.8 GHz AMD Opteron workstation running Linux.

5.1 Interpretation Speed

To compare Mobit's interpretation speed against other Scheme interpreters we used two host Scheme systems to execute Mobit: Gambit and Larceny (version 0.93). These instances of Mobit are respectively called Mobit/Gambit and Mobit/Larceny.

We ran the test programs with the two instances of Mobit and the builtin interpreters of the following Scheme systems: Bigloo 3.0a, Chicken 2.608, Gambit 4.0 beta 22, Gauche 0.8.9, Guile 1.6.7, and MzScheme 360. To be consistent with our goal of comparing implementations of interpreters, in the case of MzScheme the `--no-jit` option was used to disable the JIT compiler.

For the test programs the execution speed of Mobit/Larceny is consistently faster than Mobit/Gambit (between a factor of 1.5 and 2.3 times faster). Table 1 shows the execution times for all interpreters relative to Mobit/Larceny. A number less than one indicates that the interpreter is faster than Mobit/Larceny.

	Earley	Mazefun	Paraffins	Tak	Ctak
Bigloo	.9	.5	.5	.6	82.2
Chicken	1.9	1.0	1.4	.8	7.5
Gambit	.9	.6	1.0	.6	2.0
Gauche	.1	.1	.2	.1	2.8
Guile	.9	.8	1.0	.8	56.8
MzScheme	.2	.3	.4	.2	13.8
Mobit/Gambit	1.7	1.6	2.3	1.6	1.5
Mobit/Larceny	1.0	1.0	1.0	1.0	1.0

Table 1. Execution time for various interpreters relative to Mobit/Larceny

	Earley	Mazefun	Paraffins	Tak	Ctak
Speedup	2.0	2.1	1.8	2.1	1.9

Table 2. Execution speedup when support for serialization is removed from Mobit

If we exclude the Ctak benchmark we see that Gauche is consistently the fastest of the interpreters. Mobit/Larceny is 5 to 10 times slower than Gauche. On the other hand Mobit/Larceny is faster than the Chicken interpreter on all the benchmarks except Tak. The performance of Mobit on these benchmarks when compared to the other interpreters is certainly on the slow side, but still in the same ballpark as other interpreters. For instance it is within a factor of 2 of the speed of Bigloo, a factor of 1.66 of the speed of Gambit, and a factor of 1.25 of the speed of Guile.

Mobit's performance shines on Ctak which makes heavy use of first-class continuations using `call/cc`. Indeed Mobit/Gambit and Mobit/Larceny are faster than the other interpreters. Mobit/Larceny is up to 82.2 times faster than Bigloo, 56.8 times faster than Guile, 13.8 times faster than MzScheme and 7.5 times faster than Chicken (which is surprizing because Chicken's structure is specifically designed for reducing the cost of `call/cc` [3]). Here we see that the use of a CPS style in Mobit makes the implementation of `call/cc` simple and efficient, in addition to allowing the serialization of continuations. Because of this we envisage the implementation of Termite processes using Mobit's first-class continuations. There is little incentive performance-wise to extend Mobit with builtin processes. We expect Termite to be implemented for the most part through a library. This modularity will improve the maintainability of Mobit and Termite.

5.2 Serialization overhead

Each closure generated by `lambda*` supports serialization through code which is executed at run time. There are dynamic tests to distinguish the three operations (execution, extraction and initialization). The presence of assignments in the initialization operation slows down the access to the closures' free variables which would otherwise be immutable. This happens for closures generated by `lambda-k` and `lambda-p`.

To evaluate the overhead of supporting serialization we redefined the lambda macros so that they only perform the execution operation. The speedup obtained for the benchmarks is given in Table 2. The execution speed improves fairly consistently by a factor of 2.0. This means that the performance of the raw fast interpreter at the core of Mobit is reasonably good (it is slower than Gauche and MzScheme, roughly the same speed as Bigloo, and faster than Gambit and Guile).

6. Related work

Distributed programming languages face the same fundamental implementation problems as Termite: the distribution of data and code. Many systems use some form of copying through serialization to transfer data between nodes. The way in which functions and code are distributed varies considerably between systems. Here we focus on functional languages.

The GdH [13] distributed Haskell implementation uses a closed distributed system model (i.e. nodes cannot leave or join the distributed system dynamically). All the nodes must be running the same program. Although GdH does support the concept of remote evaluation, closures are second-class data because they cannot be transferred freely between nodes.

In Erlang [2] a distributed system's code is copied to every node through some manual installation process. Modules of code are stored in the file system so that the Erlang runtime system can locate them by name and dynamically load them. Closures sent between nodes A and B identify the closure's code using the name of the module, which must exist on A and B (and be consistent). The advantages of this approach are that the user has direct control over the installed code (which is good for security reasons) and that the serialization of closures is simple and compact. The drawback is that it becomes a barrier to the introduction of new code and the transfer of closures in loosely coupled open distributed systems where the nodes are managed by different organizations or do not have a file system (such as tiny embedded systems).

Both *m*Haskell [14] and Kali Scheme [4] address this issue by using a bytecode representation for code. The bytecodes associated with a closure's code are contained in the serialized encoding of the closure. Kali Scheme's serialization preserves sharing and cycles and supports continuations. To avoid the cost of sending large continuations, it serializes the topmost few frames and keeps a reference to the continuation's tail. A request to transfer the next chunk of the tail will be sent when it is needed. The garbage collection problem this introduces requires the use of a distributed garbage collector. Kali Scheme's implementation requires non-trivial extensions to the Scheme 48 virtual machine which are hard to maintain. Indeed the code is no longer operational in recent versions of Scheme 48.

An interesting approach for implementing Scheme code migration was presented by Sumii [17]. This approach, based on type-directed partial evaluation [7], does not handle all of Scheme. In particular, dynamic recursion, `eq?`, `apply`, and `call/cc` cause problems.

Our work is most closely related to the Tube mobile Scheme system [12]. The Tube's implementation is based on a CPS conversion of the source program. The serialization of closures is achieved by a function which returns an S-expression representation of the corresponding lambda-expression. Deserialization is obtained by passing the S-expression to `eval`. Portability was not a design goal for the Tube. It is unclear how hard it would be to port to other systems (it is build on top of Bigloo) and if different Scheme implementations can be used in the same distributed system. There is no published evaluation of the Tube's interpretation speed. Unlike for the Tube which requires `eval` on all nodes, Mobit does not require the whole interpreter to be available. Nodes can save code space by eliminating the `comp` and code generation procedures, keeping only the closure constructors, while still allowing closures to be serialized/deserialized. This is useful when the distributed system contains a mix of node types: workstation class nodes for development (which can support `eval`, `comp`, etc) and embedded system class nodes with very little memory.

7. Conclusion

We have described the implementation of Mobit, a portable Scheme in Scheme interpreter. It supports the serialization of closures and continuations, provides very efficient first-class continuations, and the execution speed is comparable to other Scheme interpreters. Because of these features it is particularly well suited as the basis for implementing concurrent languages and distributed systems. We intend to use it for a portable implementation of the Termite language.

Mobit's code is derived from an existing interpreter which uses the fast interpretation method. We have shown how serializable closures and continuations have been added to this interpreter with minimal impact on the interpreter's structure and maintainability. It constitutes an unusual use of macros which takes advantage of the power of unhygienic "defmacro" style macros.

Two extensions to the interpreter are planned to increase its flexibility. Firstly it will be useful to make the closures safe for space to avoid space leaks and needless communication of dead variables. Secondly an assignment conversion which creates boxes for the mutable variables will allow Termite to support mutation of local variables by simulating the boxes with lightweight processes.

Another interesting avenue is to extend Mobit's portability by implementing the closure constructors and the runtime system in other dynamically typed languages. For this we are contemplating JavaScript, Erlang, Python and Ruby. It would allow distributed Scheme programs to run on a wide variety of platforms and to easily interface with other software (web browsers, web servers, databases, etc).

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Harold Abelson and Gerald Jay Sussman and Julie Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed., MIT Press, Cambridge (MA), 1996.

[2] Joe L. Armstrong, *The Development of Erlang*, International Conference on Functional Programming, pp. 196–203, 1997.

[3] Henry Baker, *CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A.*, Nimble Computer Corporation, http://home.pipeline.com/~hbaker1/CheneyMTA.pdf, 1994.

[4] Henry Cejtin and Suresh Jagannathan and Richard Kelsey, *Higher-Order Distributed Objects*, ACM Transactions on Programming Languages and Systems, 17(5):704–739, 1995.

[5] William Clinger, *et al*, *The Larceny Project*, http://www.ccs.neu.edu/home/will/Larceny/.

[6] William Clinger and Jonathan A. Rees (editors), *The Revised^4 Report on the Algorithmic Language Scheme*, Lisp Pointers, 4(3), Association fo Computing Machinery, 1991.

[7] Olivier Danvy, *Type-Directed Partial Evaluation*, Lecture Notes in Computer Science 1706, pp. 367–411, 1998.

[8] Marc Feeley, *Scheme Now!* http://snow.iro.umontreal.ca/, 2007.

[9] Marc Feeley, *Gambit-C, A portable implementation of Scheme*, http://www.iro.umontreal.ca/~gambit/doc/gambit-c.html, 2007.

[10] Marc Feeley and Guy Lapalme, *Using closures for code generation*, Computer Languages, 12(1):47–66, 1987.

[11] Guillaume Germain and Marc Feeley and Stefan Monnier, *Concurrency Oriented Programming in Termite Scheme*, Scheme and Functional Programming 2006, pp. 125–135, 2006.

[12] David Alan Halls, *Applying Mobile Code to Distributed Systems*, PhD thesis, University of Cambridge, 1997.

[13] R.F. Pointon and Phil Trinder and Hans-Wolfgang Loidl, *The Design and Implementation of Glasgow distributed Haskell*, In Proceedings of the 12th International Workshop on Implementation of Functional Languages, pp. 101–116, 2000.

[14] André Rauber Du Bois and Phil Trinder and Hans-Wolfgang Loidl, *mHaskell: Mobile Computation in a Purely Functional Language*, Journal of Universal Computer Science, 11(7):1234-1254, 2005.

[15] John C. Reynolds, *Definitional Interpreters for Higher-Order Programming Languages*, Higher-Order and Symbolic Computation, 11(4), pp. 363–397, 1998.

[16] Manuel Serrano and Pierre Weis, *Bigloo: A Portable and Optimizing Compiler for Strict Functional Languages*, Static Analysis Symposium, pp. 366–381, 1995.

[17] Eijiro Sumii, *An implementation of transparent migration on standard Scheme*, Scheme and Functional Programming 2000, pp. 61–63, 2000.

[18] Felix Winkelman, *CHICKEN - A practical and portable Scheme system*, http://www.call-with-current-continuation.org/, 2007.