Features for Multiprocessing in SPUR Lisp

Benjamin Zorn
Paul Hilfinger
Kinson Ho
James Larus
Luigi Semenzato

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley California

4 March 1988

[0]This research was funded in part by DARPA contract number N00039-85-C-0269 as part of the SPUR research project.

Abstract

This paper describes simple extensions to Common Lisp for concurrent computation on multiprocessors. Functions for process creation, communication, and synchronization are described. Multiple threads of control are created with *process* objects. Communication and synchronization are managed using *mailboxes*. *Signals* provide asynchronous communication between processes. SPUR Lisp includes *future* and *delay* values, which were first introduced in Multilisp [6]. These features provide a flexible and efficient basis on which higher-level multiprocessing abstractions can be implemented and studied.

Contents

1 Introduction

SPUR Lisp is a superset of Common Lisp [16] that will run on the SPUR multiprocessor workstation that is being developed at Berkeley [8]. Part of the goal of the SPUR project is to build multiprocessor workstations that can be used to study different styles of concurrent programming. We intend to use SPUR Lisp as a research vehicle to understand and experiment with different programming abstractions such as the queue-base multiprocessing proposed by Gabriel and McCarthy [4], Halstead's Multilisp [6], Connection Machine Lisp proposed by Steele and Hillis [17], and techniques for automatic restructuring [7,11]. To accomplish this goal, we designed a flexible and efficient set of multiprocessing primitives that can be used to implement a wide range of high-level abstractions.

Our design is based upon a small set of strongly-held beliefs that have provided guidance in making important decisions. We believe that language features should not be provided unless there is clear evidence that they are useful to programmers. To this end, we support our design decisions with examples that illustrate the necessity of our language features. We also believe that features should not be defined if they can be easily and efficiently implemented by features already in the language. A consequence of these decisions is that features in this document are low-level; they form a basis on which a set of higher-level abstractions can be implemented.

This document is intended for two audiences. We hope to provide a casual reader with an overview of the language features, descriptions of the functions and special forms, and examples their use. However, this document is also a reference for language implementors and contains a careful and concise description of the semantics of the features.

This paper is organized in the following way. Section 2 describes processes and interactions between processes. Section 3 describes mechanisms for communication and synchronization, while Section 4 describes asynchronous communication between processes. Section 5 describes the semantics of futures and the functions used to support futures. Section 6 presents several examples that illustrate the use of the features described in this document.

2 Processes

The basic mechanism for multiprocessing in SPUR Lisp is the *process*, which is a Lisp object that embodies an independent thread of control. We made processes lightweight so that abstractions using processes can take advantage of fine-grained parallelism. Processes should require little overhead to create and are intended be used freely by programmers for even relatively small computations.

2.1 Processes

Processes are created by executing the special form `make-process`, which takes an arbitrary form and creates an independent thread of control to evaluate it. Processes terminate in two ways: by completing their evaluation normally or by being terminated explicitly. A process executes until one of these two events occurs, even if all references to the process are deleted (i.e., an non-terminated process cannot be garbage collected). Processes can be in one of

several states. The possible states are: `:executing`, which is state of an executing process; `:blocked`, which is the state that a process enters when it waits for a shared resource (like a mailbox or processor); `:terminated`, which is the state that indicates that the process has completed execution, and `:suspended`, which is the state entered when a programmer explicitly suspends a process. Processes can inquire about the states of other processes using the `process-state` function.

Processes directly affect other processes using three primitive operations: `kill-process`, `suspend-process`, and `resume-process`. `suspend-process` and `resume-process` provide an abstract mechanism that we foresee being used mostly at the interface level, where a user may want to suspend the state of a process in order to debug it. Suspended processes are resumed with the `resume-process` function. When a suspended process is resumed, all signals that were enabled prior to its suspension are re-enabled (see Section 4).

`kill-process` immediately terminates a process. In many cases this is not desirable because it may cause a process to leave data structures in an inconsistent state (e.g., the process may leave files open). Because `kill-process` does not allow clean-up actions to be taken by the killed process, a predefined signal `:sigkill` is provided (see Section 4.1).

Note that there are no protection mechanisms built into processes at the lowest level. Any process with a reference to another process is free to copy the reference, obtain the status of the process, or send signals to that process. Other development systems such as the Symbolics 3600 [13] and the Xerox Interlisp system [1] have been designed with the belief that protection mechanisms are a disadvantage to a capable user. Because SPUR is intended to be used as a research tool, we also believe that protection mechanisms are unnecessary.

2.2 Changes to Common Lisp Semantics

Each process that is created shares the heap with all other processes and has its own private stack areas and registers. Special binding in the context of shared memory multiprocessing needs further explanation. The semantics of SPUR Lisp special variable binding are best described in terms of ordinary deep binding of dynamic variables. When a process is created, all special bindings of its parent are visible to it. When a process introduces new bindings of special variables, these shadow the bindings of the variables in its parent. These new bindings are in turn visible to any child processes that the process creates. Note that the bindings are independent of the processes. That is, a process may introduce some special bindings, spawn children, and then die. The bindings that a process creates may persist after the process dies because the descendents of the process may still use the bindings. The obvious implementation of these semantics is to make the bindings of a process into a set of name/value pairs and a pointer to the bindings of the parent. These bindings are like any other Lisp object in that when all references to them disappear, they can be garbage collected.

Because Common Lisp was designed as a uniprocessor Lisp standard, there are features and functions in Common Lisp that require careful use in a multiprocessor Lisp environment. In particular, many operations change heap structures that are visible to all processes. For example, package operations, like `use-package`, change the globally visible state of a package. `defun` redefines the globally visible function definition of a symbol. Implementations

of these primitives must make sure that concurrent executions of such global operations are serialized. Common Lisp also defines many global variables, particularly in the context of the I/O system, such as `*print-level*` and `*standard-output*`, which initially are shared between all processes. To create its own instance of these global variables, a process is required to rebind the variable. SPUR Lisp processes can be extended so that they always rebind a useful subset of the Common Lisp global variables.

2.3 Process Functions

This section contains descriptions of process functions in the same format used in *Common Lisp: The Language* [16]. In later sections we provide similar descriptions for mailboxes, signals, and futures. Throughout these descriptions, we adhere to the Common Lisp notation for error conditions, with one minor change. To avoid confusion over the word "signal", where the Common Lisp book says "an error is signaled," we say "an error is raised." When we say "an error is raised," this means that all implementations are required to detect the error and call `error` or `cerror`. When we say "it is an error," this means that programs containing such errors are invalid but implementations are not required to detect the error. In reference to types, if an argument "must be" a particular type, this means "it is an error" to provide another type. Because error handling functions are not currently part of the Common Lisp definition, we do not specify what actions occur when an error is raised, other than to say that either `error` or `cerror` is called.

`make-process` *form* `&optional` *process-name* [Special Form]

`make-process` takes a form and immediately returns a process object with the side effect that an independent process is created to evaluate the form. Evaluation of the form begins immediately. *process-name* must be a string or symbol and is used for debugging and documentation. If *process-name* is unspecified, it defaults to "anonymous process".

`processp` *object* [Function]

`processp` returns *t* if *object* is a SPUR Lisp process and *nil* otherwise.

`process-state` *process* [Function]

This function returns the current process state of its argument, which must be a process. The information provided by process state is the instantaneous process state at the time of the call. The values this function may return are: `:executing`, `:blocked`, `:terminated`, and `:suspended`. Implementations are allowed to return an additional value from `process-state` that describes the state of the process in more detail. Portable programs should not rely on this additional value.

`kill-process` *process* [Function]

`kill-process` terminates its argument, which must be a process. `kill-process` does not return until the process argument is terminated. The `kill-process` function should

only be used in cases in which no clean-up actions need to be taken by the killed process before terminating. If the user wants unwind-protects executed before terminating, he should use the `:sigkill` signal with the `signal-process` form (see Section 4.1). Killing an already terminated process raises an error. `kill-process` returns the process killed.

`suspend-process` *process* [Function]

`suspend-process` suspends its argument, which must be a process. `suspend-process` does not return until the process argument is suspended. The suspended process enters the `:suspended` state with all its signals disabled. The process continues executing only after being explicitly caused to resume. Suspending a suspended process has no effect. Suspending a terminated process raises an error. `suspend-process` returns the process suspended.

`resume-process` *process* [Function]

`resume-process` causes its argument, which must be a process, to resume execution if it is currently suspended. When a process is resumed, all signals that were enabled before it was suspended are re-enabled. Resuming an executing process has no effect. Resuming a terminated process raises an error. `resume-process` returns the process resumed.

`*self-process*` [Variable]

`*self-process*` is a special variable always bound to the currently executing process.

3 Communication and Synchronization

Before the details of communication and synchronization are presented, we need to describe the semantics of concurrent read and write operations with multiple processes in a shared global memory. In Lisp, the basic unit of measure is an object reference, (i.e., a typed pointer or immediate datum) referred to here simply as a pointer. In SPUR Lisp, reading a pointer from memory and writing a pointer to memory are atomic operations. If one process writes a pointer and another reads it, then the process reading the pointer either gets the old value or the new value, but not an intermediate, inconsistent value.

3.1 Mailboxes

Choosing a synchronization mechanism is an important part of designing multiprocessing primitives. Many mechanisms have been proposed, ranging from Dijkstra's semaphores [2], to Hoare's monitors [9], to the rendezvous in Ada [10]. We concluded that simple semaphores are too low-level for most applications. Monitors, which are best suited for client/server applications, are not general enough to support many interesting styles of multiprocessing. We chose a simple mechanism that combines synchronization and communication in an intuitive way. The mechanism is called a *mailbox* and is loosely based on mailboxes as defined in the RED language [14]. To avoid confusion, we emphasize that mailboxes and

messages in SPUR Lisp are not related to message passing as defined in the distributed operating systems literature. In particular, message passing in distributed systems typically works between non-shared memory computers; our messages require a shared memory implementation.

In SPUR Lisp, a mailbox is an unbounded buffer with synchronization. A mail item, or *message*, is any Lisp object reference. There are two major operations on a mailbox: **send**, which places a message in the mailbox, and **receive**, which removes a message from a mailbox. The send and receive operations on mailboxes are implicitly synchronized. For a single mailbox, there can be at most one send operation and one receive operation being executed concurrently; additional operations must be serialized. The order of serialization of concurrent mailbox operations is not specified.

Mailboxes can be used for synchronization. A process receiving from an empty mailbox blocks until there is a message to receive. After unblocking, the receiving process receives the first message from the mailbox and continues. Examples illustrating the use of mailboxes for other common synchronization constructs, such as semaphores and locks, are presented in Section 6.

Mail is removed from mailboxes in first-in, first-out order. Mail sent from a single process to a mailbox is enqueued in the mailbox in the order it was sent. No interprocess order is guaranteed if multiple processes send mail to the same mailbox. Likewise, if multiple processes receive from the same mailbox, then the order of the receives is not specified.

Receive has the ability to read from any one of a number of mailboxes, a facility that we call *multiple receive*. If the argument to receive is a Common Lisp sequence of mailboxes, mail is received from one of them. The order of mailboxes in the sequence is significant. Intuitively, the mailbox sequence is scanned from the beginning and the first mailbox with mail satisfies the receive. If multiple processes concurrently receive from the mailboxes in the sequence, however, race conditions can cause mail to appear to be received out of order. The exact semantics of multiple receive are described in the next subsection.

SPUR Lisp provides a timing mechanism with the function **send-after-delay**, which sends a message to a mailbox after a delay of at least the specified length. The process calling **send-after-delay** continues immediately and does not wait for the send to complete. **send-after-delay** can be used for a number of common timing constructs that are illustrated in Section 6.

After discussing the communication features that were included in SPUR Lisp, we now mention what has been left out, and why. Other multiprocessing languages provide additional structure to the communication primitives. Some applications require a mail communication system that has structured messages containing, for example, a sender's return mailbox or process reference. Another kind of structured mail has message fields that are explicitly typed, as in RED [14]. Either kind of structured mail can be built upon our mailbox primitive. We decided to provide a simple communication mechanism so the overhead of complex mail structures is avoided when unnecessary.

We also considered allowing processes to lock and unlock mailboxes. This feature would allow higher-level structured mailbox abstractions like structured receive (receive based upon the contents of the mailbox) and conditional receive (receive based upon an arbitrary boolean condition). Several multiprocessing languages including PLITS [3] and Linda [5] provide such sophisticated receive facilities. SPUR Lisp mailboxes can be extended to be

lockable, and higher-level mailbox abstractions can be based upon this extended mailbox type.[1]

3.2 Mailbox Functions

This section contains a description of the SPUR Lisp mailbox functions.

`make-mailbox &optional` *mailbox-name* [Function]

`make-mailbox` creates a new mailbox and returns it. *mailbox-name* must be a symbol or string associated with a mailbox that serves to document its function. If *mailbox-name* is unspecified, it defaults to "anonymous mailbox."

`mailboxp` *object* [Function]

`mailboxp` returns *t* if *object* is a SPUR Lisp mailbox, and *nil* otherwise.

`send` *message mbox* [Function]

`send` adds a message to a mailbox. *mbox* must be a mailbox. `send` is an atomic operation. Concurrent `send` operations to the same mailbox are serialized in an unspecified order. Mail sent to a mailbox by a single process is guaranteed to be received from that mailbox in the same order as it was sent. Send returns the mailbox being sent to.

`send-after-delay` *message mbox delay* [Function]

`send-after-delay` sends a message to a mailbox after a delay at least as long as the duration specied by the argument. *mbox* must be a mailbox. *delay* must be an integer time unit in terms of internal-time-units (upon which *internal-time-units-per-second* is based). The process calling `send-after-delay` continues immediately and does not wait for the send to take place. Like `send`, `send-after-delay` returns the mailbox being sent to.

`receive` *mailboxes* [Function]

`receive` removes a message from a mailbox in a sequence and returns the message. `receive` is an atomic operation. Concurrent `receive` operations on the same mailbox are serialized in an unspecified order. *mailboxes* must be either a single mailbox or a Common Lisp sequence of mailboxes. The process calling `receive` blocks if there is no mail in any mailbox in the sequence. If the sequence of mailboxes is empty, `receive` raises an error. Intuitively, if more than one mailbox in the sequence contains mail, `receive` removes mail from the first mailbox in the sequence that contains mail and returns the message. More specifically, if all mailboxes in the sequence are initially empty, mail is returned from

[1] The greatest argument for including lockable mailboxes in the original design is that user-level lockable mailboxes are much less efficient than system provided lockable mailboxes. If lockable mailboxes become widely used by SPUR Lisp programmers, we will add them as a standard feature at a later time.

any mailbox in the sequence that receives mail. If some of the mailboxes initially contain mail, there are two cases. In the absence of concurrent receives on any mailbox in the sequence, mail is returned from a mailbox in the sequence no later than the first mailbox that contained mail when `receive` was called. In the presence of concurrent receives on mailboxes in the sequence, the message returned from `receive` is unspecified. If two or more processes receive from two sequences whose intersection is not empty, the processes are guaranteed not to deadlock.[2] `receive` returns two values, the message received and the mailbox from which the message was received.

`mailbox-empty-p` *mailbox* [Function]

`mailbox-empty-p` returns *t* if the mailbox is empty and *nil* otherwise.

`mailbox-message-count` *mailbox* [Function]

`mailbox-message-count` returns the current number of messages in the mailbox.

The value returned by `mailbox-empty-p` and `mailbox-message-count` is only correct for an instant, and may be incorrect by the time that the call returns. This information should only be considered as a hint about the state of the mailbox.

3.3 Interactions Between Processes and Mailboxes

Process operations can interact with mailbox operations. In general, mailbox operations are atomic and complete before process operations can occur. Processes blocked on a receive operation can be affected by interprocess operations. In this case, the following actions occur:

1. The blocked receive is canceled. In all respects, it appears as though the receive never started.
2. The interprocess operation takes place (i.e., kill, suspend, signal).
3. If the operation was a signal and no non-local goto takes place, the receive is re-executed after the signal handler completes, otherwise the receive is ignored.
4. If the operation was a suspend, the receive is re-executed after the process is resumed.

4 Signals

Mailboxes are the primary communication medium between processes. The second mechanism, *signals*, allow processes to asynchronously interrupt other processes. Signals are intended to be used sparingly. Each process is responsible for defining handlers for signals that are sent to it. Signal handlers cannot be semantically nested, as in Ada [10] or

[2] These semantics are provided to assure the user that multiple receives cannot easily cause deadlocks. As an implementation note, we recommend that multiple receive be implemented so that a process only locks one mailbox at a time.

CLU [12], but signal handlers can be redefined dynamically. Signal handlers are functions that are executed asynchronously when a process receives a signal. The dynamic context of a signal handler, which includes the active catch frames and special variable bindings, is the same as that of the signaled process when the interrupt arrives. The lexical environment of the signal handler is different from that of the signaled process.

When a signal is handled, all signals to that process are disabled for the duration of the handler. By disabled, we mean that no interrupts are taken and subsequent signals are enqueued in a FIFO queue. If a user wants to enable a signal during the execution of a signal handler, the `enable-signal` function allows him to do so. `enable-signal` is provided to allow signals of different priorities to exist. When a signal handler completes, the process that handled the signal continues either from the point at which the signal was received, or from a different dynamic extent if the signal handler executed a non-local goto (e.g., a `throw` or `go`). If a signal handler exits by means of a `throw` or `go`, all signals that were enabled before the signal handler was entered are re-enabled as soon as the dynamic extent of the signal handler is exited.

Signal handlers can be redefined using the `with-signal-handler` special form, which is similar in syntax to `let`. `with-signal-handler` also allows a user to define a handler for "all other signals." This facility can be used to rebind all signal handlers to a null function if the user wants to ignore signals. If a process does not handle a signal sent to it, or cannot catch a throw generated during its executing, an error is raised. Processes do not inherit signal handlers from the processes that created them. A process abstraction that inherits signal handlers can be built on top of simple processes.

4.1 The :sigkill Signal

A default signal handler for `:sigkill` executes a throw of the symbol `:sigkill` that is caught by a predefined catch frame at the outermost level of the process. This throw causes unwind-protect frames to be executed. After the outermost catch of a throw to `:sigkill` the process terminates. The signal handler for `:sigkill` can be redefined by the user.

4.2 Signal Functions

`signal-process` *signal process* `&rest` *args* [Function]

`signal-process` asynchronously causes an interrupt and transfer of control to a signal handling routine in the process specified. *signal* must be a symbol. `signal-process` returns without waiting for the signal to be handled. Signals sent from the same process are guaranteed to be handled in the signaled process in the same order as they were sent. The signal argument passed is handled by user provided signal handlers defined by the `with-signal-handler` special form. The first argument to the signal handler is the signal itself, and the arguments after the process argument are passed as the remaining arguments to the signal handler. One signal, `:sigkill`, has a predefined handler that executes a throw with the tag `:sigkill`. By default, there is a catch frame provided at the outermost level of a process that catches the tag `:sigkill` and terminates the process. If a signal is sent

to a terminated process, or if a process fails to handle a signal that is sent to it, an error is raised. `signal-process` returns the signal sent to the process.

`with-signal-handler` *({(name signal-handler)}*) {form}** [Special Form]

`with-signal-handler` dynamically associates functions with signals. The bindings are active throughout the execution of the forms in the body. Each *name* must be a symbol and names the signal to be handled. If *name* is *t* and the pair is the last pair in the binding list, then the function associated with this name is used to handle all other signals not specified in the current `with-signal-handler` form. Each *signal-handler* must be a function, which is called with one or more arguments. The first argument to the signal handler is the signal that caused the handler to be invoked. The remaining arguments are the arguments passed in the call to `signal-process`. When a signal handler is executing, all subsequent signals are enqueued in a FIFO queue until the handler completes. When the process leaves the dynamic extent of the signal handler, either by completing normally or by executing a non-local goto (via `throw` or `go`), only signals that were enabled before the signal handler was entered are re-enabled. The value returned by the function handling a signal is ignored. `with-signal-handler` returns the value returned by the last form in its body.

`enable-signal` *signal* [Function]

`enable-signal` allows a signal handler to re-enable a signal. `enable-signal` must be used within the dynamic extent of a signal handler; when the signal handler finishes, the effect of `enable-signal` is undone. Because signal handlers automatically disable all signals throughout their extent, without `enable-signal`, signal handlers could not be interrupted by other signals, and signals with different priority levels could not be implemented. If the argument is not a symbol, an error is raised. If the signal is already enabled, the function has no effect. `enable-signal` returns the signal enabled.

5 Futures

We decided to include in SPUR Lisp the *future* construct described by Halstead [6]. Futures are the only high-level abstraction defined in this document. As already mentioned, we intend to use the mechanisms defined here to implement a variety of interesting high-level multiprocessing abstractions. Most high-level abstractions like `qlambda` [4] and `pcall` [6] require no changes to the traditional semantics of Lisp. However, since futures involve changing the semantics of computing a value, we include our interpretation of those semantics in this document.

A future creates an "eventual value." By executing (`future X`), the programmer can start the computation of `X` in parallel, and at the same time continue the current computation. The future form returns a place holder value immediately, while at the same time creating a thread of control to evaluate the form `X`. The place holder behaves exactly as a normal object in any context in which the actual value is not required. Thus a future object can be passed as an argument, copied, and placed in a data structure without the actual value of the future object being required. However as soon as its value is required

(e.g., someone adds 1 to the place holder), the semantics of a future are that the process needing the value blocks until the value of the future has been computed. In the same way that a monitor reduces the burden of explicit synchronization on a programmer, a future allows for fork and join with an implicit synchronization and makes parallel programming easier.

To make further discussion of futures easier, we now introduce some terminology. We say a future *begins* when the process computing the value that the future represents starts executing. We say a future *completes* when the process computing the value of the future finishes. We say the future is *required* when another process needs the value of the future. If a future is required before it completes, the process requiring the value blocks. Finally, to distinguish between the future itself and the value that the future represents, we say *future object* to mean the future itself, and *future value* to refer to the value it represents.

After a future has completed, the future value can rightfully replace all occurrences of the future object. In SPUR Lisp, we allow this replacement to take place any time after the future completes. In particular, garbage collection is an ideal time to replace future objects by completed future values.

There is a design issue involving how futures interact with the tests for equality `eq` and `eql`. One approach would be to say that the tests operate on future objects and thus neither requires the future's value. Another approach is to say that `eql` requires a future value and `eq` does not. We feel that both of the interpretations are poor because they mean that existing code written with `eq` and `eql`, but not futures, might fail if suddenly one of the operands was a future. Our design goal for futures is to make future objects as transparent as possible in code. Typically, the value returned by a function should not depend on whether an argument is a future object or any other Lisp object. Our interpretation is that `eq` and `eql` both require the value of future operands. This interpretation may result in greater overhead in implementing these comparisons, but we feel that increased transparency outweighs a small performance penalty. We provide the function `future-eq` for comparisons that do not require that value of their operands.

In order to allow implementations freedom of choice when implementing futures, we do not specify that the thread of control created by a future corresponds to a SPUR Lisp process. In particular, the value of `*self-process*` within an executing future is not specified. Because we provide no way to access the process corresponding to a future, futures cannot be affected by process functions such as `process-signal`, `kill-process`, etc. Futures can terminate in two ways: by returning a value or by executing a non-local goto such as a throw exits the dynamic extent of the future. Executing a throw out of a future raises an error.

delay is a feature similar to future that is also defined in Multilisp and Scheme [15]. While a future begins when the future form is evaluated, a delay begins only when the delay value is required. This behavior allows delay values to be used to represent infinite data structures like streams of integers. As with futures, we assume that `eq` and `eql` require the values of delay operands. `future-eq` also works on delays.

5.1 Future Functions

Functions related to futures are summarized below.[3]

`future` *form* [Special Form]

`future` creates a thread of control to evaluate the form provided as an argument and returns an object that contains a promise of the value of the form whenever it is required. The future completes when the form returns a value. If the form executes a non-local goto exiting its dynamic extent, an error is raised. The future object can be manipulated as any object, but whenever its value is required, the process needing the value blocks until the process computing the value completes.

`delay` *form* [Special Form]

`delay` is a special form similar to `future`, except that the evaluation of the form begins only when the value of the delay object is explicitly required.

`futurep` *object* [Function]

`futurep` returns *t* if *object* is a SPUR Lisp future object and *nil* otherwise. Note that `futurep` does not require the value of the future. After the value of a future has been computed, the implementation is allowed to replace all references to the future object with references to the future value at any time. This implies that the value of `futurep` after a future has completed is either *t* or *nil* depending on whether the future object has been replaced.

`delayp` *object* [Function]

`delayp` returns *t* if *object* is a SPUR Lisp delay and *nil* otherwise. Like `futurep`, `delayp` does not require the value of its argument.

`future-eq` *obj1 obj2* [Function]

`future-eq` is a function that acts exactly like `eq` unless either of the arguments is a future or delay. If either argument is a future or delay, then unlike `eq`, which requires the value of that argument, `future-eq` does not require the value, but compares using the delay or future object itself.

[3] Readers familiar with other Lisp systems containing futures, such as Butterfly Lisp and Multilisp, may notice the absence of the function `touch`, which requires the value of a future and returns it. In the spirit of minimality, we do not include `touch`, because Common Lisp special forms `and` and `or` accomplish the same effect.

6 Examples

In this section we take familiar examples of multiprocessor programming and show how they would be implemented in SPUR Lisp. These examples demonstrate the various features available in SPUR Lisp and show how these features can be used to implement other interesting styles of multiprocessor programming.

6.1 Synchronization Primitives

Send and receive can easily be used to implement other common synchronization primitives. Semaphores can be implemented with mailboxes. A binary semaphore is a mailbox with one message in it. Counting semaphores require n initial messages in them. The primitive operations are `make-semaphore`, `p-semaphore`, and `v-semaphore`. We implement the operations below:

```
(defun make-semaphore (n)
  (let ((semaphore (make-mailbox "semaphore")))
     (dotimes (i n)
       (send nil semaphore))
     semaphore))

(defun p-semaphore (semaphore)
  (receive semaphore))

(defun v-semaphore (semaphore)
  (send nil semaphore))
```

A lock is conceptually simpler than a semaphore. The three operations are `make-lock`, `lock-lock`, and `unlock-lock`. Locks differ from semaphores because `lock-lock` corresponds to test-and-set. `lock-lock` returns a boolean value indicating whether the lock action succeeded. If the process does not obtain the lock, it does not block, as a call to `p-semaphore` would.

We implement a lock as a mailbox. To avoid blocking when testing if a lock is in use, we use a multiple receive on a sequence in which one mailbox is known to have mail. That mailbox can be a dedicated process mailbox, or as in our example, can be created fresh every time we test the lock. The following code implements locks in SPUR Lisp:

```
(defun make-lock ()
  (make-mailbox "lock"))

(defun lock-lock (lock)
  "Perform a non blocking lock operation on the LOCK argument."
  (let ((full-mbox (make-mailbox)))
    ;; make sure there's mail in one mbox
    (send nil full-mbox)
    (multiple-value-bind (msg mbox) (receive (list lock full-mbox))
```

```
      ;; test that we received from the lock or other mailbox
      (if (eql mbox lock)
          t
          nil))))

(defun unlock-lock (lock)
  (send nil lock))
```

6.2 Timing Primitives

SPUR Lisp provides one time related primitive, `send-after-delay`. We can define a `process-sleep` function very simply:

```
(defun process-sleep (delay)
  (let ((mbox (make-mailbox)))
    (send-after-delay nil mbox delay)
    (receive mbox)))
```

Using `send-after-delay`, we can define a receive that only blocks for a specified period. `receive-with-timeout` returns two values, the message and a boolean indicating whether the receive timed out.

```
(defun receive-with-timeout (mbox delay)
  (let ((timeout-mbox (make-mailbox)))
    (send-after-delay nil timeout-mbox delay)
    (multiple-value-bind (msg read-mbox) (receive (list mbox timeout-mbox))
      (if (eql mbox read-mbox)
          (values msg t)
          (values nil nil)))))
```

6.3 More Multiprocessing Constructs

Because mapping functions operate on sequence data structures, it is easy to see how they can be extended for multiprocessing. Here we define `pmapcar`, and `pmapc`. `pmapcar` acts exactly like `mapcar`, except that successive function applications are performed in parallel. This definition of `pmapcar` waits for all the results to be computed before returning. If we had used futures, `pmapcar` could have returned a list of futures immediately. `pmapc` performs the function applications in parallel and returns `nil` immediately.

```
(defun pmapcar (op &rest args)
  "Perform the mapping OP on the ARGS in parallel, returning when all
results have been computed."
  (do* ((args args (mapcar #'cdr args))
        (an-arglist (mapcar #'car args) (mapcar #'car args))
        (mbox (make-mailbox) (make-mailbox))
        (result-mboxes nil)
        (result nil))
```

```
      ((some #'null args)
       (dolist (mbox result-mboxes result)
         (push (receive mbox) result)))
    (push mbox result-mboxes)
    (make-process (send (apply op an-arglist) mbox))))

(defun pmapc (op &rest args)
  "Perform the mapping OP on the ARGS in parallel, returning as soon as all
computations have started."
  (do* ((args args (mapcar #'cdr args))
        (an-arglist (mapcar #'car args) (mapcar #'car args)))
       ((some #'null args) nil)
    (make-process (apply op an-arglist))))
```

The next example is a slight variation of the producers/consumers problem. In this example, we implement a software pipeline in which processes create, filter, and consume data in parallel. We implement a set of general functions for this purpose, and show how they can be combined.

```
;;; Example: a generalized software pipeline where processes create,
;;; filter, and consume data in parallel.

;;; pipeline components: every pipeline has 1 source and sink, and
;;; any number of filters

(defun pipe-source (out-box create-op n end-of-mail)
  "Use CREATE-OP to create N objects and send them to OUT-BOX."
  (dotimes (i n)
    (send (funcall create-op) out-box))
  (send end-of-mail out-box)
  (signal-process :sigkill *self-process*))

(defun pipe-sink (in-box destroy-op end-of-mail)
  "Use DESTROY-OP to consume objects received from IN-BOX."
  (do ((item (receive in-box) (receive in-box)))
      ((eql item end-of-mail)
       (signal-process :sigkill *self-process*))
    (funcall destroy-op item)))

(defun pipe-filter (in-box out-box filter-op end-of-mail)
  "Perform FILTER-OP on items from IN-BOX, sending results to OUT-BOX."
  (do ((item (receive in-box) (receive in-box)))
      ((eql item end-of-mail)
       (send item out-box)
       (signal-process :sigkill *self-process*))
    (send (funcall filter-op item) out-box)))
```

```
;;; example of putting components together

(defun set-up-pipeline ()
  "Create a pipeline with 1 source, 1 sink, and 1 filter process.
Create 1000 items using #'a-create-op and send them through the pipe."
  (let* ((mbox1 (make-mailbox "source to filter mailbox"))
         (mbox2 (make-mailbox "filter to sink mailbox"))
         (EOM :end-of-mail))                    ; end-of-mail
    (make-process (pipe-source mbox1 #'a-create-op 1000 EOM))
    (make-process (pipe-filter mbox1 mbox2 #'a-filter-op EOM))
    (make-process (pipe-sink mbox2 #'a-destroy-op EOM))))
```

6.4 Structured Processes and Mailboxes

The following examples demonstrate the ease with which the primitives that we have defined can be extended into abstract data types to support more structured kinds of multiprocessing. Our first example demonstrates how processes can be augmented to include information about who created them (their parent) and the processes they have created (their children).

```
;;; Structure processes are processes that keep track of the process
;;; that created them and the processes that they have spawned.

(defvar %process-children-table (make-hash-table))
(defvar %process-parents-table (make-hash-table))

(defun make-s-process (expr &optional (process-name process-name-p))
  "Make structured process from an EXPRESSION and call it NAME (optional).
Structured processes keep track of their children and parents."
  (let ((new-process (if process-name-p
                         (make-process expr process-name)
                         (make-process expr))))
    (push new-process (gethash *self-process* %process-children-table))
    (setf (gethash new-process %process-parents-table) *self-process*)
    new-process))

(defun process-children (process)
  "Returns two values: the list of children of the given PROCESS and
a boolean indicating if the process is a structured process."
  (gethash process %process-children-table))

(defun process-parents (process)
  "Return the parent of the given PROCESS.  NIL means that the process
is not a structured process."
  (gethash process %process-parents-table))
```

Structured mailboxes are mailboxes that contain mail items with a return address for the process that sent the mail.

```
(defstruct signed-mail
  message
  sender)

(defun send-signed (message mailbox)
  "Send a MESSAGE containing the sender's identity to a MAILBOX."
  (send (make-signed-mail message *self-process*) mailbox))

(defun receive-signed (mailbox-sequence)
  "Receive a signed message from a MAILBOX-SEQUENCE.  Returns a
multiple value of the message and the sender.  The sender is NIL if
the message was not signed."
  (let ((message (receive mailbox-sequence)))
    (if (signed-mail-p message)
        (values (signed-mail-message message) (signed-mail-sender message))
        (values message nil))))
```

6.5 Parallel Tree Search using Structured Processes

This example demonstrates how the structured process type can be used to implement a simple parallel tree search, which involves creating a process for each node. Each process examines the data in the node, and creates subprocesses for each of the children of the node. When a process discovers a node that satisfies the search, mail is sent to the root of all the processes and a kill signal is sent to its children, which is propagated recursively.

```
;;; Example: use processes to perform a parallel tree search.  We use
;;; structured processes and the process-sleep functions defined in
;;; previous examples.

(defun tree-search (tree search-fn)
  "Search a TREE for a node which satisfies SEARCH-FN.  If there is no such
node, this function never returns."
  (let* ((answer-mbox (make-mailbox))
         (root (make-s-process (tree-process tree answer-mbox search-fn)))
         (answer (receive answer-mbox)))
    (signal-process :sigkill root)
    answer))

(defun tree-process (tree answer search-fn)
  "Search a subtree of a TREE using SEARCH-FN, responding to ANSWER."
  (with-signal-handler
      ((:sigkill #'kill-self-and-children))
```

```
    ;; spawn processes for children
    (unless (tree-leaf-p tree)
      (make-s-process (tree-process (left-child tree) answer search-fn))
      (make-s-process (tree-process (right-child tree) answer search-fn)))
    ;; search the current node
    (if (funcall search-fn (tree-data tree))
        (send tree answer))
    ;; sleep forever (receive on an empty mailbox)
    (receive (make-mailbox))))

(defun kill-self-and-children (signal)
  "Kill a process and all its descendents."
  (dolist (c (process-children *self-process*))
    (signal-process :sigkill c))
  (throw :sigkill nil))
```

6.6 Dataflow Computation in SPUR Lisp

Our final example shows how SPUR Lisp can be used to perform concurrent evaluations of an expression using a dataflow style of computation. For simplicity, we limit expressions to consist only of binary operators and variables. The expression is translated into a *dataflow graph*. Figure 1 illustrates the dataflow graph for the expression `(* a (+ b c))`.

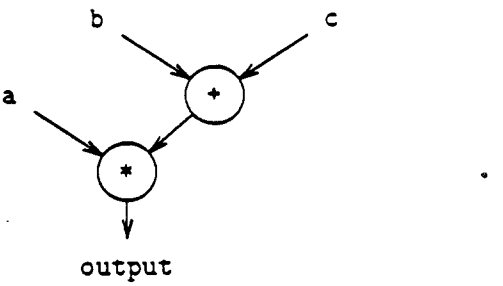

Figure 1: Dataflow graph for the expression `(* a (+ b c))`. Circles represent operators, which are implemented with processes. Arrows represent I/O queues, which are implemented with mailboxes.

Nodes in the graph correspond to operators in the expression and are implemented with processes; variables in the graph correspond to input and output data; and arrows in the graph correspond to I/O queues, which we implement with mailboxes. Each operator has three mailboxes: two for reading the inputs, and one for producing the output, which is the result of the binary operation on the inputs.

The following functions generate dataflow nodes for the + and * operators. They take two mailboxes as arguments, and return a result mailbox. As a side effect, they create a process that reads values from the argument mailboxes, operates on them, and puts the

result in the outgoing mailbox. We make these functions accessible from the property list of the symbols representing the operators.

```
(defun +-node-generator (left right)
  (let ((out (make-mailbox)))
    (make-process (loop (let ((x (receive left))
                              (y (receive right)))
                          (send (+ x y) out))))
    out))

(setf (get '+ 'node-generator) #'+-node-generator)

(defun *-node-generator (left right)
  (let ((out (make-mailbox)))
    (make-process (loop (let ((x (receive left))
                              (y (receive right)))
                          (send (* x y) out))))
    out))

(setf (get '* 'node-generator) #'*-node-generator)
```

The following function takes an expression and creates a dataflow graph for the expression. It assumes that every variable in the expression has an associated mailbox, which can be retrieved from the property list of the symbol representing the variable.

```
(defun setup-dataflow-graph (expression)
  (cond ((symbolp expression)
         (get expression 'dataflow-mbox))
        (t
         ;; assume all other inputs are binary operators
         (funcall (get (first expression) 'node-generator)
                  (setup-dataflow-graph (second expression))
                  (setup-dataflow-graph (third expression))))))
```

The rest of the code will be presented in a top-down fashion. Suppose we wish to create a dataflow graph in the following way:

```
(setq example-dataflow (setup-dataflow '(a b c) '(* a (+ b c))))
```

The first argument to `setup-dataflow` is a list of the input variables to our dataflow computation. The second argument the expression to be evaluated. `setup-dataflow` returns an object, whose structure can be ignored for the moment, which can be used as follows:

```
(run-dataflow example-dataflow '((1 2 3) (4 5 6) (7 8 9))) => (11 26 45)
```

Note that the second argument to `run-dataflow` is a list, whose elements are sequences of values to be fed to each input of the dataflow machine. Every element corresponds to an input variable, in the same order as the list of input variables passed to `setup-dataflow`.

We now look at implementations of `setup-dataflow` and `run-dataflow`. The function `setup-dataflow` creates a list of mailboxes, one for each input variable, and puts them on the property list of each variable. It then creates the dataflow graph, and returns the output mailbox, where the results of the computation will be placed, and the list of input mailboxes.

```
(defun setup-dataflow (input-list expression)
  (let ((mailbox-list
         (mapcar #'(lambda (symbol)
                     (declare (ignore symbol))
                     (make-mailbox))
                 input-list)))
    (mapc #'(lambda (symbol mailbox)
              (setf (get symbol 'dataflow-mbox) mailbox))
          input-list mailbox-list)
    (let ((outbox (setup-dataflow-graph expression)))
      (list outbox mailbox-list))))
```

`run-dataflow` uses `pmapcar` to start one process for each input variable, whose task is to send input values sequentially to its corresponding input mailbox. The results are then collected and returned.

```
(defun run-dataflow (dataflow inputs)
  (let ((outbox (first dataflow))
        (inboxes (second dataflow))
        (results nil))
    (pmapcar #'(lambda (values inbox)
                 (mapc #'(lambda (value)
                           (send value inbox))
                       values))
             inputs inboxes)
    (mapcar #'(lambda (x)
                (declare (ignore x))
                (push (receive outbox) results))
            (first inputs))
    (nreverse results))
```

7 Conclusions

This paper has defined the semantics of multiprocessing features in SPUR Lisp. Those features include *processes*, which embody an independent thread of control; *mailboxes*, which are used for communication and synchronization; *signals*, which are necessary for

asynchronous interrupts between processes; and *futures*, which provide a high-level fork-join with automatic synchronization.

Our design philosophy is to add powerful yet efficient features on which many interesting higher-level designs can be based. We have illustrated the implementation of some higher-level designs in our examples.

Our definition of features for multiprocessing in SPUR Lisp continues to evolve. Any comments or questions would be welcome. Please address them to Ben Zorn, Computer Science Division, Evans Hall, University of California, Berkeley, CA, 94720, USA, or send electronic mail to zorn@ernie.Berkeley.EDU.

References

[1] R. R. Burton, R. M. Kaplan, L. M. Masinter, B. A. Sheil, A. Bell, D. G. Bobrow, L. P. Deutsch, and W. S. Haugeland. Papers on Interlisp-D. Technical Report SSL-80-4. Xerox Palo Alto Research Center, Palo Alto, California, September 1980.

[2] Edsger W. Dijkstra. *Hierarchical Ordering of Sequential Processes*, pages 72–93. Academic Press, 1972.

[3] Jerome A. Feldman. High level programming for distributed computing. *Communications of the ACM*, 22(6):353–368, June 1979.

[4] Richard P. Gabriel and John McCarthy. Queue-based multi-processing Lisp. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 25–43, Austin, Texas, August 1984.

[5] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[6] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[7] W. Ludwell Harrison, III and David A. Padua. PARCEL: Project for the automatic restructuring and concurrent evaluation of Lisp. Technical Report CSRD 653, Center for Supercomputer Research and Development, February 1987. Preliminary.

[8] Mark Hill, James Larus, Susan Eggers, George Taylor, et al. SPUR: A VLSI multiprocessor workstation. *IEEE Computer*, 19(11):8–22, November 1985.

[9] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[10] Jean D. Ichbiah et al. Ada programming language reference manual. Technical Report ANSI/MIL-STD-1815A-1983, ANSI, February 1983.

[11] James R. Larus. Curare: Restructuring Lisp programs for concurrent execution. Technical Report UCB/CSD 87/344, Computer Science Division (EECS), University of California, Berkeley, February 1987.

[12] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Shaffert, R. Scheifler, and A. Snyder. *The CLU Reference Manual.* Lecture Notes in Computer Science. Springer-Verlag, New York, New York, 1981.

[13] David A. Moon. Architecture of the Symbolics 3600. In *Proceedings of the Twelfth Symposium on Computer Architecture*, Boston, Massachusetts, June 1985.

[14] John Nestor and Mary van Deusen. *RED Language Reference Manual.* Intermetrics, Inc., 1979.

[15] Jonathan Reese and William Clinger (*Editors*). Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.

[16] Guy L. Steele, Jr. *Common Lisp: The Language.* Digital Press, Burlington, Massachusetts, 1984.

[17] Guy L. Steele, Jr. and W. Daniel Hillis. Connection machine Lisp: Fine-grained parallel symbolic processing. In *1986 Conference on Lisp and Functional Programming*, Cambridge, Massachusetts, August 1986.