Implementation of BaLinda Scheme using Light Weight Processes

W. F. Wong

Department of Information Systems and
Computer Science,
National University of Singapore,
Lower Kent Ridge Road,
Singapore 0511.
Republic of Singapore.
email : wongwf@iscs.nus.sg

Abstract

As part of the BaLinda Lisp project, a BaLinda Scheme interpreter/compiler has been implemented. This was done by incorporating BaLinda constructs, written using the Lightweight Processes library of the SunOS, into a public domain Scheme interpreter, Scheme->C. This technical report will describe how this implementation was carried out together with the future plan of action.

1. Introduction

The BIDDLE/BaLinda Lisp project [5][9] was started in 1988. Since then, we have achieved important milestones such as implementing a sequential interpreter for BaLinda Scheme [6] on the IBM PC, a simulator for the BIDDLE architecture [7] and an interpreter for BaLinda Lisp on a network of transputers [3]. Current efforts include the implementation of a BIDDLE simulator on the EM-4 machine and a compiler for BaLinda Lisp on a network of transputers. However, what is currently lacking is an implementation on a more publicly available platform, such as the workstations. Such an implementation would help promote the availability of BaLinda Lisp. It is with this goal in mind that I have implemented the BaLinda constructs using the Lightweight Processes (LWP) library of the SunOS into a public domain Scheme interpreter/compiler.

In this technical report, I shall report on why and how the implementation was carried out, give some performance figures and conclude with my future directions.

2. Lightweight Processes

The standard Unix operating systems support multitasking via the *fork* primitive. When a fork is performed, a separate Unix process, with its own virtual address space, is set up. Forked processes can communicate via files and perform synchronization via any of three methods using semaphores, message passing or shared memory. However, such processes, often called “heavy-weight processes”, suffer from some performance problems. The major two of which is that it is rather expensive to fork a process, because a lot of housekeeping work needs to be done, and it is inconvenient to perform synchronization or sharing of variables.

To overcome this “heaviness”, the concept of *lightweight processes* (LWP) was introduced [4]. The idea is to have more than one *thread* of control running within a single address space. Each thread, while having its own stack and register sets, shares the address space with other similar threads. Synchronization can then be done within the same address space, which makes it efficient. SunOS 4 (and above) supports this concept by providing a set of library routines to create and administer threads.

For synchronization, monitors and conditional variables are supported. A conditional variable must be enclosed in a monitor. Waiting and notification can be performed on a conditional variable. If a task performs a wait on a conditional variable, the task will suspend as well as release the monitor atomically. A task will be awaken if another task performs one of the notification operations, in which case it will resume execution *within* the monitor. If there are more than one task waiting on a conditional variable, and the notification came in the form of a broadcast, the all these tasks will be awakened but will compete for the monitor. When a task successfully obtains the monitor, it will continue execution from the point after the wait. As the name implies, conditional variables provides a mean of waiting for a certain condition to become true.

In other words, LWP supports a shared memory based multiprocessing. On a uniprocessor, threads will run within a single heavyweight process scheduled by a simple round robin scheduler. Certain library calls can result in a thread being blocked. When this happens, control is passed back to the scheduler which will try to find another thread to run. All this happens within a single Unix process. This will simulate the effects of multiprocessing. With the introduction of bus-based, shared memory multiprocessor workstations, programs written in LWP for a uniprocessor can then be ported to a true multiprocessor for execution just by recompilation. All these features make LWP an attractive candidate for my implementation. Because BaLinda Lisp is based on shared memory multiprocessing, the semantic gap between BaLinda Lisp and LWP is much closer than, say, between transputers and BaLinda Lisp. Being lightweight permits a BaLinda Lisp programmer to spawn off more tasks than by using expensive fork operations. Also, the promise of a multiprocessor implementation of LWP means that the software can be easily ported to a shared memory multiprocessor.

3. Scheme->C

Scheme [2] is a lexically scoped, dynamically typed Lisp dialect originally invented in MIT in 1975. It is among the first to introduce the concept of procedures and *continuations* as first class objects. Although it was never in widespread usage, it continues to hold a place of its own throughout the proliferation of Lisp dialects. A standardization effort have resulted in a standard Scheme language. This is another of its positive characteristics.

The implementation I used is called Scheme->C [1]. It was done by Joel F. Bartlett of DEC in 1988/89. It is available in the public domain. The major aim in that project was to produce a portable Scheme system. In order to achieve this, it aimed at compiling Scheme to C codes, instead of machine codes. The bulk of the system is written in Scheme and bootstrap C codes are provided.

Internally, objects are represented in a tagged representation which gives their detail type and other information. Pointers to objects are, in turn, represented by yet another tag scheme which distinguishes pointers from fixed point numbers. The important *pair* (or cons-cell) is then formed by two tagged pointers for the `CAR` and `CDR` field respectively. In other words, a very straightforward and traditional representation of Lisp objects was adopted.

Scheme operations can then be bootstrapped. First, the primitive operations, such as `CONS` or `APPEND` are written as C procedures which manipulate directly with the Scheme objects. Higher level Scheme operations are then written by calling these primitive operations as C procedures. Still "higher" Scheme operations, which can be rewritten into other Scheme operations, are treated as macros which is taken care of in a macro expansion phase prior to doing any further execution. In this way, the entire Scheme system can be neatly bootstrapped. The cost of all these is of course efficiency. But by relying on state-of-the-art C compilers, one can hope to produce reasonably fast object codes.

4. Implementing BaLinda Lisp constructs with LWP

I shall now describe the details of how the BaLinda Lisp constructs were implemented in LWP. The following BaLinda Lisp constructs is supported :

- `(`**`exec`** *`func`* arg_1 arg_2 `... )` This is the task creation construct. It will spawn a task executing *`func`* with arg_i as the arguments. It will return immediately to the caller with the task identifier of the spawned task.
- `(`**`wait`** *`tid`*`)` This will force the caller to wait for the completion of task *`tid`*. It will return with the result of the task. In other words, if *`tid`* is the task identifier returned by `(`**`exec`** *`func`* arg_1 arg_2 `... )`, the `(`**`wait`** *`tid`*`)` returns the result of `(`*`func`* arg_1 arg_2 `... )`.
- `(`**`out`** $field_1$ $field_2$ `... )` This is will put the tuple `(`$field_1$ $field_2$ `... )` into the tuple space.
- `(`**`in`** $field_1$ $field_2$ `...` $field_i$ `?` var_1 var_2 `...` var_j`)` This will search for a tuple, of length $i + j$ and matching `(`$field_1$ $field_2$ `...` $field_i$`)` in tuple space. Should sure a tuple exist, then `(`var_1 var_2 `...` var_j`)` is bound to the remaining j fields of the matched tuple. Should such a tuple not exist in the tuple space, the task performing the **`in`** will block until a matching tuple is **`out`**ed by some other task. Upon completion, the matching tuple is removed from the tuple space.
- `(`**`rd`** $field_1$ $field_2$ `...` $field_i$ `?` var_1 var_2 `...` var_j`)` This behaves the same as an in except that the matching tuple is not removed from the tuple space.

Notice that I did not implement the BaLinda Lisp **`future`** construct. This is because it will require substantial redoing of the Scheme->C software. Furthermore, because **`exec`** and **`wait`** are more general, it is possible to devise a source code level macro expander to rewrite **`future`**s in terms of **`exec`**s and **`wait`**s. Also, speculative processing is not implemented as it is meaningless to do so on a uniprocessor.

There are two important data structures in the implementation. One is to keep information about tasks while the other is for the tuple space. The following data structure is used to capture task information :

```
struct TASK {
     thread_t tid;
     mon_t    monitor;
     cv_t     death;
     int      dead, cond;
     TSCP     result;
};
```

`tid` is the internal task identifier given by the system upon creating a LWP, `monitor` is a monitor for updating the structure atomically, `death` is a conditional variable to implement **`wait`**, `dead` indicate if the task is dead, `cond` indicate the C exit condition of the task while `result` is a TSCP (Tagged Scheme->C Pointer) which holds the result to be returned by the task. To create a task, space for the task structure is first allocated. Then using the LWP creation procedure, a LWP is created. Next, the monitor and the conditional variable is created. The actual implementation of **`exec`** and **`wait`** will be described later.

The tuple space is simply a two dimensional array of tuples. It is indexed by the length of the tuple in one dimension and a hash key in the other.

```
struct LOCKS {
     mon_t monitor;
     cv_t  retry;
     TSCP  space;
} space_lock[NO_OF_TUPLES][NO_OF_HASH];
```

monitor is a monitor guarding each subspace, retry is a conditional variable to implement the tuple operations and space is a TSCP pointing to the actual subspace which is a list of tuples. Because the matching criterion dictates that each of the tuple fields have to be examined in order to come up with a match, it is very important to narrow down the search as early as possible. Therefore, the aim of using two keys - the length and an additional hash key is to achieve this. However, as in all hashing problems, it is not possible to avoid the worst cases.

I shall now describe how **`exec`** is implemented. Because any Lisp system will first completely evaluate all the arguments before entering a procedure, it is necessary to intercept **`exec`**s together with its arguments early in the evaluation phase of the interpreter/compiler. It turned out that there were many subtlety to the proper handling of **`exec`** which will be described in details in the following sections as its actual implementation differs significantly in the case of the interpreter and the compiler.

The tuple operations is done as follows. In an **`out`**, the hash key is worked out based on the first field of the tuple. The monitor for the respective subspace (indexed by the length and the hash key of the tuple) is entered. This is necessary to ensure atomic updating. The tuple is simply appended to the tuple space which is a list of tuples. The append is done to the head of the list because it will facilitate the search performed by the

in and **rd** operations. A "broadcast" on the conditional variable is then performed. This will awaken any tasks waiting on the subspace. The monitor is then exited.

To perform an **in** or **rd**, the hash key is worked out in exactly the same way as in the case of `out`. Currently, an **in** or **rd** tuple without any matching field is not supported. The monitor for the subspace is then entered. All the tuples in the subspace are examined one at a time for a match. Should no match be found, the task will wait on the conditional variable of the subspace. This will force the task to sleep until an **out** does a broadcast on the conditional variable of the subspace. Upon awaking from a conditional variable, it will reattempt to search the tuple space again. Because conditional variables guarantee atomicity, there is no danger of more than one task in possession of a subspace.

A **wait** is simply achieved by first entering the monitor of the task structure and then waiting on the death conditional variable. Upon completion, a task will enter the monitor of its task structure, write the condition code as well as the result, which is in the form of a TSCP, and then notifies any one waiting on its death conditional variable.

wait and **out** can be written as straightforward Scheme code with the respective calls to C routines to perform the task oriented functions. However, **exec**, **in** and **rd** require special attention. This will be described in the following section.

5. Putting it together in the Scheme->C interpreter

The Scheme->C system is written in three parts. First is the library of Scheme routines. Next comes the interpreter which is really a front end to read in Scheme programs and then call the respective library routines. The last part is the compiler which will translate Scheme programs into C programs that will make the necessary library calls. Therefore, the interpreter is written quite differently from the compiler. One major difference is in the way local variables are handled. In the interpreter, an environment list (conveniently maintained by the Scheme list operations) containing the local bindings is maintained. Any binding can be looked up from this environment list. In the compiler, however, since Scheme procedures translate to C procedures, local variables are kept on the stack by mapping them to C variables (containing TSCPs). There is no concept of an environment list. This will significantly speed up the application. But for my purpose, I have to deal with two separate pieces of software.

In the interpreter, after reading in a Scheme expression, macro expansion is performed. Thereafter, in the `EVAL` procedure, the environment is built by binding local variables to the respective arguments. Then each subexpression is interpreted by `EXEC` procedure. It is in `EXEC` that codes are inserted to intercept the **exec**, **in**, **out** and **rd** operation. The skeleton of the code involve is as follows :

```
;; Process fields in a IN/RD tuple up till the '?'

(define (IN/RD-TUP-EXEC tup ans env)
    (if (null? tup)
        (reverse ans)
        (let ((x (car tup)))
          (if (eq? x '?)
              (append (reverse ans) tup)
              (IN/RD-TUP-EXEC (cdr tup) (cons (exec-any x env) ans) env)))))

;;; Interpret items represented by pairs.

(define (EXEC exp env)
        . . .
          ((exec)
             (let ((new-env env)
                  (comp-form (compile (expand (cdr exp)) env)))
               ((lap (c e) (MY__EXEC c e)) comp-form new-env)))
          ((out)
             (exec-out
                   (map (lambda (x) (exec-any x env)) (cdr exp))))
          ((in)
           (let ((res (IN/RD-TUP-EXEC (cdr exp) '() env)))
             (set! res (exec-in res))
             (if (pair? res)
                 (do ((new-bind res (cdr res)))
                     ((null? new-bind) res)
                   (if (assq (caar new-bind) env)
                       (set-cdr! (assq (caar new-bind) env) (cadar new-bind))
                       (set-top-level-value! (caar new-bind)
                                             (cadar new-bind)))))))
          ((rd)
           (let ((res (IN/RD-TUP-EXEC (cdr exp) '() env)))
             (set! res (exec-rd res))
             (if (pair? res)
                 (do ((new-bind res (cdr res)))
                     ((null? new-bind) res)
                   (if (assq (caar new-bind) env)
                       (set-cdr! (assq (caar new-bind) env) (cadar new-bind))
                       (set-top-level-value! (caar new-bind)
                                             (cadar new-bind)))))))
             . . .
```

For **`exec`**, evaluation of its arguments is prevented. Instead, a macro expansion is done on each of the argument. The resulting expression and the environment are passed to the C procedure as a TSCP. The C procedure will start up a LWP with a C stud procedure as the entry of the LWP. When the task starts executing, it will execute a stud procedure. The stud will call the Scheme->C procedure to perform the evaluation of the Scheme expression within the given environment. Upon completion, the Scheme->C procedure will return the result as a TSCP. The stud then updates the task structure to reflect the completion of the task. This includes setting up the conditional variable as well as leaving the result in the task structure for a subsequent **`wait`** to pick up.

`out` is straightforward. Each argument in the tuple is first completely evaluated and then the Scheme procedure `EXEC-OUT` is called.

in and **rd** are perhaps the most complicated. First, the arguments in the tuple up to the '?' has to be completely evaluated in the current environment. This is done by the Scheme procedure `IN/RD-TUP-EXEC`. The resulting tuple is then passed to the Scheme procedure `EXEC-IN` (or `EXEC-RD` in the case of **rd**). Upon successful completion, the resulting bindings to be done will be returned in a list. This is traversed and the binding is done to the local environment (by means of destructively modifying the environment list) or the global environment (by means of the Scheme procedure `SET-TOP-LEVEL-VALUE!`).

As mentioned before **wait** is implemented in terms of a Scheme procedure which will call a C routine.

6. Putting it together in the Scheme->C compiler

The Scheme->C compiler works quite differently, and far less straightforward, than the interpreter. In order to incorporate the BaLinda Lisp constructs without disturbing the main compiler, I have chosen to intercept the input as soon as it is read in by the compiler and perform a customized macro expansions. In other words, all the BaLinda Lisp constructs were written as macros. The codes involved are is given in Appendix C. To better understand what is involved, it is better to examine the rewritings involved. Let us start from the simpler ones.

An **out** is rewritten as follows :

(**out** a_1 a_2 . . . a_n) $\Rightarrow$

(EXEC-OUT (LIST *expand*(a_1) . . . *expand*(a_n))

where *expand*(x) performs (recursively) macro expansion on the arguments. In the process of calling EXEC-OUT, the arguments would be completely evaluated. Therefore in most cases, *expand*(a_i) = a_i. A **in**/**rd** is expanded as follows.

(**in** a_1 a_2 . a_i ? v_1. . v_j) $\Rightarrow$

```
(LET ((res (LIST a1 ... ai '? 'v1 ... 'vj)))
  (SET! res (EXEC-IN res))
  (IF (AND (LIST? res) (NOT (NULL? res)))
      (LET ((resvec (LIST->VECTOR (MAP CADR res))))
        (SET! v1 (VECTOR-REF resvec 1))
                . . .
        (SET! vj (VECTOR-REF resvec j))))))
```

In producing res, the arguments a_1 to a_i will be forced to evaluate. But it is necessary to prevent the variables, v_1 to v_j, from being evaluated. This is done by putting them in quotes. Depending on whether it is an **in** or a **rd**, the procedure EXEC-IN or EXEC-OUT is called. This will perform the tuple operations. What remains is binding the variables to the results of the matched tuple. This is done by converting the results into a vector; and then, depending on how many variables there are, generate the necessary SET! expressions.

The **exec** operation is the hardest to handle. The reason being that unlike the interpreter, the compiler does not employ an environment list. As such, once a task is spawned, arguments, which may be dynamically bound variables in the parent is not available. To overcome this, all the arguments in the except the function itself must be forced to evaluate. An **exec** is rewritten as such :

```
(exec f a1 a2 . ai)    ==>
 (LET ((mynew-exp `(f ,a1 ,a2 . . . ,ai))
    ((LAP (c e) (MY__EXEC c e))
       ((LAMBDA (lis)
          (IF (PAIR? lis)
              (LET ((h (CAR lis)))
                (DO ((res (LIST h))
                     (rlis (CDR lis) (CDR rlis)))
                    ((NULL? rlis) (REVERSE res))
                  (LET ((p (CAR rlis)))
                    (IF (OR (NULL? p) (PAIR? p))
                        (SET! res (CONS (LIST 'QUOTE
                                         p) res))
                        (SET! res (CONS p res))))))
              lis))
          mynew-exp)
       '())))
```

First, by means of the *quasiquote* (*backquote*) and *unquote* mechanism, the arguments of f are forced to evaluate. Because this is done in the parent, the locations of the any variables that may be in the arguments can then be fully identified. But this creates a new problem. Some of the arguments may evaluate to lists. When the child process starts executing f, it will attempt to first evaluate the arguments it receive for f. Any list will be treated as a function call in this process. We have to prevent this argument evaluation because we have already done so in the parent. To do this, an unnamed function (the other choice would be to make this function global which may have some undesirable consequences) is invoked which will add a QUOTE to any list in the arguments. After this, the C routine can then be invoked to start up the child with f and its arguments. I used the same C routine in both the interpreter and the compiler and so it is necessary to pass in an empty list for the environment list argument to the routine.

7. Performance

We have tested the interpreter and the compilers with some small test programs. In the following, we shall report the more interesting result of compiled quicksort. Two versions of quicksort [8] were tested. One of the version uses a shared array and involves more tasks and a lot of tuple operations while another involves using lists which is more expensive in terms of internal memory but is faster. Table 1 gives the results.

n	Compiled Array Version		Compiled List Version	
	Time (seconds)	Number of Tasks Spawned	Time (seconds)	Number of Tasks Spawned
100	0.53	83	0.07	30
400	2.34	339	0.32	130
800	5.06	711	0.68	266
1000	6.31	874	0.87	336
2000	crashed	N. A.	1.65	594
3000	crashed	N. A.	1.96	594
4000	crashed	N. A.	2.56	594
5000	crashed	N. A.	2.86	594
6000	crashed	N. A.	3.89	594
7000	crashed	N. A.	crashed	N. A.
8000	crashed	N. A.	5.50	594
9000	crashed	N. A.	crashed	N. A.

Table 1. Performance on Two Versions of Parallel Quicksort

Neither of the two programs were able to execute for n greater than 8,000. The problem is primarily one of (virtual) memory insufficiency. As expected, the array version, because of its spawning a large number of tasks crashed for a relatively small n. The list version worked more better, achieving good speed and a large n. It still crashed because of insufficient internal stack space to handle the larger partial results. In the case of the crash at n = 7,000, I attribute it to the pattern of the random number. It so happens that one partition contains way too many items for the stack to handle and as a result the particular child task and the entire process crashed.

Still, I think the results are encouraging. The compiler is able to handle as many as 800 tasks with large lists generated as partial results with a fairly respectable speed.

8. Conclusion and Future Plans

This technical report described the implementation of BaLinda Scheme using the SunOS Lightweight Processes library and the Scheme->C Scheme interpreter/compiler system. The result is a interpreted and compiled version of BaLinda Scheme that is fairly portable across a number of platforms. I also reported on some preliminary performance results using the quicksort program.

As mentioned earlier, it is my intention to move the implementation to a true bus based multiprocessor. From this, I intend to build and study the space call mechanism proposed in [10]. Together as a group, we are currently in the process of building the necessary tools for a larger and realistic implementation of the BaLinda dialects.

References

[1] J.F. Bartlett, *Scheme->C a Portable Scheme-to-C Compiler, WRL Research Report 89/1*. Jan 1989.

[2] W. Clinger and J. Rees (ed) *Revised^4 Report on the Algorithmic Language Scheme*. Nov 1991.

[3] M.D. Feng, Y.Q. Gao and C.K. Yuen, "Implementing Linda Tuplespace on a Distributed System", *DISCS Technical Report TRG5/93. Depart. of Information Systems and Computer Science, National University of Singapore*. May 1993.

[4] Sun Microsystems, *Programming Utilities and Libraries*. Mar 1990.

[5] W.F. Wong and C.K. Yuen, "BIDDLE : A BIdirectional Data Driven Lisp Engine", *Proc. of 1989 IEEE Conf. on Tools for A.I.* pp. 207-214. 1989.

[6] W.F. Wong, "A Parallel Lisp System", *MSc. Thesis*, *Depart. of Information Systems and Computer Science, National University of Singapore*. 1990.

[7] J.J. Yee and C.K. Yuen, "BIDDLE : A Dataflow Architecture for Lisp", *Proc. of 25th Hawaii International Conference on System Science*, pp. 611-618. Jan 1992.

[8] C.K. Yuen, "Quicksort in BaLinda Lisp : A Study in Language Extensions to Meet Algorithm Requirements", *DISCS Technical Report TRB2/91. Depart. of Information Systems and Computer Science, National University of Singapore*. Feb 1991.

[9] C.K. Yuen, M.D. Feng, W.F. Wong and J.J. Yee, *Parallel Lisp Systems : a Study of Languages and Architectures*. Chapman and Hall 1993.

[10] C.K. Yuen and W.F. Wong, “Space Calls : An Object Oriented Extension to BaLinda Lisp”, *DISCS Technical Report TRH4/93. Depart. of Information Systems and Computer Science, National University of Singapore*. Apr 1993.

Appendix A

```
;;
;; The following Scheme code is in the file scrtuser.sc which allows the user
;; add new runtime routines.
;;
;; The following makes the routines EXEC-IN, EXEC-RD, EXEC-OUT and WAIT
;; available globally.
;;
(module scrtuser
    (top-level
        EXEC-IN EXEC-RD EXEC-OUT WAIT))

;;
;; Performs a wait
;; Simply call the C procedure to wait on the death conditional variable of
;; the task tid.
;;
(define (WAIT tid)
    ((lap (x) (MY__WAIT x)) tid))

;;
;; Returns the variables to be bounded
;; When a matching tuple is found, the variables and their respective values in
;; the matching tuple is put in a list to be returned by MATCH.
;;
(define (MATCH-AUX pat tuple alis)
    (if (null? pat)
        alis
        (let ((h_pat (car pat))
              (h_tup (car tuple)))
          (MATCH-AUX (cdr pat) (cdr tuple)
                 (append alis (list (list h_pat h_tup)))))))

;;
;; Performs matching
;; Match fields up to the '?'
;;
(define (MATCH pat tuple)
    (if (or (null? pat) (null? tuple))
        (if (and (null? pat) (null? tuple))
            #t
            #f)
        (let ((h_pat (car pat))
              (h_tup (car tuple)))
          (if (eq? h_pat '?)
              (let ((t_pat (cdr pat))
                    (t_tup tuple))
                (if (= (length t_pat) (length t_tup))
                    (MATCH-AUX t_pat t_tup '())
                    #f))
              (if (eq? h_pat h_tup)
                  (MATCH (cdr pat) (cdr tuple))
                  #f)))))

;;
;; Performs hashing
;;
```

```
(define (HASH obj)
    (let ((hcode (cond ((eq? obj '?) 0)
                       ((integer? obj) obj)
                       ((real? obj) (floor obj))
                       ((string? obj) (string-length obj))
                       ((char? obj) (char->integer obj))
                       ((symbol? obj) (string-length (symbol->string obj)))
                       (#t #f))))
      (if (eq? hcode #f)
          (error 'HASH "Invalid Tuple !")
          (modulo hcode 100))))

;;
;; Performs the OUT operation
;; Do a hash, call the C function to get the lock and the subspace
;; then append the tuple to the subspace.
;;
(define (EXEC-OUT tuple)
    (let ((s_no (length tuple))
          (h_no (hash (car tuple))))
      ((lap (x y) (MY__GET_LOCK x y)) s_no h_no)
      (let ((space ((lap (x y) (MY__GET_SPACE x y)) s_no h_no)))
         ((lap (x y z) (MY__UPDATE_SPACE x y z))
                           s_no h_no (append (list tuple) space)))
      ((lap (x y) (MY__TUP_OUT x y)) s_no h_no)  ;; Do the notification
      ((lap (x y) (MY__REL_LOCK x y)) s_no h_no)))

;;
;; Performs a tuple match and IN in the tuple subspace
;; For each tuple in the tuple subspace, perform a match.
;; If matching tuple is found, remove it from the tuple space.
;;
(define (EXEC-IN-TRY-AUX tuple sp space s_no h_no)
    (if (null? space)
        #f
        (let* ((cur (car space))
               (res (MATCH tuple cur)))
          (if res
              (let ((new_sp (append sp (cdr space))))
                ((lap (x y z) (MY__UPDATE_SPACE x y z)) s_no h_no new_sp)
                res)
              (EXEC-IN-TRY-AUX tuple (append sp (list cur))
                                   (cdr space) s_no h_no)))))

;;
;; Get subspace during a retry
;; For every retry, the subspace must be obtained again.
;;
(define (EXEC-IN-TRY tuple s_no h_no)
    (let ((space ((lap (x y) (MY__GET_SPACE x y)) s_no h_no)))
      (EXEC-IN-TRY-AUX tuple '() space s_no h_no)))

;;
;; Perform IN operation
;; Do a hash, obtain the locks and the subspace and try for a match.
;; If match fails, suspend on the conditional variable.
;;
(define (EXEC-IN tuple)
    (let ((s_no (if (memq '? tuple)
                    (- (length tuple) 1)
                    (length tuple)))
          (h_no (hash (car tuple))))
      ((lap (x y) (MY__GET_LOCK x y)) s_no h_no)
      (do ((ans (EXEC-IN-TRY tuple s_no h_no) (EXEC-IN-TRY tuple s_no h_no)))
```

```
          (ans (begin
                  ((lap (x y) (MY__REL_LOCK x y)) s_no h_no)
                  ans))
        ((lap (x y) (MY__SPIN x y)) s_no h_no))))

;;
;; Performs a tuple match and IN in the tuple subspace
;; For each tuple in the tuple subspace, perform a match.
;; Unlike EXEC-IN-TRY-AUX, a successful match do not result in the
;; removal of the matching tuple.
;;
(define (EXEC-RD-TRY-AUX tuple space)
    (if (null? space)
        #f
        (let* ((cur (car space))
               (res (MATCH tuple cur)))
          (if res
              res
              (EXEC-RD-TRY-AUX tuple (cdr space))))))

;;
;; Get subspace during a retry
;; For every retry, the subspace must be obtained again.
;;

(define (EXEC-RD-TRY tuple s_no h_no)
    (let ((space ((lap (x y) (MY__GET_SPACE x y)) s_no h_no)))
      (EXEC-RD-TRY-AUX tuple space)))

;;
;; Perform a RD operation
;; Similiar to EXEC-IN
;;
(define (EXEC-RD tuple)
    (let ((s_no (if (memq '? tuple)
                    (- (length tuple) 1)
                    (length tuple)))
          (h_no (hash (car tuple))))
      ((lap (x y) (MY__GET_LOCK x y)) s_no h_no)
      (do ((ans (EXEC-RD-TRY tuple s_no h_no)))
          (ans (begin
                  ((lap (x y) (MY__REL_LOCK x y)) s_no h_no)
                  ans))
        ((lap (x y) (MY__SPIN x y)) s_no h_no))))
```

Appendix B

```
/*****************************************************

     C routines
     ==========

   These are the C routines which supports the
   operations needed by the Scheme procedures.
   The names by which they are known in Scheme
   their C names are different  because  there
   is an additional mapping involved.

*****************************************************/

#include <stdio.h>
#include <objects.h>
```

```
extern TSCP sc_eval(XAL2(TSCP, TSCP));
extern TSCP scrt6_display(XAL2(TSCP, TSCP));

#define STACK_SIZE 1000000

int alloc_stack_size = 0;

struct LOCKS {
    mon_t  monitor;
    cv_t   retry;
    TSCP   space;
} space_lock[NO_OF_TS][NO_OF_HASH];

int c_init_space()
{
    int i, j;

    if (pod_setmaxpri(MAXPRIO))
       lwp_perror("SETPRI ERROR - ");
    for (i=0; i<NO_OF_TS; i++) {
        for (j=0; j<NO_OF_HASH; j++) {
            mon_create(&(space_lock[i][j].monitor));
            cv_create(&(space_lock[i][j].retry), space_lock[i][j].monitor);
            space_lock[i][j].space = EMPTYLIST;
        }
    }
}

struct SCARG {
    TSCP exp, env;
    struct TASK *task;
};

int dummy_stud (ptr)
char *ptr;
{
    struct SCARG *iptr;
    TSCP res;

    iptr = (struct SCARG *)ptr;
    res = sceval_myexec(iptr->exp, iptr->env);
    if (mon_enter((iptr->task)->monitor))
       lwp_perror("POST DEATH - ");
    (iptr->task)->dead   = 1;
    (iptr->task)->result = res;
    if (cv_broadcast((iptr->task)->death))
       lwp_perror("POST DEATH - ");
    if (mon_exit((iptr->task)->monitor))
       lwp_perror("POST DEATH - ");
}
int TaskCount = 0;

int c_myexec(exp, env)
TSCP exp, env;
{
    struct TASK *tptr;
    int i;
    struct SCARG *args;

    tptr = (struct TASK *)(malloc(sizeof(struct TASK)));
    args = (struct SCARG *)(malloc(sizeof(struct SCARG)));
    args->exp = exp;
    args->env = env;
```

```
    args->task = tptr;
    tptr->dead = tptr->cond = 0;

    if (alloc_stack_size == 0)
       alloc_stack_size = STACK_SIZE;

    lwp_setstkcache(alloc_stack_size, 2);

    mon_create(&(tptr->monitor));
    cv_create(&(tptr->death), tptr->monitor);

    if (lwp_create(&(tptr->tid), dummy_stud, MAXPRIO, 0,
                   lwp_newstk(), 1, args))
       lwp_perror("CREATE ERROR - ");

    TaskCount++;

    i = (int)(tptr);
    return(i);
}

TSCPP c_mywait(iptr)
int iptr;
{
    struct TASK *tptr;

    tptr = (struct TASK *)(iptr);
    if (mon_enter(tptr->monitor))
       lwp_perror("WAIT - ");
    if (tptr->dead == 0) {
       if (cv_wait(tptr->death))
          lwp_perror("WAIT - ");
    }
    if (mon_exit(tptr->monitor))
       lwp_perror("WAIT - ");

    return(tptr->result);
}

int c_acquire_space(s, h)
int s, h;
{
    if (mon_enter(space_lock[s][h].monitor))
       lwp_perror("GET LOCK - ");
    return(0);
}

int c_release_space(s, h)
int s, h;
{
    if (mon_exit(space_lock[s][h].monitor))
       lwp_perror("REL - ");
    if (lwp_yield(SELF))
       lwp_perror("YIELD ERROR - ");
    return(0);
}

TSCPP c_get_space(s, h)
int s, h;
{
    TSCP x;
    x = space_lock[s][h].space;
    printf("\nGot space !\n");*/
    return(x);
```

```
}

int c_update_space(s, h, t)
int s, h;
TSCP t;
{
    space_lock[s][h].space = t;
    return(0);
}

int c_tuple_outed(s, h)
int s, h;
{
    if (cv_broadcast(space_lock[s][h].retry))
       lwp_perror("OUT - ");
    return(0);
}

int c_spin(s, h)
int s, h;
{
    cv_wait(space_lock[s][h].retry);
    return(0);
}

int c_total_task()
{
   printf("Total Number of LWP spawned = %d\n", TaskCount);
   return(TaskCount);
}

int c_set_stksize(s)
int s;
{
   if (s > 0)
      alloc_stack_size = s;

   return(s);
}
```

Appendix C

```
;;
;; Process tuples in IN and RD.
;; Fields before the '?' force to evaluate while those after (and including) the
;; '?' are quoted to prevent their evaluation.
;;
(define (IN/RD-TUP-PROC tup res flag)
   (if (null? tup)
       (reverse res)
       (if flag
           (let ((x (car tup)))
              (let ((n-res (cons `(quote ,x) res)))
                 (IN/RD-TUP-PROC (cdr tup) n-res flag)))
           (let ((x (car tup)))
             (if (eq? x '?)
                 (let ((n-res (cons '(quote ?) res)))
                   (IN/RD-TUP-PROC (cdr tup) n-res #t))
                 (IN/RD-TUP-PROC (cdr tup) (cons x res) #f))))))
;;
;; Insert UNQUOTE to force evaluation of arguments.
```

```
;;
(define (MYADDEVAL exp nexp)
  (if (null? exp)
      (reverse nexp)
      (MYADDEVAL (cdr exp) (cons (list 'unquote (car exp)) nexp))))

;;
;; Splice in the SET! for IN and RD.
;;
(define (IN/RD-SET-AUX tup i res)
  (if (null? tup)
      (reverse res)
      (let ((x (car tup)))
           (if (pair? x)
               (expand-error 'IN/RD tup)
               (let ((nexp `(set! ,x (vector-ref resvec ,i))))
                 (IN/RD-SET-AUX (cdr tup) (+ i 1)
                            (cons nexp res)))))))

;;
;; Splice in the code to do the variable binding in a IN or RD.
;;
(define (IN/RD-SET-PROC tup)
  (let ((rest (do ((nt tup (cdr nt)))
                  ((or (null? nt) (eq? (car nt) '?)) nt))))
    (if (null? rest)
        '(#t)
        (IN/RD-SET-AUX (cdr rest) 0 '()))))

;;
;; The customized macro expander
;; Looks out for the BaLinda Lisp operations and perform the necessary
;; expansion.
;;
(define (MYEXPAND exp)
  (if (pair? exp)
      (if (list? (car exp))
          (cons (MYEXPAND (car exp)) (MYEXPAND (cdr exp)))
          (case (car exp)
            ((exec)
              (let ((eexp (list 'quasiquote (cons (cadr exp)
                                                   (MYADDEVAL (cddr exp) '())))))
                `(let ((mynew-exp ,eexp))
                   ((lap (c e) (MY__EXEC c e))
                      ((lambda (lis)
                         (if (pair? lis)
                             (let ((h (car lis)))
                               (do ((res (list h))
                                    (rlis (cdr lis) (cdr rlis)))
                                   ((null? rlis) (reverse res))
                                 (let ((p (car rlis)))
                                   (if (or (null? p) (pair? p))
                                       (set! res (cons (list 'quote p) res))
                                       (set! res (cons p res))))))
                             lis))
                       mynew-exp)
                       '()))))
            ((in)
              `(let ((res (list ,@(IN/RD-TUP-PROC (cdr exp) '() #f))))
                 (set! res (exec-in res))
                 (if (and (list? res) (not (null? res)))
                     (let ((resvec (list->vector (map cadr res))))
                       ,@(IN/RD-SET-PROC (cdr exp))))))
            ((rd)
```

```
                `(let ((res (list ,@(IN/RD-TUP-PROC (cdr exp) '() #f))))
                   (set! res (exec-rd res))
                   (if (and (list? res) (not (null? res)))
                       (let ((resvec (list->vector (map cadr res))))
                         ,@(IN/RD-SET-PROC (cdr exp))))))
             ((out)
                `(exec-out (list ,@(MYEXPAND (cdr exp)))))
             (else (cons (car exp) (MYEXPAND (cdr exp))))))
      exp))

;;
;; The Scheme procedure to read in the text input, store it as lists,
;; and perform the initial macro expansion.
;;
(define (READ-TEXT)
    (let ((form '()))
        (if sc-splice
            (begin (set! form (car sc-splice))
                  (set! sc-splice (cdr sc-splice)))
            (begin (set! form (MYEXPAND (read-from-sc-input))) ;; Added here !
                     (set! form (sc-expand form))
                  (if (log? 'macro)
              . . .
```