EuLisp Threads: A Concurrency Toolbox

NEIL BERRINGTON   (*N.Berrington@southampton.ac.uk*)
University of Southampton, UK

PETER BROADBERY   (*pab@maths.bath.ac.uk*)
University of Bath, UK

DAVID DE ROURE   (*D.C.DeRoure@southampton.ac.uk*)
University of Southampton, UK

JULIAN PADGET   (*jap@maths.bath.ac.uk*)
University of Bath, UK

Abstract. Many current high level languages have been designed with support for concurrency in mind, providing constructs for the programmer to build explicit parallelism into a program. The EuLisp threads mechanism, in conjunction with locks, and a generic event waiting operation provides a set of primitive tools with which such concurrency abstractions can be constructed. The object system (TELOS) provides a powerful approach to building and controlling these abstractions. We provide a synopsis of this 'concurrency toolbox', and demonstrate the construction of a number of established abstractions using the facilities of EuLisp: pcall, futures, stack groups, channels, CSP and Linda.

1. Introduction

Programs for modern computer systems must frequently address issues relating to concurrency. These programs are becoming more commonplace, especially at applications programming level, as the availability of multiple processors and networked computer systems increases. Recent high-level programming languages offer concurrency features to support the programmer in this task.

EuLisp was designed with concurrency features from the outset. In the Lisp tradition, the intention was not to enforce a particular concurrency model, but rather to provide the primitive tools and means of combination so that programmers (and library-writers) could build the appropriate abstractions for the task at hand.

Programming with multiple threads of control within a single address space is emerging as an important and increasingly common-place paradigm

in modern computing. We note two reasons:

- Solutions to a large class of programming problems often fall naturally into the thread model, particularly programs which respond to asynchronous events. Many programs associated with user interaction and with networks fall into this class, hence this programming style is becoming more common as these application areas expand.

- Programs written with threads can be executed on uniprocessor or multiprocessor hardware (perhaps with distributed shared memory). In the multiprocessor case, the threads can execute on any available processor. This promotes portability with the potential for making good use of the available resources, and provides a degree of scalability. In this way, threads provide a means to harness the power of modern computing platforms.

It is for these reasons that EuLisp adopted a threads mechanism as its primitive concurrency model. The importance and utility of a thread model is demonstrated by the adoption of threads in recent operating systems, for example MACH, DCE[19], Chorus[20]. This trend will facilitate the implementation of EuLisp systems on new platforms.

In contrast with many other concurrent lisp languages[21][8], EuLisp attempts to provide a model of concurrency which can be extended by efficient implementations of concurrency abstractions which are not explicitly supported by the kernel, such as futures and Linda. The thread model is quite general, and does not require pre-emptive scheduling or true multiprocessing, which may be infeasible on some systems. Even without this form of scheduling, the thread model can be used to create useful abstractions that use separate threads as a problem-solving aid, rather than a means of obtaining concurrency.

The EuLisp object system, Telos[2] can be used in order to provide a degree of reflectiveness and extensibility in the thread model — one can effectively provide a protocol for a particular operation, and a simple implementation, and allow the user to extend or specialise it. This specialisation allows the new abstractions to be more of an integrated part of the existing kernel, rather than an additional library with independent ideas about how to create and manipulate threads — we illustrate this later in the Linda section.

In this paper we give an overview of the EuLisp thread model (sections 2–4) and give reasons why the particular primitives have been chosen. In subsequent sections we discuss several different parallel abstractions and their realisation in EuLisp. Finally (section 12), we compare these with existing languages.

2. The EᴜLɪsᴘ threads interface

The EᴜLɪsᴘ definition[14] allows for the concurrent execution of expressions through the use of threads, and atomic communication and mutual exclusion via the use of semaphores. Threads are instances of an abstract data type which represent a flow of control in a program, and they interact with other threads via shared memory. When a thread is created, an initial function is supplied; arguments are provided when the thread is started, and the thread executes the application of the initial function to these arguments. When the function returns, the value of the thread is set to the return value of the function, and the thread completes normally.

The following interface is provided for the creation and control of threads.

`<thread>` A class object representing the class *thread*. This object is preinstalled in the Tᴇʟᴏs hierarchy.

`(make <thread> 'init-function` *function*`)` Allocates, initializes and returns a thread with *function* as its initial function.

`(thread-start` *thread . args*`)` Starts a thread by applying the initial function to the arguments *args*.

`(thread-reschedule)` Indicates that the current thread is prepared to cede control to another thread.

`(thread-value` *thread*`)` Blocks the current thread until *thread* has completed, and returns the result of the application of the initial function to the arguments passed by `thread-start`.

The definition does not force a conforming EᴜLɪsᴘ implementation to adopt any particular scheduling policy, allowing implementors to choose a suitable one for the hardware and operating system platform they are targeting. A portable EᴜLɪsᴘ program must therefore not assume a particular scheduling policy itself; for example, assuming a time-slicing scheduler and therefore never explicitly calling `thread-reschedule` may cause the program to fail on a scheduler which lacks preemptive tasking, where threads must voluntarily give control back the the scheduler.

A fairness guarantee also needs to be stated. Due to the possibility of a host operating system handling the execution of threads[5][1], it is impossible to make a strong guarantee.

However a sufficient, although weak, guarantee is that if a thread reschedules infinitely often, then every other ready thread will also be scheduled infinitely often.

2.1. Condition handling

Conditions raised on a thread may be handled by any condition handlers placed in the dynamic extent of that thread, or by a default handler when no other handlers are present on the thread.

The default handler causes execution on the thread to be aborted and the thread to signal an error on any thread that tries to obtain its value.

A mechanism for threads to raise conditions on other threads is also provided. The function `signal` takes an optional thread argument, and will cause a condition to be raised on that thread. The condition must be an instance of `<thread-condition>`, itself a subclass of the `<condition>` class. This restriction is made so that a handler function for externally signalled conditions need only deal with a well-defined subset of all possible conditions. This enables a handler to distinguish between an internal and external signals. In addition, it would be possible for one thread to cause, say, a division by zero condition to be raised on some other thread — in general, this does not seem to be desirable behaviour.

A conforming implementation guarantees that after the condition has been registered with the target thread, it will be signalled on that thread no later than when the thread is next rescheduled for execution—no guarantee is made that the condition will be raised immediately, even if the thread is currently executing, due to the difficulty of maintaining such a strong guarantee in a distributed environment. The normal procedure is that conditions are processed by threads in registration order.

3. Semaphores

Semaphores are provided for synchronisation between threads and mutual exclusion. The following primitives are provided.

`<lock>` A class object representing the class *lock*. This object is pre-installed in the TELOS hierarchy

`(make <lock>)` Allocates and initializes a new (open) lock.

`(lock` *lock*`)` Perform a `P` operation [6] on *lock*.

`(unlock` *lock*`)` Perform a `V` operation on *lock*.

An implementation is free to choose its strategy for the locking operation: for example blocking a thread on a lock until it becomes free or busy waiting by rescheduling a thread until the lock becomes free.

4. Wait

The generic function `wait` has standard methods defined for a stream and for a thread. The purpose of `wait` is to enable the portable implementation of event-driven programming by allowing programs to *wait* on events.

The method for threads takes a thread to wait on and a time-out period and blocks the current thread until the time-out period has elapsed or the thread being waited on has become determined. A thread is *determined* when it has either finished normally or has had a condition handled by the default handler. The method returns true if a thread became determined (or was already determined) during the call and () otherwise. A call to `wait` with a zero time-out period is equivalent to a poll to see if the thread is determined.

The definition does not specify how the method should be implemented, and, like semaphores, a `wait` could be performed using busy waiting or a blocking mechanism.

This concludes the discussion of the EuLisp thread model. The rest of the paper is concerned with its use in the development of various parallel abstractions.

5. Pcalls

Pcalls (parallel calls) [8] allow functions to be called with their arguments evaluated in parallel. Thus (`pcall + 1 2 3`) is equivalent to (`apply + (list 1 2 3`)) except all the arguments to pcall are evaluated in parallel, including the function argument. It is the programmer's responsibility to decide when to use this construct, and he/she should be aware of the amount of potential parallelism its use could release (possibly too much) and whether side effects in the parameter expressions will behave as expected when evaluated in parallel.

Modelling this construct using threads splits the pcall into three distinct phases:

1. Spawn threads to evaluate each argument;

2. Collect the results of the threads;

3. Perform application.

These phases can be coded using macro and function definitions as shown in Figure 1.

```
(defmacro pcall args
  '(let ((all-args (map thread-value (make-threads ,args))))
     (apply (car all-args) (cdr all-args))))

(defmacro make-threads (args)
  '(list
     ,@(map
         (lambda (a) '(make <thread> 'init-function (lambda () ,a)))
         args)))
```

Figure 1: Macro implementation of `pcall`

6. Futures

Futures[8] introduce the opportunity for concurrent evaluation of expressions whose results will not be needed until some time in the future.

The form (`future` *expression*) returns a place holder which is a handle on the future. In MultiLisp, once a future has been evaluated the place holder is replaced with the value of the expression, and if the value was requested before the future has been determined, then the requesting thread is blocked and waits for the future to finish evaluating.

EuLisp has no support for replacing one object with another, as Multi-Lisp does when replacing place holders with a future's value, and so this transparent mechanism cannot be reproduced. Instead a function is provided which the user can use to obtain the value of a future. This places a restriction on the user who must know which parameters are futures and explicitly request their values.

Using EuLisp's generic functions, this explicit "touching" of futures can be hidden in some functions, for example the generic function `binary+` can be given a method which specialises on and touches futures, thus allowing the `+` function to handle futures invisibly. However, this is not a universally attractive solution because of the vast number of additional methods that must be defined—and the likelihood of forgetting some.

A naïive implementation of futures maps simply onto the EuLisp thread primitives. A future can be represented as a thread evaluating the future expression, in which case a function to get the value of a future maps directly to `thread-value`. However, it is preferable to make futures into a class of their own; a possible implementation of eager task creation is shown in Figure 2.

Many authors have noted that the eager future model, where a process is spawned every time `future` is called, rapidly paralyzes the computational

```
(defmacro future (exp)
  '(let ((future-thread
           (make <thread>
                 'init-function (lambda () ,exp))))
     (thread-start future-thread)
     (make <future> 'future-thread future-thread)))

(defun future-value (future)
  (thread-value (future-thread future)))
```

Figure 2: Eager futures

resources of most systems which become occupied with future management rather than future execution. This problem and policies for future creation are analyzed in detail in [12]. There, it is shown that the simple mechanism of not creating processes when the number awaiting execution exceeds some threshold (load based partitioning) can lead to deadlock. The novel alternative put forward in [12] is lazy task creation, where it is the *parent* that might be executed on another thread, rather than the child. A sketch of this tactic is given in Figure 3.

```
(defmacro future (exp)
  '(let/cc k                                 ;; rest of the program...the parent
     (let* ((child-result
              (make <placeholder>))          ;; for child → parent communication
            (parent-task                     ;; object for stealing
              (lambda ()
                (thread-start                ;; start thread to execute parent code
                  (make <thread> 'init-function k)
                  child-result))))           ;; passing the placeholder
       (enqueue (steal-queue (current-thread)) parent-task)
       ((setter value) child-result ,exp)    ;; evaluate exp...the child
       (if (dequeue (steal-queue (current-thread)) parent-task)
           (k child-result)  ;; carry on because parent was not stolen
         (suicide)))))
```

Figure 3: Lazy task creation

There are a couple points to be made about this sketch: it assumes that `dequeue` is an atomic operation; it requires that a continuation created on one thread can be called on another. This last part is the more significant since, at present, this is precluded by the EuLisp definition. The reason is that multiple threads invoking one continuation would cause expressions

on that continuation to return more than once, making the implementation of EuLisp more complex than we would like. However, it would not be necessary to allow general cross-calling of continuations, just that one can be used as the `init-function` value when making a thread.

```
(defmethod send-channel ((chan <channel>) obj)
  (lock (chan-buffer-guard chan))
  (add-item (chan-buffer chan) obj)
  (unlock (chan-buffer-guard chan)))

(defmethod read-channel ((chan <channel>))
  (let ((read-val nil))
    (lock (chan-buffer-guard chan))
    (setq read-val (remove-item (chan-buffer chan)))
    (unlock (chan-buffer-guard chan))
    (if (value-p read-value)
        (value-of read-val)
      (progn (thread-reschedule)
             (read-channel chan)))))
```

Figure 4: Channel operations

7. Channels

Channels [9] [23] provide an abstraction allowing threads to communicate objects. Communication can either be synchronous or asynchronous.

A channel is modelled using a structure containing the following data:

- A buffer to store objects before they are collected. This could be bounded or unbounded.

- A semaphore to guard access to the buffer resource.

Sending an object down a channel causes that object to be added to the channel's buffer. The sending thread then proceeds. In a bounded buffer the sending thread will block if the buffer is full. It will continue once an object has been removed from the buffer.

Retrieving an object from a channel is the reverse process. If the channel's buffer is empty then the receiving thread will block until data has been sent to the channel. When an object is present it is removed from the buffer (FIFO ordering) and the receiving thread proceeds. The code for each of these operations appears in Figure 4.

```
(defun context-switch (sg)
   (let ((csg *current-stack-group*))
      ((setter status) csg ':resumable)
      ((setter status) sg ':active)
      (setq *current-stack-group* sg)
      (unlock (semaphore sg))        ;;unblock someone else
      (lock  (semaphore csg))))      ;;block self
```

Figure 5: Stack group context switch

8. Stack groups

Stack groups offer a similar abstraction to coroutines. They allow multiple processes to be defined although no concurrency occurs. Stack groups simply pass control around between themselves. This means that they do not use an underlying scheduler, although one of the stack groups may have that role [22].

As with most other control constructs, stack groups can be modelled using continuations. However in EuLisp this is not feasible since full continuations are not supported—instead, we have used threads, which are a packaged and simplified form of continuation. A stack group is represented by a structure containing the thread which runs the stack group's computation, a semaphore which is used to control whether the stack group is executing, the value returned when a stack group has completed, the resumer of the stack group (see below) and the status of the stack group.

To create a new stack group the macro **make-stack-group** takes three arguments:

1. A string containing the stack group's name;

2. A function the stack group is to perform;

3. The argument(s) of the stack group function.

The macro translates into code which allocates a new *stack-group* structure and fills in the required slots. The new stack group is then allowed to run, and it immediately blocks, ready to have control passed to it and begin executing the stack group function. There are a number of ways of passing control between stack groups but all methods involve a context switch. This involves unblocking the stack group we wish to resume and blocking the current stack group. The code for a function implementing this operation is given in Figure 5. There are three means for the programmer to switch stack groups:

1. `stack-group-resume` switches context to a given stack group passing it a given value;

2. `stack-group-funcall` has the same effect as `stack-group-resume` but also sets the resumer slot of the stack group it is resuming, so that it can itself be resumed in the future by `stack-group-return` (see below);

3. `stack-group-return` resumes the stack group which is in its resumer slot.

All three functions set the required slots of the stack groups and call `context-switch`. Finally when a stack group is finished it calls the function `end-stack-group`. This sets the return value slot and unblocks the stack group's resumer so another stack group may execute.

As an example of the use of stack groups we show the code for the classic "same fringe" predicate which takes two binary trees and returns true if both trees have the same fringe (full code is given in the appendix).

9. Either

The `either` construct[17] (also known as the parallel-or operator) takes two expressions and spawns two processes to evaluate them. The construct returns when one of the processes completes, the remaining process is then killed, and the value of the first process is returned. This construct can be used where there is more than one method to arrive at a solution, for example searching, and in providing fault tolerance on unreliable networks.

The either form can be implemented by a macro that performs the following operations:

1. Spawn two threads to evaluate the two expressions;

2. Use the generic function `wait` to block the current thread until one of the spawned threads has finished;

3. Kill the thread which has not finished using `signal`;

4. Return the value of the evaluated expression.

NOTE — `wait` can be extended to wait for a set of threads by defining a new class denoting a group of threads and adding a method for this new class. It is then possible to define this method in terms of waiting for a single thread.

Figure 6 combines the channels primitives with `either` to model a simple database retrieval system. Two database servers retrieve information from

```
(deflocal data '((apple . 20) (banana . 30) (peach . 25)))

(deflocal db1-in  (make-channel))
(deflocal db1-out (make-channel))
(deflocal db2-in  (make-channel))
(deflocal db2-out (make-channel))

(defun server (db input output)
  (let ((message (read-channel input)))
    (send output (assq message db))
    (server db input output)))

(deflocal server1
  (thread-start (make <thread> server) data db1-in db1-out))

(deflocal server2
  (thread-start (make <thread> server) data db2-in db2-out))

(defun search (item in out)
  (send-channel in item)
  (read-channel out))

(defun lookup (item)                         ;; parallel search
  (either (search item db1-in db1-out)
          (search item db2-in db2-out)))
```

Figure 6: A fault tolerant database server

shared data. Requests can be made to lookup an item on either database, or both. By sending the same request to both databases and retrieving the first result returned we gain some fault tolerance—the system should still work even if one server fails.

10.   CSP

The CSP primitives, introduced by Hoare[9] can be used as the basis for a programming language, of which the most well-known example is occam. CSP is a process algebra with three components

- Computation

- Communication

- Process networks

The primitives involved are very similar to those described for channels in section 7, with some extensions:

- sending or receiving an object via a channel, with the additional requirement that the sending and receiving threads must synchronize for the transfer to take place.

- a way of creating multiple parallel threads (for the PAR construct), and synchronising after they have completed.

- a mechanism for doing non-deterministic selection (for the ALT construct).

10.1. Implementation

There were initially five macros in the occam-style extension to EuLisp:

(`PAR` *expression**) Execute each expression in parallel.

(`ALT` {(`IN` *channel var*) *expression**} *) Non-deterministically select and execute an expression whose associated IN channel has a value available, with the value assigned to var. In occam, the guard can be any boolean expression, but since EuLisp has other forms for conditional expression, `ALT` has been restricted to testing for input from channels.

(`IN` *channel* {*var*}) Input a value on the channel, and assign the value to the variable if it is specified, otherwise return the value.

(`OUT` *channel value*) Output a value on the channel. An error is signalled on attempting to output a value on a channel from a thread that is not connected to the channel.

(`SEQ` *expression**) Evaluate each expression in sequence. This only renames `progn`, but is provided for completeness.

The creations of channels and their connection to threads must be done manually in the EuLisp implementation, whereas, in occam, code to construct the process network is generated as part of the compilation process. We plan to address this in a future version of the CSP module. Extensions such as those outlined above are reasonably straightforward (an exercise in macro writing), but the resulting language is limited due to the static nature of occam—it is not intended that it be possible to create an unknown number of threads, or do non-deterministic selection on an arbitrary number of channels. This does not fit well with the dynamic nature of Lisp and the solutions that culture develops. Thus, we indulged in some further extensions to resolve these "shortcomings".

The problem of creating an arbitrary number of processes is addressed by the `FOR` construct, which is similar that in occam, but has no restriction on the types of expressions that can be used as predicates. The issue of doing an `ALT` operation on an arbitrary number of channels is addressed by the `IN-FROM` construct which has the syntax

(`IN-FROM` (*chan-var value-var*) *chan-list expression**)

When an `IN-FROM` statement is executed, a channel from *chan-list* which has input pending is selected non-deterministically, and the expressions are evaluated in the current environment, augmented with *chan-var* bound to the selected channel and *value-var* bound to the value input on the selected channel. These two operations provide more flexibility than the standard set above—indeed, those are implemented in terms of `FOR` and `IN-FROM`.

To illustrate the use of these extensions the appendix shows an implementation of the dining philosophers problem. The major difficulty is specifying the network over which the processes act—as it can be created dynamically one cannot simply compile this information into a startup function, but must make it explicit in all parts of the program that fork threads and create channels. Currently this is done by creating a channel which has input and output ends, forking the new threads, and finally connecting the ends of the channel to the new processes. Note that the code in the appendix the channels are two-way (thus both `IN` and `OUT` operations may be executed on the same channel). Two-way channels are simply a pair of normal channels connected in opposite directions between a pair of processes. The standard connect operations are specialised (they are implemented as generic functions) to make the two connections, so that the user may view the object as a single channel.

11.   Linda

Linda [4] is proposed as a a coordination language suited to a wide variety of architectures, and also is able to make the prototyping of parallel programs easier. While there is a readily identifiable set of core operations (`in`, `out`, `rd`) and data structures (pools and tuples) in the Linda model, there is still much debate about the desirability of non-blocking operations (`inp`, `rdp`) and the semantics of `eval`. For a detailed survey of this area see [3].

The EuLisp language allows one to describe Linda in an object-oriented fashion. In this section we make a case study of producing such a system. The resulting system[1] is described in detail below. In keeping with the lack on agreement on system aspects, we add some of our own extensions,

---

[1] called Ellis

motivated by the fact that TELOS enables generic operations to be defined on the Linda system's classes.

multiple pools: In standard Linda, all operations work on a single pool. It is far more convenient to regard pools as first class, instantiable objects—as many others have done.

generic operations: The key Linda operators (`linda-in`, `linda-out` and `linda-read`) take an additional *pool* argument which may be used to specialize the operation to provide different behaviours.

generic matching primitives: The match operation on tuples in pools is generic, so that tuple lookup can be specialized.

The original model has a potential bottleneck and a serious problem with distributed processing in the use of a single pool[2]. The single pool also creates other problems because it is like the global environment of classical imperative languages. This leads to two phenomena:

unintended aliasing: Any tuple created by one process is visible to every other process—the only thing preventing retrieval of the "wrong" tuple is adherence to some globally consistent naming scheme. This conflicts with the notion of decoupling that Linda encourages. If two sets of tuples do not have unique names the result is *unintended aliasing.*

temporal aliasing: When two components operate as producer and consumer, but the former generates faster than the latter can accept *and* there is an implied ordering on the tuples. One way to resolve this is by enumerating tuples, but this is neither elegant nor general.

Both of these problems can be resolved with the help of multiple pools which provide a mechanism akin to COMMON blocks in FORTRAN. Indeed, all of the extensions listed above help in the production of more modular programs—in systems with single pools ad-hoc techniques are used to ensure that no tuples in different parts of the application can be matched using the same pattern. Also, since no process is able to operate on a pool without access to the pool object, a malicious or incorrect process will not be able to interfere with another's data.

---

[2] Which is not to say that multiple pools are without problems

11.1. Decomposition

In designing the Ellis system, as with any other library designed in an object-oriented style[11], the implementation is split into a number of interacting classes of object. For Linda, the choice of object classes are fairly obvious.

We begin by identifying the objects used in the standard Linda model:

tuples: The objects returned by `linda-in` and `linda-read` operations.

pool: The object which stores the tuples, and processes requests for them.

The `linda-in` and `linda-read` operations require an object which is used to match a tuple or set of tuples. We view patterns and tuples as different entities—without this it becomes difficult to implement some of the extensions described below. It also means that "wildcard values" in tuples are no longer needed. Of course, we also need a class to model Linda processes and some form of scheduling to mediate requests for new threads and pools and to map Linda processes to processors. This is all provided by a scheduler class, which also permits the introduction of a notion of locality. A scheduler object comprises a thread and any number of pools. When (user) Linda processes access these pools, they come under the influence of the scheduler which owns the pools and can be controlled by it. Although a process can access the pools belonging to any scheduler, it is going to be faster to access the pools of the scheduler on the same processor as the accessing process.

Each class is freely instantiable, and may be extended via inheritance and method specialization to support different or more efficient services as desired.

11.2. Implementation

The initial target architecture was a shared memory system which provides subclassable threads and semaphores as part of its kernel. The same system has also been run under virtual shared memory on the KSR-1. While the implementations for shared and virtual shared memory (VSM) can be the same, as we further develop systems on the KSR we expect the lower level details to change because of the higher cost of cache misses and process contention. Despite the simplicity of relying on cache misses to get the right data to the right processor, it seems that high performance will only be achievable if the VSM hardware is bypassed.

The mapping between Linda's threads and the model provided by Eu-Lisp is direct; a Linda thread is simply a subclass of thread with no extra

functionality. The only remaining problem is that of choosing an appropriate data structure for pools, tuples and patterns. For ease of implementation, patterns which have a symbolic key are used. This key is then used as an index into a table of tuples stored by the pool using an `eq` comparison. The tuple is then matched from the resulting list.

11.3. Extensions

Using the model described above, one can also design a system which runs over a network of machines or processors that do not share memory —the only place we required shared memory in the implementation above was for storing the pool data structure. Such a system has been built in Feel[16], using PVM[7] as the communications mechanism for a network of Sun workstations. The actual implementation details are quite simple—the pool structure itself is not distributed, and when a pool is passed to a remote machine (via a remote thread starting mechanism), it is the name of the pool and its location that is sent. If any Linda operations are executed on the (remote) pool, then the name is interpreted as a forwarding address, and a proxy process on the original machine carries the operation, and returns the result. This means that we can reuse much of the above implementation, just adding methods on the new classes of pool and tuple that contact the appropriate server. The scheduler class is redefined as a list of schedulers and a pool. When the creation of a new task is requested, the task is wrapped up as a tuple and placed in the pool. When a processor becomes idle, the associated scheduler executes a `linda-in` operation on the pool to get more work. This approach provides a simple mechanism for load balancing without high overheads.

Matching patterns to tuples in Ellis is a generic operation and, thus, susceptible to specialization. One interesting avenue to explore is the idea of a system which regards tuple matching as an active part of the program, rather than the passive rôle it has in the standard Linda model. With the help of active tuple matching, remote procedure call (RPC) could be modelled by sending the arguments in a pattern, with the results returning as the "matched tuple". This form of communication has the advantage that the receiver and sender need not be aware of each others' locations—a number of different servers could be capable of doing the match operation. Also, by using a suitable distribution function, a requesting process could attach to any of them. There are many other possibilities following from this model that we have yet to explore. What makes this convenient is that the same framework can be used because of the generic nature of the system.

12. Related work

Several languages now have support for concurrency using a threads abstraction.

Sting[10][15] is an efficient parallel dialect of Scheme. Threads provide an interface to an underlying virtual machine (which may be running many virtual processors). A number of optimizations have been made to ensure that the threads are lightweight including thread stealing and reuse of thread control blocks. The EuLisp definition purposely does not state how creation and scheduling of threads is to be achieved and an efficient implementation could employ these techniques.

A distributed interpreter[18] for a Lisp-like language has been implemented in the ICSLA project at INRIA-Rocquencourt. Following a similar philosophy to the concurrency toolbox, it offers a single primitive responsible for thread creation and termination, another responsible for migration of threads to other sites on a network, and an atomic exchange operation. The language is based on Scheme and makes extensive use of first class continuations; in contrast, the EuLisp threads model deliberately restricts the use of continuations. The ICSLA model would itself be a suitable basis for implementing EuLisp threads and the abstractions built from them.

Modula 3 [13] defines a threads package. Like EuLisp it provides procedures to control threads and semaphores to implement mutual exclusion. It also has a mechanism for controlling threads by allowing them to wait on conditions which other threads could signal (either unblocking one thread waiting on the condition or using a broadcast to unblock all threads waiting on the condition). A similar feature could be implemented in EuLisp using the thread primitives, semaphores and `wait`.

13. Conclusions and Future Work

In this paper we have given a more detailed description of the EuLisp thread model than appears in the definition by virtue of examining a number of abstractions built from the basic components. This has the advantage of providing a much better feel for the capabilities of the model, how its definition maps to different architectures and illustrates a "rationale by use" for why the thread model is as it is.

We also believe that the thread model, while useful on its own, is rendered significantly more powerful when taken in conjunction with Telos. Not only are we able to model other common parallel abstractions used in other languages as well as in Lisp, but the user may modify their behaviour via inheritance and specialisation, rather than using a fixed set of

runtime options to determine behaviour. One can also model these abstractions without using the implementation, but re-use the specification and protocol functions, and instead use a home-grown set of classes as the implementation. Programs using the old implementation should then map to the new without change. This should make programs using the abstraction interface more portable.

Our future work plan envisages the expansion of the primitives described here to work over general distributed environments. Both a virtual shared memory and an RPC abstraction for distributed computing are being developed using TELOS.

The moral of the story is that while there is no universal solution to the expression of parallel programs the combination of threads and an object system with a meta-object protocol provides a rich environment for exploring new paradigms.

14.   Acknowledgements

References

1. *Programmer's Utilities and Libraries.* Sun Microsystems, Inc (1990).

2. Bretthauer, H., Davis, H., Kopp, J., and Playford, K. Balancing the EuLISP Metaobject Prototcol. In *Reflection and Metalevel Architecture*, Proc. of the International Workshop on New Models for Software architecture (November 1992) 113–118.

3. Butcher, P.R.A. *Lucinda, General Purpose Programming for Parallel Distributed Systems.* PhD thesis, University of York (1993). in preparation.

4. Carriero, N. and Gelernter, D. Linda in context. *Communications of the ACM*, 32, 4 (1989) 444–458.

5. Cooper, Eric C. and Draves, Richard P. *C Threads.* Technical Report Research report CMU-CS-88-154, Carnegie Mellon University (June 1988).

6. Dijkstra, E.W. Solution of a problem in concurrent programming control. *CACM*, 8, 9 (September 1965) page 569.

7. Geist, G. and Sunderam, V. *Network Based Concurrent Computing on the PVM System*. Oak Ridge National Laboratory (1991).

8. Halstead, Robert H. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7, 4 (October 1985).

9. Hoare, C. A. R. *Communicating Sequential Processes*. *Prentice-Hall International Series in Computer Science*, Prentice-Hall, London (1985).

10. Jagannathan, Suresh and Philbin, Jim. A Foundation for an Efficient Multi-Threaded Scheme System. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming* (June 1992).

11. Kiczales, G. Towards a new model of abstraction in software engineering. In *Reflection and Metalevel Architecture*, Proc. of the International Workshop on New Models for Software architecture (November 1992) 1–11.

12. Mohr, Eric. *Dynamic Partitioning of Parallel Lisp Programs*. PhD thesis, Yale University (1991).

13. Nelson, Greg, editor. *Systems Programming with Moodula-3*. *Series in Innovative technology*, Prentice Hall (1991).

14. Padget, J.A. and Nuyens, G. (Eds.). The EuLisp Definition. (1993). in preparation.

15. Philbin, Jim. Scheduling policy management in Sting. In *Workshop Proceedings on Parallel Symbolic Computing: Languages, Systems and Applications* (October 1992).

16. Playford, K. J. and Broadbery, P. A. *Feel: An Implementation of EuLisp*. Concurrent Processing Research Group, School of Mathematical Sciences, University of Bath (June 1991). May be obtained by anonymous ftp from ftp.bath.ac.uk.

17. Prini, Gianfranco. Explicit parallelism in Lisp-like languages. In *Conference Record of the 1980 Lisp Conference*, The Lisp Conference, P.O. Box 487, Redwood Estates CA 95044, Stanford (California USA) (August 1980) 13–18.

18. Queinnec, Christian and DeRoure, David C. Design of a concurrent and distributed language. In *Parallel Symbolic Computing* (October 1992). To be published in Springer-Verlag Lecture Notes in Computer Science.

19. Rosenberry, W., Kenney, D., and Fisher, G. *Understanding DCE*. O'Reilly & Associates, Sebastapol CA (1992).

20. Rozier, M., Abrossimov, V., Armand, F., Bowle, I., Giln, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Leonard, P., and Neuhauser, W. CHORUS distributed operating systems. *Computing Systems Journal*, 1, 4 (1988).

21. Sabot, G. W. *The Paralation Model: Architecture Independent SIMD Programming*. MIT Press, Cambridge, MA (1988).

22. Weinreb, D. and Moon, D. *The Lisp Machine Manual*. Symbolics Inc. (1980).

23. Zorn, Benjamin, Ho, Kinson, Larus, James, Semenzato, Luigi, and Hilfinger, Paul. Multiprocessing extensions in Spur Lisp. *IEEE Software* (July 1989).

A. Larger examples

A.1. Same Fringe using stack groups

```
(defun same-fringe (tree1 tree2)
  (let ((sg1 (make-stack-group "fringe1" fringe tree1))
        (sg2 (make-stack-group "fringe2" fringe tree2))
        (atom1 nil)
        (atom2 nil))
    (labels
      ((loop ()
        (setq atom1 (stack-group-funcall sg1 nil))
        (setq atom2 (stack-group-funcall sg2 nil))
        (cond ((and (eq atom1 'finished)
                    (eq atom2 'finished)) t)
              ((eq atom1 atom2) (loop))
              (t nil))))
      (loop)))

(defun fringe (tree)
  (fringe1 tree)
  'finished)

(defun fringe1 (tree)
  (cond ((atom tree) (stack-group-return tree))
        (t (fringe1 (car tree))
           (fringe1 (cdr tree)))))
```

A.2. Dining philosphers problem in CSP

```
(defun philosophize (i lchan rchan doorchan)
  (SEQ (enter i doorchan)
       (OUT rchan 'req) (OUT lchan 'req)
       (eat i)
       (OUT rchan 'free) (OUT lchan 'free)
       (leave i doorchan))
  (philosophize i lchan rchan doorchan))

(defun enter (i doorchan)
  (OUT doorchan 'enter)
  (IN doorchan))

(defun leave (i doorchan)
  (OUT doorchan 'leave))

(defun doorman (chans)
  (doorman-aux ready-chans live-chans 0))

(defun doorman-aux (ready live-chans i)
  (IN-FROM
     (chan req)
     live-chans ;;get a request
     (cond
       ((eq req 'enter) ;;enter?
        (cond
          ((= i (- n-phil 1)) ;;n-phil is the number of philosophers
           (doorman-aux chan (delete! chan live-chans) i))
          (t (OUT chan 'ok) ;;no problem...
             (doorman-aux ready live-chans (+ i 1)))))
       ((eq req 'leave)
        (cond
          (ready ;;someone waiting?
           (OUT ready 'ok)
           (doorman-aux nil cons ready live-chans) i))
          (t (doorman-aux ready live-chans (- i 1))))))))

(defun fork-task (lchan rchan)
  (let ((dummy nil)) ;;get grab+then wait for release
    (ALT ((IN lchan dummy) (IN lchan dummy))
         ((IN rchan dummy) (IN rchan dummy))))
  (fork-task lchan rchan))
```

```
;;Initialisation — First construct the channels, then call the
;;main functions with a connected end of the channel.
(defun doit (n)
  (let ((left-channels (map make-Chan-Pair (make-vector n)))
        (right-channels (map make-Chan-Pair (make-vector n)))
        (doorman-chans (map make-Chan-Pair (make-vector n))))
    (PAR
     (FOR (i 0) (< i n) (++ i)
      (SEQ (init-phil i)
           (philosophize i
             (connect-chan-pair (element left-channels i))
             (connect-chan-pair (element right-channels i))
             (connect-chan-pair (element doorman-chans i)))))
     (FOR (i 0) (< i n) (++ i)
      (fork-task (connect-chan-pair (element left-channels i))
                 (connect-chan-pair (element right-channels
                                     (remainder (+ i 1) n)))))
     (doorman (map connect-chan-pair doorman-chans) n)))
```