Pscheme: Extending Continuations to Express Control and Synchronization in a Parallel LISP

Chi Yao
Benjamin Goldberg

Department of Computer Science[1]
Courant Institute of Mathematical Sciences
New York University

Abstract

In this paper, we describe Pscheme, a parallel dialect of Scheme. The primary construct for specifying parallelism, synchronization, and communication is a natural extension of first-class continuations which we call a *port*. We describe the behavior of ports, along with the other parallel constructs of Pscheme. Because the user has precise control over the parallel computation, the Pscheme constructs can be used to build higher-level parallel programming abstractions, such as futures, semaphores, and Ada-style rendezvous. We provide the Pscheme code for these abstractions and discuss the current implementation of Pscheme on a shared-memory multiprocessor.

1. Introduction

In this paper, we describe Pscheme, a parallel dialect of Scheme [Cl91]. The primary construct for specifying parallelism, synchronization, and communication is a natural extension of first-class continuations which we call a port. The benefit of using Pscheme is that the user has complete control over the order of parallel evaluation, while at the same time benefiting from the use of a very high level language like Scheme. Other parallel variants of LISP and Scheme have been proposed, based on first-class continuations, but we feel that our ports provide the most natural parallel extension of sequential continuations of the methods suggested.

Like first-class continuations, ports are very powerful constructs that can lead to complicated programs that are difficult to read. We envision ports (and continuations) to be primarily the tool of advanced "systems programmers" who provide libraries of higher level parallel constructs (futures, barriers, engines, etc.) for use by the general programming community. We show how ports are sufficiently powerful to easily implement such high level parallel constructs.

We have built a compiler for Pscheme. The compiler generates C which runs on a shared memory multiprocessor, the NYU Ultracomputer. The implementation has only recently become sufficiently robust to begin experimentation and timings. In the full version of the paper, the experimental results will be described.

1.1. Motivation

The notion of a port as an extension of a continuation arose from research into reflective parallel languages. In particular, we were trying to find a programming abstraction that captures the notion of an arc in a dataflow graph. For example, the illustration in Figure 1 represents the execution of a procedure `f` with inputs `a`, `b`, and `c`. In stream-based or data-flow languages, expressions `a`, `b`, and `c` might produce a stream of values (tokens) which `f` would consume, producing a stream of output. However, in an applicative language like Scheme, the figure generally represents a one-time function application of `f` to the values of expressions `a`, `b`, and `c`.

The idea of a Pscheme port is to capture the arc as an first-class entity into which values can be sent. Thus, if a port could be created for each of arcs emanating from a, b, and c in Figure 1, then each time a trio

1. Street Address: 251 Mercer Street, New York, NY 10012. email: chiyao@cs.nyu.edu, goldberg@cs.nyu.edu.

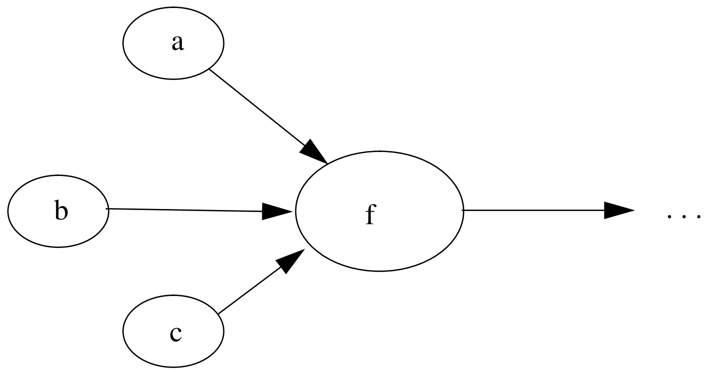

Figure 1. A simple dataflow diagram

of values are sent along the three ports, f is invoked to consume the input and produce output. This allows a natural way to create data-flow parallel programs using a Scheme-based language.

In the following sections, we describe in detail how ports are created and used, and how parallelism is expressed in Pscheme. Along the way, we hope to show that ports are a very natural extension of first-class continuations for parallel computing.

2. Parallelism, Control, and Synchronization in Pscheme

Pscheme contains relatively few constructs for specifying parallel program behavior. These constructs, primarily **`pcall`**, **`call/mp`**, **`call/sp`**, **`throw`**, **`die`**, and **`exclusive`** are simple to describe and use, but are very powerful when used in conjunction with each other.

2.1. pcall: A simple construct for expressing parallelism

Before discussing ports and continuations, we introduce **`pcall`**, the basic construct for expressing parallelism in Pscheme. It is a simple fork-join construct seen in many languages. The innovation of the work described here results from the interaction of pcall with ports, as described later in this paper.

The pcall construct takes the form

$$(\mathbf{pcall}\ e_0\ e_1\ \ldots\ e_n)$$

and causes the evaluation of e_0, e_1, ... , e_n in parallel. When each of the expressions e_i has been evaluated to its value v_i, the procedure value v_0 is invoked with arguments v_1 ... v_n. It is important to note that the body of the procedure is not invoked until the parallel evaluation of the arguments has completed. Thus, pcall serves as a barrier synchronization for the evaluation of the expressions e_0 ... e_n. This synchronization becomes very important in our later examples.

2.2. Ports: Parallel Extensions of Continuations

The expression

```
(pcall f a b c)
```

can be thought of as being graphically represented by the picture in Figure 1. When a, b, and c have been evaluated, their values are sent along their corresponding arcs to f.

In order to capture the arcs as first-class values, we introduce a construct called ***call-with-current-multi-port***, which is written **`call/mp`** (the term “multi-” will be addressed later). Like its sequential analog **`call/cc`**, **`call/mp`** takes a single parameter which is a function that takes a single parameter, the current port –that is, the arc.

Thus, the expression

```
(pcall f   (call/mp (lambda (p1) a))
           (call/mp (lambda (p2) b))
           (call/mp (lambda (p3) c)))
```

binds **p1**, **p2**, and **p3** to the arcs as shown in Figure 1.

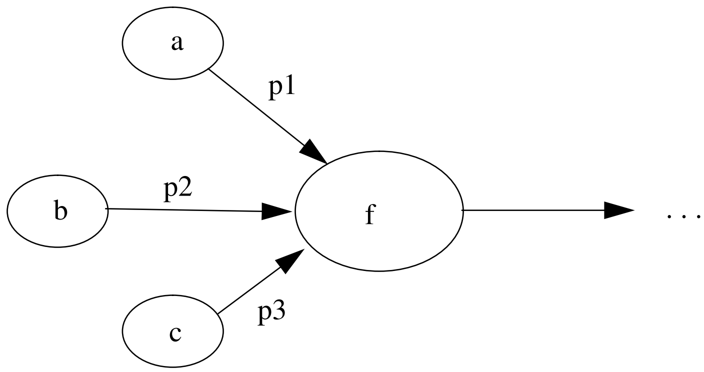

Figure 2. The arcs are captured by ports P1, P2, and P3.

Looking at the illustration as a data-flow graph, it is clear that f is invoked whenever a trio of values are sent to ports **p1**, **p2**, and **p3**. It was mentioned above that one way to produce values along the arcs is for the evaluation of **a**, **b**, and **c** to complete. However, now that **p1**, **p2**, and **p3** are tangible objects, values can also be sent down the arcs by explicitly "throwing" the values to the respective ports (using the **throw** construct described below).

With the explicit use of ports, f can be invoked many times, once each time all of its input ports have a value available (due to a throw or a returned value). Thus, not only is parallelism generated by the use of pcall, but also by multiple invocations of f occurring in parallel due to multiple values being thrown to each of f 's input ports. The image of a single f being a consumer of the values is no longer accurate. A new activation of f is created for each trio of values thrown to its input ports (later on, we will introduce a new construct, **exclusive**, that specifies that only one activation of f can execute at any given time).

2.3. Multi vs. Single Ports

It might also be desirable for a given function to be invoked only once, but to be able to invoke it either by having its arguments return values, or by explicitly passing values to its input ports. That is, it might be desirable to restrict the flow of data to each input port to a single value. After a value is sent to the port, the port shuts down and refuses to accept any more values. Subsequent values thrown to that port have no effect.

This special kind of port is called a single-port, and is made into a first-class object by the construct **call/sp**. The usual port, one to which many values can be sent, is therefore called a multi-port.

2.4. Pscheme Thread Creation and Termination

In Pscheme, threads (processes) are created by the pcall construct, as described above. In addition, the **throw** construct, which sends a value to a port explicitly, sometimes also creates a thread. Thread termination is specified by the **die** construct, which terminates the current thread's computation.

(throw e1 e2) evaluates expressions **e1** and **e2**, resulting in a port *p* and a value *v* respectively, sends *v* to *p*, and returns *v*. If *p* is a closed single-port, then the throw has no effect other than to return *v*. However, If *p* is a multi-port or an open single-port, a new thread is created if the receiver is not blocked waiting for values from other ports. Otherwise, *v* is queued within *p*. If a process throws one value after another to the same port, the arrival order is the same as the sending order.

(die) terminates the computation of the current thread and returns no value. A **(die)** in a parent process does not kill its child processes.

Here is an example of the typical use of a single-port; to commit to the first value computed by competing parallel processes. For example, suppose a binary tree is encoded in the list form

(key left-child right-child).

To find an element with a key value in this tree, a parallel search along all branches can be performed. The first value passed to the answer port is the value that **find** returns.

```
(define (find1 x tr port)
   (cond ((null? tr) #f)
         ((eq? x (car tr)) (throw port #t) (die))
         (else (pcall  (lambda (a1 a2) #f)
                       (find1 x (cadr tr) port)
                       (find1 x (caddr tr) port)))
   ))

(define (find elt tree)
   (call/sp (lambda (p) (find1 elt tree p)))))
```

2.5. Exclusive Functions

As described above, a new activation of a function is created each time there is a value available on each of its input ports. Thus, this is a method for creating multiple invocations of the same function in parallel.

It might be the case, however, that it is desirable for there to be only one invocation of the function active at any given time. Such a function is said to be *exclusive* (and has essentially the same behavior as a Hoare monitor). An exclusive function is desirable, for example, if it modifies some data structure that requires mutually exclusive access. Another (related) possibility is that the function produces output or results whose order is important.

The Pscheme construct **exclusive** is used to create exclusive functions. The expression

```
(exclusive e)
```

evaluates **e** to some function value f, and returns a function f' with the same behavior as f except that f' is exclusive. When f' is called, subsequent requests to call f' are blocked and queued until the current invocation of f' returns or a **(die)** is executed during the evaluation of the current invocation of f'.

Consider the program below for comparing the fringes of two trees

```
(define (samefringe t1 t2)
   (call/sp (lambda(p)
               (pcall  (exclusive compare)
                       (call/mp (lambda(p1) (search t1 p1)))
                       (call/mp (lambda(p2)(search t2 p2)))))))

(define (search tree outport)
   (cond ((null? tree) nil)
         ((atom? tree) (throw outport tree))
         (else (search (car tree) outport)
               (search (cdr tree) outport))
   ))

(define (compare a b)
   (cond ((null? a) (null? b))
         ((not (eq? a b)) #f)
         (else (die))
   ))
```

Notice that parallelism between the searching of the two trees is desirable. However, an exclusive version of **`compare`** is essential to avoid a race condition in which **`compare`** reports success because the rightmost leaves of the two trees are identical, even though some of the interior leaves (which may be different in the two trees) have not yet been compared. The **`call/sp`** in samefringe avoids the problem of success being reported after failure has already been reported.

`(exclusive f)` returns a function *f '* which can have only one activation at any time. However, during the evaluation of *f '*, exclusiveness can be violated if **`pcall`** is invoked, or a multi-port is captured inside of *f '* and gets thrown a value. The language definition only enforces exclusiveness of the *entry point* of an exclusive function. The effect of creating parallel threads during the evaluation of an exclusive function is left unspecified by the language definition and should be avoided.

3. The relationship between continuations and ports

Although we introduced **`call/mp`** in the context of **`pcall`**, they are really orthogonal constructs. We saw above that **`pcall`** is useful by itself, and is actually a very common parallel construct. What is the effect of using **`call/mp`** without using **`pcall`**? Consider the expression

```
(f  (call/mp (lambda (p1) a))
    (call/mp (lambda (p2) b))
    (call/mp (lambda (p3) c))).
```

Because the evaluation of expressions **`a`**, **`b`**, and **`c`** occurs sequentially, the illustration used in Figure 1 would be misleading for this example. Assuming a left-to-right evaluation order among function arguments, the evaluation order is clearly seen when the expression is represented in continuation passing style (cps). Assuming k is the continuation for the entire expression, the cps-converted expression would be

```
(a′ (lambda (v1) (b′ (lambda (v2) (c′ (lambda (v3) (f′ v1 v2 v3 k)))))))
```

where **`a′`**, **`b′`**, **`c′`**, and **`f′`** are the cps-converted versions of **`a`**, **`b`**, **`c`**, and **`f`**, respectively. This might be illustrated graphically as in Figure 3, in which each continuation is a procedure with a single input port. When

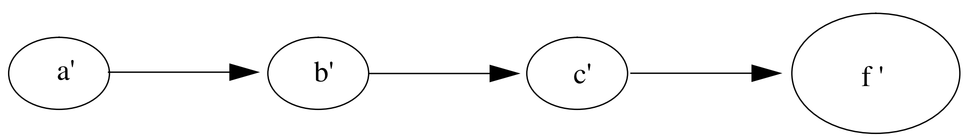

Figure 3. The sequential representation of function application using cps

a value is sent to the port, the continuation is invoked. This is exactly the behavior caused by the capture of the continuation using **`call/cc`** and the invocation of that continuation. This is yet another reason that we are convinced that ports are the logical extension of continuations in a parallel Scheme.

In a sequential computation, such as the one pictured above, each function has only one input multi-port. When a value is thrown to that port, a new thread computing the function call is created. This is not the case when call/cc is used in a sequential language. Thus, one way to simulate the effect of invoking a continuation by throwing to a port is to kill off the throwing process. Specifically, the expression

```
(call/cc f)
```

in ordinary (sequential) Scheme is equivalent to

```
(call/mp (lambda (p) (f (lambda (v) (throw p v) (die)))))
```

in Pscheme.

Pscheme provides all features in Scheme except call/cc and first class continuations. When a Pcheme expression contains none of the six parallel constructs described above, its semantics is the same as

Scheme's. That is, expressions are evaluated in applicative order, and the relative evaluation order among the arguments in a function call is unspecified.

Any Pscheme program can only return a final value once. Top-level parallelism, where a program keeps returning values from multi-port outputs, is not allowed. In other words, any top-level expression (the expression that represents the whole program) is implicitly associated with a single-port.

4. Programming in Pscheme

4.1. An application of multi-ports

A typical use of multi-ports is stream based programming, as is often seen in data-flow languages like Lucid [WA85]. For example, to calculate Fibonacci numbers, we observe that the infinite Fibonacci stream f satisfies

```
f = 1 :: 1 :: (add_list f (cdr f))
```

The stream f can be generated by an adding cell with feedback wires to its two inputs as shown in Figure 4.

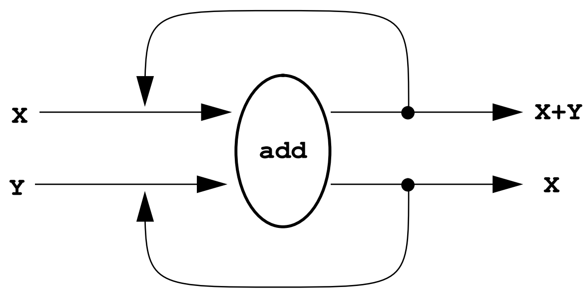

Figure 4. A dataflow diagram for Fibonacci

The input arcs are captured by multi-ports in Pscheme, and the adding cell is simply an exclusive function which also counts the cycle number *n* in order to compute up to *fib(n)*. Since the two input ports are not visible inside **add**, we use two global variables **next1** and **next2** to store the captured ports. The initial values 0 and 1 are sent to the ports and added together after **next1** and **next2** are initialized properly. **add** only returns the sum when the counter reaches *n*. Otherwise, it dies.

```
(define (fib n)
   (letrec ((counter 0) (next1 nil) (next2 nil)
            (add (lambda (x y) (set! counter (+ 1 counter))
               (if (eq? n counter) (+ x y)
                         (begin  (throw next2 x)
                                 (throw next1 (+ x y))
                                 (die))))))
        (pcall  (exclusive add)
                (call/mp (lambda(p) (set! next1 p) 1))
                (call/mp (lambda(p) (set! next2 p) 0)))
   ))
```

4.2. Using pcall and ports to create higher-level parallel constructs

Pscheme's parallel constructs are sufficiently expressive to be able to support a wide variety of parallel programming paradigms. Although programs that make heavy use of **call/mp**, **throw**, etc. may be difficult to read (much as sequential programs that use call/cc are), it is possible to build higher-level parallel constructs to specify more naturally different styles of parallel computation.

Since pcall provides a general synchronization mechanism, and multi-ports provide a queueing mechanism, they can be used to express blocking, synchronous communication and asynchronous communication between processes. In appendix A, we give the Pscheme code for defining semaphores, the Ada rendezvous mechanism, and MultiLisp futures [Ha85].

5. The Operational Semantics of Pscheme

A complete operational semantics of Pscheme is given in [Yao93]. We have provided an operational semantics in the CML style developed by Reppy [Rep91], and Berry, Milner, and Turner [BMT92]. The execution of a Pscheme program is interpreted in terms of transition rules on the states of a parallel environment with concurrent processes and a shared memory. The operational semantics of Pscheme is given in a *port passing style*, resembling the continuation passing style often used for sequential Scheme.

A program will give an answer *a* if and only if there is a series of state transitions between the starting state, which has only one process, and the terminating state outputting *a*. A system state is defined by the active processes, the content of the shared memory, and the data items queued in FIFO ports.

6. An Implementation of Pscheme on a Multiprocessor

Our Pscheme implementation uses queue-based multiprocessing on shared memory multiprocessor. Each newly created thread is placed on a shared global task queue. Each available processor removes threads from the global queue and executes them. A thread is a series of calls to sequential functions in CPS form.

Our Pscheme compiler using a variation of the continuation-passing/closure-passing transformations used by the SML/NJ compiler [App92]. It is extended with assignment conversion and an ordering mechanism implementing ports. The compiler generates C code that runs on NYU's Ultra II computer, an 8-processor shared memory machine with very efficient fetch-and-add operations and parallel queue operations. When the new Ultra III computer becomes operational in early 1994, we will be porting the Pscheme system to it.

7. Related work

Several models have been proposed to build control into parallel LISPs. Many of them rely on parallel extensions of continuations [Ha90].

Multilisp [Ha85] provides futures to represent fine-grain parallelism. The meaning of continuations are extended in the spirit that any mixed use of futures and continuations expects to have the same result as the sequential program without using futures. Since futures are introduced to support parallelism while retaining the flavor of sequential LISP, it is not appropriate for the control that is hidden by futures to be exposed by continuations. There have been proposals to improve the interaction between futures and continuations, such as the thread legitimacy test of Katz and Weise [KW90].

Hieb and Dybvig's *process continuation* [HD90] provides fine control on parallel processes in an environment where there is a tree structure relationship among them. This means that every process has a parent who spawned it. A process continuation is a *partial*} (or *local*) continuation representing the control state captured back to the spawn point. It is actually the parallel counterpart of Felleisen's $\mathcal{F}$-continuation [Fe88]. Such partial continuations are *functional* in the sense that invoking them is like calling a function representing the captured computation. Capturing them suspends all tasks from below the spawn point to the capture point. Process continuations are useful to control parallel search, implement parallel OR or kill other tasks. On other hand, the degree of parallelism is quite limited, and they only apply to tree structured parallelism.

PaiLisp [IM90][IS92] is another parallel extension of Lisp. It distinguishes single-use and multiple-use continuations. PaiLisp handles continuations according to the identity of the process that captures it (processes can be spawned by any expression.) When invoking a continuation, if the invoking process is the capturing process, sequential semantics applies (i.e. it is a goto). If not, the invoking process goes on, and the capturing process abandons its current computation (if any) and unconditionally jumps to the control point the continuation represents. PaiLisp may not have such precise control as process continuations, but a PaiLisp process has control of other processes in addition to those in the same subtree.

These three versions of parallel continuations have one point in common. None of them allows the invocation of a continuation to fork a new thread, thus the race conditions described in this paper will not occur. Throwing to a Pscheme port may fork a new thread, but this necessitates the use of constructs to control interference among activations. Once of these constructs, **`exclusive`**, gives rise to monitor-like functions such as those found in Qlisp [GM88].

8. Final Remarks

Does Pscheme provide a better method for specifying parallel computations than existing paradigms? Our limited experience seems to suggest that the notion of a port is easily understood by a programmer familiar with call/cc and provides a mechanism for specifying an unusually wide variety of parallel programs. With the completion of our Pscheme implementation, we intend to perform extensive experiments to gauge the expressiveness of Pscheme and its effectiveness as a parallel programming language.

References

[App92]
A. W. Appel. *Compiling with Continuations*, Cambridge University Press, 1992.

[BMT92]
D. Berry, R. Milner, and D.N. Turner. A Semantics for ML Concurrency Primitives, in *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1992.

[Cl91]
W. Clinger, J. Rees et al., *Revised^4 Report on the Algorithmic Language Schem*e, November1991.

[Fe88]
M. Felleisen. The Theory and Practice of First-Class Prompts, in *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1988.

[GM88]
R.P. Gabriel and J. McCarthy. Qlisp, in *Parallel Computation and Computers for Artificial Intelligence*, J.S. Kowalik ed., Kluwer Academic Publishers, 1988.

[Ha85]
R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation, in *ACM Transactions on Programming Languages and Systems*, October 1985.

[Ha90]
R. Halstead. New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools, in *Proceedings of the US/JAPAN Workshop on Parallel Lisp*, LNCS 441, Springer-Verlag, 1990.

[HD90]
R. Hieb and R.K. Dybvig. Continuations and Concurrency, in *Proceedings of the ACM Conference on the Principles and Practice of Parallel Programming*, 1990.

[IM90]
T. Ito and M. Matsui. A Parallel Lisp Language PaiLisp and its Kernel Specification, in *Proceedings of the US/JAPAN Workshop on Parallel Lisp*, LNCS 441, Springer-Verlag, 1990.

[IS92]
T. Ito and T. Seino. On PaiLisp Continuation and its Implementation, in *Proceedings of the ACM SIGPLAN Workshop on Continuations*, June 1992.

[KW90]
M. Katz and D. Weise. Continuing Into the Future: On the Interaction of Futures and First-Class Continuations, in *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

[Rep91]
J.H. Reppy. *An operational semantics of first-class synchronous operations*, Department. of Computer Science, Cornell University, 1991.

[WA85]
W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language,* Academic Press, 1985.

[Yao93]
C. Yao. *An operational semantics of Pscheme*, Department of Computer Science, New York University, 1993.

Appendix A. Pscheme Definitions of Common Parallel Constructs

A.1. Semaphores

(make-semaphore) returns a semaphore *s* such that a thread evaluating **(***s* **`p)** performs the "p" operation on *s*, and returns the symbol **`dontcare**. The call **(***s* **`v)** performs the "v" operation and returns **`dontcare** as well.

s is created by **pcall**'ing a function **f** of two parameters **a1** and **a2**, which executes **(throws a2 a1)** and **(die)**. Any thread evaluating **(***s* **`p)** captures its current single-port *p*, passes it to the port associated with **a1**, and then dies. Evaluating **(***s* **`v)** causes *p* to be passed the port associated with **a2**. *s* is initialized by having the **`dontcare** symbol queued in the port associated with **a2** so that the first thread calling **(***s* **`p)** will not block. Here is the code:

```
(define (make-semaphore)
   (letrec
      ((entry-port nil) (exit-port nil)
       (semaphore
         (lambda(op)
            (case op
               ((p) (call/sp
                      (lambda(p)
                         (throw entry-port (lambda() (throw p `dontcare)
                                                       (die)))
                         (die))))
               ((v) (throw exit-port `dontcare))))))
     (pcall  (lambda (next-process release-signal) (next-process))
             (call/mp (lambda(p) (set! entry-port p) (lambda() semaphore)))
             (call/mp(lambda(p)(set! exit-port p)
                                (throw exit-port `dontcare))))))
```

A.2. Ada Rendezvous

(make-rendezvous f) returns what we call a rendezvous object *r* to serve as, in Ada terms, an entry of a task. (In Ada, a rendezvous communication occurs when one task calls the entry of another task.) A thread evaluating **((***r* **`send)** *arg-list***)** causes f to be applied to *arg-list*. It synchronizes with **(***r* **`accept)** being called by another thread. Both the send and accept expressions return the result value of **(apply f** *arg-list***)**.

Rendezvous objects are implemented using **pcall** and multi-ports in a similar manner to semaphores. Both the caller and the callee of a rendezvous communication will capture their current single-ports and pass them to a pair of synchronizing input ports to a **pcall**'ed function which evaluates **(apply f** *arg-list***)** and passes the result back to the caller and callee.

```
(define (make-rendezvous f)
   (letrec
      ((port1 nil) (port2 nil)
       (rendezvous
         (lambda (op)
            (case op
               ((send) (lambda args
                         (call/sp (lambda (p)
                                    (throw port1
                                       (lambda() (throw p (apply f args))))
                                    (die)))))
               ((accept) (call/sp (lambda(p) (throw port2 p) (die)))))))
     (call/sp
       (lambda (result)
          (pcall(lambda (sender-thunk receiver-port)
                  (throw receiver-port (sender-thunk))
                  (die))
               (call/mp (lambda(p) (set! port1 p) (lambda() rendezvous)))
               (call/mp (lambda(P) (set! port2 p) result)))))))
```

A.3. MultiLisp futures

(future *exp***)** initiates a new thread evaluating *exp*, and returns a single-entry future object *f*, which can be stored or passed to other functions. A strict operator needing *exp*'s value *v* can call **(future-val** *f* **)**, which returns *v* immediately if the evaluation of *exp* has completed, or blocks until *v* is available. Once *exp* is evaluated, it is never re-evaluated.

Similarly, a future object can be created by **pcall**'ing a function **f** of two parameters **val** and **req**, representing the future value and the value requester's current port, respectively. Before the future value *v* is available, there might be several requests for it, so **f** performs **(throw req val)**, and then throws **val** back to the port associated with **val** to synchronize with subsequent requests.

```
(def-syntax (future exp)
   (make-future (lambda() (call/sp (lambda(p) exp)))))

(define (future-val future-object) (future-object))

(define (make-future thunk)
   (call/sp
    (lambda (return)
      (letrec  ((value nil) (touched? nil) (val-port nil) (req-port nil)
                (future-obj
                  (lambda()
                     (if touched?
                        value
                        (call/sp (lambda(p) (throw req-port p)(die))))))
         (pcall   (lambda(val req)
                     (throw req val) (throw val-port val) (die))
                  (call/mp (lambda(p)(set! val-port p)
                                     (set! value (thunk))
                                     (set! touched? #t)
                                    value))
                  (call/mp (lambda(p)(set! req-port p)
                                     (throw return future-obj)
                                     (die))))))))
```