FEEL: An Implementation of EuLisp
Version 0.92

Concurrent Processing Research Group
School of Mathematical Sciences
University of Bath, United Kingdom
E-mail: eulisp@maths.bath.ac.uk

April 19, 1994

Abstract

This document describes an implementation of EuLisp called Feel. The primary reference for EuLisp is the EuLisp definition. In this document, the environmental operations provided in Feel, but which are not part of the EuLisp language, are described in detail, and examples on the use of some EuLisp features are provided.

Contents

1 Getting Further Information

Information about EuLisp and a copy of the Feel implementation are all available by anonymous ftp from `ftp.bath.ac.uk` in the directory `pub/eulisp`. This document and the implementation of EuLisp are currently the responsibility of Julian Padget (`jap`), Russell Bradford (`rjb`) and Duncan Batey (`djb`) at `maths.bath.ac.uk`.

2 Differences Between FEEL And EuLisp

Inevitably there are a number of minor ways in which Feel is not an accurate implementation of EuLispas described in the EuLisp definition (to which we refer the reader for details of the language itself).

The whole of Level 0 is implemented, but some of the level 0 modules (such as `formatted-io`) do not exist, even though the functions they define are in fact available in the `eulisp0` module. Much of Level 1 is also implemented. A major area of incompatability is in the conditions, some of which are not raised when the definition says they should be.

3 Making FEEL

The kind of Feel system you can make depends on the combined capabilities of your operating system and processor. Feel has been developed in a solely Unix environment and this has considerably warped its view of the world. We have attempted ports to DOS, Windows 3.1 and the Macintosh (ver 7), but these are currently out of date and we do not have access to suitable hardware to revise them. If you are interested, please let us know!

A particular feature of EuLisp, and hence Feel, is support for multiple threads of control. Whether these do actually execute concurrently depends on the host system, but, in principle, it should be possible to develop a program using threads on one system—perhaps a uni-processor simulating concurrency—and later execute the same program on a multi-processor or a distributed processor to achieve the same net result.

Broadly speaking, Feel can be made in any of three main configurations:

Generic Under the "ANY" machine configuration, Feel attempts to be a fully portable ANSI C program. Because there is no reliably portable method of implementing threads in C, the thread operations in this mode are not available and only a serial version of EuLisp remains. This mode is most suitable for getting started quickly and also the most sensible place to begin for porting to new architectures or operating systems. Memory use is minimised which may benifit smaller machines such as PCs or any system where memory is at a premium.

BSD This (badly named) configuration mode requires that a stack switching operation be available for Feel to use. Given this code (typically a few lines of the local assembler), the thread operations become available with the limitation that only one thread is run at a time. This mode allows programs to be written in terms of threads which may later be run in parallel without alteration. This mode is most useful for allowing the development of multi-threaded applications under unsupportive operating systems such as BSD 4.2 or 4.3.

System V This configuration requires that stack switching code be available along with the standard System V shared memory manipulation primitives. Given

these things FEEL becomes a truly parallel multi-threaded system using the following model: on start-up a piece of shared memory is allocated, then FEEL forks as many times as there are physical processors in the host machine (this behaviour may be modified). Each of these forked processes runs the FEEL scheduler, running threads from the pool in the shared heap. Each such thread is run to conclusion—unless it yields control, in which case it will be returned to the pool. More processes may be forked than existing processors to simulate truly parallel operation on uniprocessor systems such as Suns running SunOS 4.1.

4 The FEEL Environment

FEEL uses the shell variable `FEEL_LOAD_PATH` when loading modules. Its value is prepended to a default path consisting of the directory in which FEEL was invoked in and other directories specified when the system was built. The resulting path is read at start-up and converted to a list of strings of information in a processor-defined format concerning the filesystems or disks or directories to be searched when loading modules. Modules are stored in files with the extension `.em`.

When booting from a previously compiled bytecode image, the shell variable `FEEL_BOOT_PATH`, prepended to a similar default path to that for `FEEL_LOAD_PATH`, is used to find the bytecode image. Bytecode images are stored as pairs of files with `.est` and `.ebc` extensions.

When using the bytecode compiler, the shell variable `FEEL_INTF_PATH` is used to find module interface files, and the shell variable `FEEL_OBJS_PATH` used to find compiled modules, before system defaults as above. Interfaces are stored in files with the extension `.i`, and compiled modules in files with the extension `.sc`.

4.1 Getting in and out

FEEL is started by typing `feel`, assuming correct paths and installation. To leave FEEL type CNTL-D, or `!exit` (see later section).

4.2 Start-Up Configuration

FEEL's default starting behaviour may be modified in two ways:

Command Line Arguments FEEL recognises the following:

-heap *n* The size of heap to use (in megabytes if $n < 50$, else bytes). This defaults to 4Mb.

-stack-space *n* Amount of storage to allocate for stacks and static data (in megabytes if $n < 50$, else bytes). This defaults to 1 Mb, but should be more for programs that use threads.

-stack-size *n* The size of the interpreter thread stack (in kilobytes if $n < 1000$, else bytes). This defaults to 96Kb. It should not be necessary to change this unless your program stops with a 'stack overflowing' message. Beware that an infinite non-tail recursion problem may also trigger this message.

-boot *name* Load the bytecode image *name*

-noimage Do not load a bytecode image at all

-compiler Load the compiler bytecode image

-sysv Start Feel in System V configuration (if available)

-procs *n* Start up using *n* processors (works in System V configuration only)

A Configuration File Having first processed its command line arguments, FEEL then looks for a file called `.feelrc` in the `$HOME` directory of the user[1]. If found, the file is read and the expressions within executed as if entered at top-level.

4.3 Interacting with FEEL

When FEEL starts, the top-level module loaded is the `user` module[2]. This imports `eulisp0`, and therefore contains most of the usual lisp functions. Most of the module manipulation functionality is defined in the `root` module. The only operations defined in the `root` module are those for loading modules and entering modules (see below). The top-level prompt provides information about which module is the current focus and the history number of the command, for example:

```
user!0>
```

signifies that the current module focus is `root` and that the command history index of this line is `0`.

```
error:user!1>
```

signifies that the error handler is executing, that the current module focus is `user`, and that the command history index of this line is `1`.

The following operations are always available at top-level, regardless of the current module focus:

`(!>` *name*`)` *Feel special form*

Arguments

name: Name of module to load.

If *name* names a module which has been loaded, then the top-level is changed to be in that module. If a module named *name* is not currently loaded, then FEEL tries to load the module from a file called *name*`.em`. If loaded succesfully, top-level is changed to be in that module.

`(!>>` *name*`)` *Feel special form*

Arguments

name: name of the module to load.

Reload and enter the specified module. This has the side-effect of resetting the exported bindings of the module, such that any importing module will reference the new values.

`!exit` *Feel special form*

Within a handler loop, returns to previous top-level. At top-level, FEEL terminates.

[1] Likely to be elsewhere on non-UNIX systems

[2] This can be changed by setting the environment variable FEEL_START_MODULE

`!n` *Feel special form*

Redo the input sequence number `n`.

`!b` *Feel special form*

`!q` *Feel special form*

Within a handler loop, `!b` prints out a backtrace of function calls and their environments. A simpler backtrace, only containing the function calls, can be printed out by typing `!q`. The `eulisp0` module exports a function, `!B` (not a special form) which may do a better job in some circumstances, but is more prone to infinite loops.

5 The Bytecode Compiler

The FEEL bytecode compiler is implemented as a FEEL application. The code produced is quite respectable, and should give significant improvements over interpreted code.

The compiled code does not do any error checking on car, cdr, vector-ref and similar functions. A later extension will define these functions as generic so that type errors can be detected.

5.1 File Types

Eulisp module files These have a `.em` suffix and contain EULISP source code.

Standard compiled modules These have a `.sc` suffix, and contain position and byte-order independent compiled code.

Interface files These have a `.i` suffix, and contain the interface exported by their module, and information on dependencies, etc.

Bytecode files These have `.ebc` and `.est` sufficies. They hold the raw bytecodes and statics for a group of modules.

Fast load files These have `.fm` extensions, and contain raw bytecodes for a single module.

5.2 Compiling

`feelc [-o` *image*`] [`*.em files*`] [`*.sc files*`] [`*other args*`]`

The FEEL bytecode compiler is started by typing `feelc`, assuming correct paths and installation. `feelc` invokes FEEL once for each module to be compiled, and finally one more time if the results are to be linked into a single bytecode image.

The script is not interactive – it takes only command line arguments. All arguments with a `.em` extension are assumed to be modules to be compiled, all arguments with a `.sc` extension are assumed to be compiled modules to be linked – linking only occurs if the `-o` *image* option is given, where *image* is the name of the resultant bytecode image.

Modules are compiled in the order presented – since the compilation of a module relies upon the existence of an interface file for each module that it imports, directly or indirectly, these modules must be given in the order that they would normally be loaded. For example, if module `top` imports module `middle`, which in turn imports module `bottom`, the compilation of `top` would be specified by typing:

```
feelc bottom.em middle.em top.em.
```

All other arguments are passed on to each invocation of FEEL unaltered. The default heap size when compiling is 10Mb.

Warning – modules are loaded before being compiled, so do not attempt to compile modules that execute non-terminating forms at top-level.

The compiler generates many incomprehensible messages – these can mostly be ignored, unless the process stops prematurely, in which case they may give some clue as to just what went wrong. Be advised that compiling large files, and linking in general, can take a long time ...

5.3 Loading

By default, the FEEL startup script `feel` loads a bytecode image which represents the major part of the implemented EULISP functionality, plus some useful extras, such as the `loops` module. To boot FEEL with a different bytecoded image, `myimage`, type:

```
feel -boot myimage
```

To fast-load a specific compiled module, `mymodule`, type:

```
(!!> mymodule)
```

at top-level (this will not work if FEEL is invoked with `feel -noimage`).

6 The EuLisp Object System

The EULISP object system is called TELOS. Every data item in EULISP is part of the class hierarchy. Simple classes can be defined by `defstruct`, more complex classes with `defclass`. There is no message send primitive in EULISP, instead generic functions are used. TELOS has been designed to offer programmability, efficiency and flexibility. The following subsections attempt to illustrate the kinds of things you can do with it by means of a few examples.

6.1 Generic Functions

You see, it's like this...[3]

Our example in this subsection is an implementation of univariate polynomials with integer coefficients. We start with a polynomial structure: this is a single term, with a reductum that is the rest of the polynomial. A reductum that is an integer marks the end of the polynomial. A term consists of the leading degree and the leading coefficient.

```
(defclass <polynomial> (<number>)
  ((ldeg
    accessor ldeg
    initarg ldeg
    initform 1)
   (lc
    accessor lc
    initarg lc
    initform 1)
   (red
```

[3] to quote Keith

```
    accessor red
    initarg red
    initform 0))
  constructor make-polynomial)
```

We define a method on `equal` so we can check if two polynomials are the same. Notice we do not have to check for the bottoming-out of the recursion on the reducta: the generic nature of `equal` ensures that when we get to the end of a polynomial (and we have an integer as a reductum rather than a polynomial) a different method is called. This relies on the fact that equal-methods for (int, poly) and (poly, int) do not exist: the generic function discriminator chooses the nearest applicable method on `equal`, which in this case is (`object`, `object`). This method returns `()` (as the args cannot be `eq`), which is just what we want.

```
(defmethod equal ((p <polynomial>) (q <polynomial>))
  (and (equal (ldeg p) (ldeg q))
       (equal (lc p) (lc q)) (equal (red p) (red q))))
```

We now need some operations on this new type. If we are trying to add polynomials to integers, we would like some method for converting between integers and polynomials. We use the function `lift-numbers` to do this.

```
(defmethod lift-numbers ((i <integer>) (p <polynomial>))
  <polynomial>)

(defmethod lift-numbers ((p <polynomial>) (i <integer>))
  <polynomial>)

(defmethod (converter <polynomial>) ((x <integer>))
  (make-polynomial 'lc x 'ldeg 0))
```

Now if we call any operation with a polynomial and an integer, the integer is lifted to class polynomial, and the operation proceeds as normal. For two polynomials, the method is easy. A minor wrinkle is when the leading terms cancel: we must take care not to have a leading coefficient of 0.

```
(defmethod binary-plus ((p <polynomial>) (q <polynomial>))
  (cond ((= (ldeg p) (ldeg q))
         (let ((sum (binary-plus (lc p) (lc q))))
           (if (zerop sum) (binary-plus (red p) (red q))
             (make-polynomial 'ldeg (ldeg p) 'lc sum
                              'red (binary-plus (red p) (red q))))))
        ((< (ldeg p) (ldeg q))
         (make-polynomial 'ldeg (ldeg q) 'lc (lc q)
                          'red (binary-plus p (red q))))
        (t (make-polynomial 'ldeg (ldeg p) 'lc (lc p)
                            'red (binary-plus (red p) q)))))

(defmethod binary-difference ...
```

and so on for the other arithmetic operations. Also we would put new methods on `generic-prin` and `generic-write` to print out the values of polynomials using a suitable syntax.

6.2 Classes

Classes in EuLisp are not static items: they can be defined and created dynamically just as any other type in the system. The following example demonstrates this by defining a class whose instances are themselves classes, whose instances are modular numbers. The intermediate classes are parameterised by an integer, which are the bases for the modular rings. This also illustrates the use of metaclasses, which control the structure of classes.

We create a metaclass `<zmodn>` which is the class of the classes `<Zmod3>`, `<Zmod5>`, `<Zmod7>`, etc.

```
(defclass <zmodn-class> (<class>)
  ((n initarg n reader zmodn-class-n))
  metaclass <class>)
```

This will be a direct subclass of `class`, and so will inherit its methods, in particular the ability to create subclasses which are themselves classes. The instances of this class will have a slot named `n`, which will be the modular base.

Now we define a superclass for all of its instances, to place them in their own sub-hierarchy of the class graph. This class has an instance variable `z`, since the instances of its subclasses are the fully instantiated modular numbers.

```
(defclass <zmodn-object> (<number>)
  ((z accessor zmodn-z))
  metaclass <zmodn-class>)
```

The metaclass of the instances of `zmodn-object` is defined to be the class `zmodn-class`. Thus the structure of the instances (the classes `Zmod5`, etc.) is determined by `zmodn-class`.

The constructor for the instances of `zmodn-class` (the metaclass) could be the following:

```
(defun make-zmodn-class (n)
   (make <zmodn-class>
         'direct-superclasses (list <zmodn-object>)
         'name (make-symbol (format nil "<zmod-~a>" n))
         'n n))
```

The `make-instance` requires values for the slots in `zmodn-class`, which include `n` (the slot we defined), and `direct-superclasses`, a slot inherited from `class`.

If you want to avoid creating duplicate `zmodn` classes with the same N, try this definition instead:

```
(defconstant *zmodn-table*
   (make <table> 'comparator = 'hash-function generic-hash))

(defun make-zmodn-class2 (n)
  (or (table-ref *zmodn-table* n)
      (let ((cl (make-zmodn-class n)))
        ((setter table-ref) *zmodn-table* n cl)
        cl)))
```

The function to create the modular objects themselves could be defined as follows:

```
(defun make-modular-number (z n)
   (make-instance (make-zmodn-class2 n) 'z z))
```

Note that this implemenatation guarentees that the number is of the appropriate range:

```
(defmethod initialize ((proto <zmodn-object>) lst)
  (let ((i (call-next-method)))
    ((setter zmodn-z) i
     (remainder (scan-args 'z lst required-argument)
                (zmodn-n i)))
    i))
```

Getting `z` from one of these instances is already defined by the reader on `zmodn-object`. Getting `n` involves going to the class. Making this available from instances means defining the following function:

```
(defgeneric zmodn-n (obj))

(defmethod zmodn-n ((z <zmodn-object>))
  (zmodn-class-n (class-of z)))
```

Next, we want to define some simple arithmetic on modular numbers, for example, addition. However, this only makes sense if we have the same modulus in both of the summands.

```
(defun compatible-moduli (n m) (if (= (zmodn-n n) (zmodn-n m)) t
  (error "incompatible moduli" Internal-Error)))
```

We define a method for addition on `<zmodn-object>`: this will then be inherited by each instance, viz., the actual rings `<zmod3>`, `<zmod5>`, and so on.

```
(defmethod binary-plus ((n1 <zmodn-object>) (n2 <zmodn-object>))
  (when (compatible-moduli n1 n2)
    (make-modular-number (+ (zmodn-z i) (zmodn-z j))
                         (zmodn-n i))))
```

We can add a method to the print function to view numbers prettily

```
(defmethod generic-prin ((n <zmodn-object>) s)
  (format s "~a<mod ~a>" (zmodn-z n) (zmodn-n n)))
```

Finally, some examples of numbers

```
(deflocal zero5 (make-modular-number 0 5))
(deflocal one5 (make-modular-number 1 5))
(deflocal two5 (make-modular-number 2 5))
(deflocal three5 (make-modular-number 3 5))
(deflocal four5 (make-modular-number 4 5))
(deflocal zero3 (make-modular-number 0 3))
(deflocal one3 (make-modular-number 1 3))
(deflocal two3 (make-modular-number 2 3))
```

Now if we try an addition:

```
> (+ two5 four5)
< 1<mod 5>
```

We didn't have to specify a plus method for each modular ring individually: the single definition on the superclass suffices.

Thanks to Harley Davis for help on this section.

6.3 Slot Descriptions

Another aspect of the programmability of TELOS is slot-descriptions. This allows the user to control how the slots of a class are accessed. Here we present an example of the use of slot-descriptions to provide a classed (typed) slot facility. The aim is to be able to define a class and, at the same time, the class of the values to be associated with a given slot. The solution is to define a new kind of slot-description to verify that only values of the correct class are stored in the slot. We start by defining a new kind of slot-description `<classed-local-slot-description>`.

```
(defclass <classed-local-slot-description> (<local-slot-description>)
   ((contents-class
     initform <object>
     initarg contents-class
     reader classed-local-slot-description-contents-class))
   metaclass <slot-description-class>)
```

The classed-local-slot-description class inherits the normal slots from `<local--slot-description>` and adds somewhere to keep track of the allowed class of its contents.

To police the class (type) constraint, we must check that whenever a value is written to a slot with this class—that the value is of the specified kind. we therefore want a new method on `compute-primitive-writer-using-slot-description`.

```
(defmethod compute-primitive-writer-using-slot-description
  ((csd <classed-local-slot-description>) cl lst)
  (let ((std-writer (call-next-method))
    (contents-cl (classed-local-slot-description-contents-class csd)))
    (lambda (obj val)
      (if (subclassp (class-of val) contents-cl)
          (std-writer obj val)
        (error "invalid class of value for slot"
               some-error 'object obj 'sd csd 'val val)))))
```

The call to the standard writer is reached only if the value satisfies the class constraint. It just means the value is acceptable—go ahead and do whatever you normally do to put the slot value inside.

All that remains is how to use one of these slots in a class. The example you give can be done as follows—but remember that `defclass` must be used instead of `defstruct` because the latter does not support user-defined slot classes.

```
(defclass <person> ()
  ((age
    initarg age
    slot-class <classed-local-slot-description>
    slot-initargs ('contents-class <integer>)
    accessor age)
   (name
    slot-class <classed-local-slot-description>
    slot-initargs ('contents-class <string>)
    accessor name)
   (ordinary-slot
    initform 'bleagh)))
```

The slots `age` and `name` are of the new class of slot with their contents class set to `integer` and `string` respectively. Of course, other slots with different classes of

slot description may also be defined.

Now, we may type the following:

```
(setq i (make <person>)) ((setter age) i 27)
```

which is fine and `(age i)` will return `27`.

```
((setter age) i 'not-a-number)
```

but this signals an error. Thanks to Luis Mandel for prompting this example.

6.4 Mixins

FEEL supplies a mixin module[4] to allow the use of mixin classes à la Flavors. A mixin class is a class that can be used in a multiple inheritance network, but has certain restrictions to enable the creation of more efficient accessors—multiple inheritance is restricted to non-instantiable classes and these classes, *mixins* are then used for specialisation of instantiable objects, *base-objects*. Mixins tend to be used to describe attributes of objects, and then these are "mixed in" with base classes to create specialized classes. The mixin implementation has two metaclasses

- `<mixin-class>` The class of a mixin class
- `<mixin-base-class>` The class of a base-object class

Instances of `<mixin-class>` are not instantiable, but allow full MI. Only instances of `<mixin-base-class>` may inherit from mixin-classes, and the list of direct superclasses of a `<mixin-base-class>` must have all mixin-classes before a single non-mixin class (In FEEL, it may inherit from any other class in the system, including `<class>`).

Note on the implementation: `<mixin-class>` has a different default slot type, `<mixin-slot-description>`. When this slot is inherited *directly* by a `<mixin-base-class>` its the accessor is computed. If the slot is not newly created, however, no new access method is computed, therefore reducing the number of such methods for a given accessor.

6.5 Eql Methods

These are supplied by the `eql` module, which effectively defines new generic function and method classes, and allows further `eql` methods to be defined. See `eql.em` for full details.

7 The PVM Module

The pvm module provides an interface to the pvm library. This section assumes that the reader has read at least some of the PVM documentation.

The pvm module differs from the pvm library in the following ways:

- Arbitrary lisp expressions (including circular structures) may be sent from machine to machine
- The format in which objects are sent is not the XDR format used by pvm, but an internal format. It is hopefully machine (and byte order) independent. See the section on the reader module for more details.

[4] called `mixins`

```
(defclass <point> ()
  ((x initform 0 accessor point-x initarg x)
   (y initform 0 accessor point-y initarg y))
  )

(defclass <colored> ()
  ((color initform 'black initarg color
          reader color))
  metaclass <mixin-class>)

(defgeneric color-of (obj)
  method (((obj <object>)) 'gray)
  method (((obj <colored>))
          (color obj)))

(defclass <colored-point> (<colored> <point>)
  ()
  metaclass <mixin-base-class>)

(setq p1 (make <point>))
(color-of p1)

(setq p2 (make <colored-point> 'x 1 'y 1 'color 'red))
(color-of p2)
```

Figure 1: Usage of mixin inhertance

- Several reads may occur simultaneously on separate threads [5]. In other words, it is possible to call thread-suspend during a read.

The pvm module exports the following functions:

`(pvm-enroll)` *PVM*

Arguments

name: A string

Result

Enrolls Feel into PVM under the given name. Must be called before any other PVM function. Return the pvm-id of the enrolled process (actually a cons cell whose car is the name provided as argument to pvm-enroll, and whose cdr is an instance number allocated by PVM).

`(pvm-leave)` *PVM*

Result

Exits from PVM control. After this is called, all PVM functions return an error message (except pvm-enroll).

[5] note that pvm is not yet interfaced to the System V version.

`(pvm-whoami)` *PVM*

Result

Returns the pvm-id (the value returned by enroll) of the process.

`(pvm-status` *id*`)` *PVM*

Arguments

id: Identifier of a pvm-process

Result

Returns non-nil if the process with pvm-id *id* is running, otherwise nil.

`(pvm-send` *dest type msg* `[`*reader*`])` *PVM*

Arguments

dest: A PVM process identifier

type: The type of the message (integer)

msg: The message (can be anything)

`[`*reader*`]`: A reader which is used to write the message.

Result

Sends a message of type *type* to the process specified by the id *dest* containing the value *msg*. If a reader is specified it is used to handle any complex lisp types inside the message (see section 9).

`(pvm-recv` *type info?* `[`*reader*`])` *PVM*

Arguments

type: The type of message to be recieved (integer)

info?: Is information on message wanted (boolean)

`[`*reader*`]`: A reader which is used to read the message.

Result

Blocks until a message of type *type* is recieved. If *info?* is nil, then the message is returned. If *info?* is non-nil, a list is returned in the following format: (*msg type from*) where msg is the message, type is the type and from is the process-id of the sending processes. Note that this only blocks the executing thread – ready threads will automatically be scheduled.

`(pvm-recv-multi` *type-list info?* `[`*reader*`])` *PVM*

Arguments

type-list: A list of possible message types (integers)

info?: Information on message wanted flag (t or nil)

`[`*reader*`]`: A reader which is used to read the message.

Result

As pvm-recv, but blocks the executing thread until a message which has a type in the type-list is received.

`(pvm-initiate-by-type` *type name*`)` *PVM*

Arguments

type: Type of machine (string)

name: Name of the new process (string)

Result

Runs the executable *name* on a machine of type *type*, and return the PVM identifier of the new process.

`(pvm-initiate-by-hostname` *hostname name*`)` *PVM*

Arguments

hostname: Hostname in which to start process

name: Name of the new process

Result

Runs the executable *name* on the machine *hostname*, and return the PVM identifier of the new process.

`(pvm-probe` *type*`)` *PVM*

Arguments

type: A message type (integer).

Result

Tests for messages of a given type. Returns the type, or nil if no message of that type is in the input queue.

`(pvm-probe-multi` *type-list*`)` *PVM*

Arguments

type-list: A list of message types (integers).

Result

Tests for messages from a list of types.

Other functions provided are

- pvm-barrier
- pvm-ready
- pvm-waituntil

- pvm-terminate

These last are mostly untested, but provide analogous functionality to their PVM equivalents.

This module is based on PVM version 2.3 – the next section describes a module based on PVM version 3.2, which has a substantially different interface.

8 The PVM3 Module

Like the pvm module, the pvm3 module is not completely analogous to the pvm3 library. The differences should be fairly intuitive, and are mainly to make the module more useable.

The pvm3 module exports the following functions:

`(pvm-mytid)` *PVM3*

Enrolls Feel in PVM if it is not already enrolled, and returns the tid (task identifier – a large integer) in either case.

`(pvm-exit)` *PVM3*

Exits from PVM control

`(pvm-config)` *PVM3*

Returns a list of hostname, architecture type pairs, describing the current PVM configuration.

`(pvm-spawn-on-host` *name host* `[`*n*`])` *PVM3*

Arguments

name: name of executable

host: name of the host

`[`*n*`]`: number of instances

Result

Runs *n* instances (1 by default) of the executable *name* on the machine specified by *host*, and returns a list of the tids of the new processes.

`(pvm-spawn-on-arch` *name arch* `[`*n*`])` *PVM3*

Arguments

name: name of executable

arch: architecture type

`[`*n*`]`: number of instances

Result

Runs *n* instances (1 by default) of the executable *name* on a machine of type *arch*, and returns a list of the tids of the new processes.

`(pvm-tasks)` *PVM3*

Returns a list of the tids of all processes currently enrolled in PVM.

`(pvm-pstat` *tid*`)` *PVM3*

Returns `t` if the process with identifier *tid* is alive, and `nil` otherwise.

`(pvm-kill` *tid*`)` *PVM3*

Terminates the process with identifier *tid*.

`(pvm-send` *tid type message* `[`*reader*`])` *PVM3*

Arguments

tid: A PVM task identifier

type: The type of the message (integer)

message: The message (can be anything)

`[`*reader*`]`: A reader used to write the message

Result

Sends a message of type *type* to the process specified by *tid* containing the value *message*. If a reader is specified it is used to handle any complex lisp types inside the message (see section 9).

`(pvm-mcast` *tid-list type message* `[`*reader*`])` *PVM3*

Arguments

tid-list: A list of PVM task identifiers

type: The type of the message (integer)

message: The message (can be anything)

`[`*reader*`]`: A reader used to write the message

Result

Sends a message of type *type* to each of the processes specified in *tid-list* containing the value *message*. If a reader is specified it is used to handle any complex lisp types inside the message (see section 9).

`(pvm-bcast` *type message* `[`*reader*`])` *PVM3*

Arguments

type: The type of the message (integer)

message: The message (can be anything)

`[`*reader*`]`: A reader used to write the message

Result

Sends a message of type *type* to all enrolled processes containing the value *message*. If a reader is specified it is used to handle any complex lisp types inside the message (see section 9).

`(pvm-receive` *tid type info?* `[`*reader*`])` *PVM3*

Arguments

tid: A PVM task identifier (integer: -1 is wild card)

type: The type of the message (integer: -1 is wild card)

info?: Is information on message wanted (boolean)

`[`*reader*`]`: A reader used to read the message

Result

Blocks the executing thread until a message of type *type* is recieved from the process specified by *tid*. If *info?* is nil, then the message is returned. If *info?* is non-nil, a list is returned in the following format: (*msg type from*) where msg is the message, type is the type and from is the process-id of the sending processes.

`(pvm-probe` *tid type*) *PVM3*

Arguments

tid: A PVM task identifier (integer: -1 is wild card)

type: The type of the message (integer: -1 is wild card)

Result

Returns `t` if a message matching *tid* and *type* is ready to be received, otherwise `nil`.

`(pvm-joingroup` *group*) *PVM3*

Arguments

group: A string used to identify the group

Result

Creates a group called *group* if one doesn't already exist, and joins the group in either case.

`(pvm-lvgroup` *group*) *PVM3*

Arguments

group: A string used to identify the group

Result

Leaves the group called *group*.

`(pvm-gtasks` *group*`)` *PVM3*

Arguments

group: A string used to identify the group

Result

Returns a list of the tids of all processes in the group identified by *group*.

`(pvm-barrier` *group count*`)` *PVM3*

Arguments

group: A string used to identify the group

count: Number of processes involved in the barrier

Result

Blocks the executing thread until *count* members of *group* have called `pvm-barrier`.

9 The Reader Module

The reader module provides functions to read and write lisp forms as bytevectors. It is intended to be reasonably machine independent, although at the current time it falls a little short. The module currently deals with reading and writing lisp forms for the pvm module.

The reader in its default form can read any 'simple' lisp expression that is: integers, floats, strings, symbols[6], lists and vectors. The extensibility is provided via an extra argument which may be supplied to control the reader's behaviour on complex lisp types. A type here means a group of classes which can be read in the same way. The type of an object is given by the integer identifier passed to `add-writer` and `add-reader`.

`(make-obj-reader)` *Reader*

Result

Makes a new reader object. The class and internals of this object are left unspecified.

`(add-writer` *reader class type-ident function*`)` *Reader*

Arguments

reader: A reader

class: A class

type-ident: An identifier (> 16)

function: A function to be called when an object of class `class` is encountered.

[6] Support for symbols may be removed in future versions because they may require some caching, which will be provided by a lisp level

Result

Adds a new writer function, *function* to the given reader. The function is called when an object of class *class* (or one of its subclasses) is encountered by a write process. It is called with three arguments: the object to be written, a value representing write buffer and the reader which called the function. The function should call `write-next` with any data associated with the object.

(`add-reader` *reader type-ident function*) *Reader*

Arguments

reader: A reader

type-ident: An identifier

function: A function.

Result

Adds a new reader function `function` to the given reader. The function is called whenever an object of type `type-ident` is encountered by a read process. It is called with two arguments: a value representing the read buffer plus the reader supplied by the caller of the read. The function then calls `read-next` to obtain any data associated with the object. If the function fails to consume all the data written by its corresponding write, an unhandled error condition results[7].

(`read-next` *ptr reader*) *Reader*

Arguments

ptr: A pointer value

reader: A reader

Result

Returns the next object in the read-buffer specified by *ptr*, using *reader* as the reader object. This functions can only be called inside the dynamic scope of a read function.

(`write-next` *object ptr reader*) *Reader*

Arguments

object: the object to be written

ptr: A pointer value

reader: A reader

Result

Writes the object *object* onto the write-buffer specified by ptr, using *reader* as the reader object.

[7] FEEL goes kaboom

9.1 Example

```
;;define a structure which we want to pass around

(defstruct example-cons-pair ()
  ((car initarg car reader example-car)
   (cdr initarg cdr reader example-cdr))
  constructor (example-cons car cdr))

;;invent a number — this *must* be more than 16
(defconstant *example-type-id* 18)

;;make a reader

(defconstant *the-reader* (make-obj-reader))

;;define readers and writers for example-cons

;;note that both these functions *can* side effect, so circular
;;structures and caching can be handled (using tables or similar), also that the
;;particular reader can be changed for the recursive call
;;to the reader (although I do neither here).

(defun write-example-cons (obj ptr rdr)
  ;;easy really. Just write whats inside.
  (write-next (example-car obj) ptr rdr)
  (write-next (example-cdr obj) ptr rdr))

(defun read-example-cons (ptr rdr)
  ;;read the internals
  (let* ((a-car (read-next ptr rdr))
         (a-cdr (read-next ptr rdr)))
    ;;construct the appropriate object
    (example-cons a-car a-cdr)))

;;add them to the reader structure

(add-reader *the-reader*
            *example-type-id*
            read-example-cons)
(add-writer *the-reader*
            example-cons-pair
            *example-type-id*
            write-example-cons)

;;we can add more types later...

;;should make
;;(pvm-send (pvm-whoami) 102
;;(example-cons (example-cons 1 2)
;;(example-cons 3 4))
;;*the-reader*)
;;work ok.

;;to receive, (pvm-recv 102 nil *the-reader*)
```

10 Thread Abstractions

EuLisp provides a set of primitive operations for thread creation and manipulation, but for most work these are too low-level and require the user to be overly concerned with their management. One of the design goals of EuLisp was to provide an experimentation environment for parallel processing, and as a result of this several thread abstractions have been built on the EuLisp thread primitives. These abstractions include: *futures*, *linda*, *timewarp* and *CSP*. The next subsections describe the first two in detail.

10.1 Futures

The nature of the EuLisp thread mechanism means that it lends itself quite naturally to providing a base for the implementation of a simple future abstraction. The acts of creating futures and of eventually interrogating them for their values map almost directly onto starting threads and accessing thread results.

The code for basic future manipulation is given below. A couple of examples of replacements for "strict" functions that allow for future objects are shown. The extensibility of generic functions and module renaming can be used to make these necessary changes transparent for users.

The following are the main functions of interest in the `futures` module ...

`(future` *expression**`)` *Future macro*

Arguments

expression: The expression to be future'd

Constructs a future object and spawns a thread to calculate the value of *expression*. An object of class *future* is returned by the expression resulting from the macro expansion. The implementation of `future` in Feel is essentially:

```
(defmacro future exp
  `(let
     ((future (make-future-object))
      (task (make-thread
              (lambda (future fun)
                ((setter future-object-value) future (fun))
                ((setter future-object-done) future t)
                t))))
       ((setter future-object-thread) future task)
       ((setter future-object-function) future (lambda () ,@exp))
       (thread-start task future (lambda () ,@exp))
       future))
```

`(futurep` *obj*`)` *Future generic*

Arguments

obj: The object to be tested

Result

nil if obj is not a future, otherwise, non-nil.

Arguments

future: The value to be evaluated

Result

Forces the evaluation of a *future* and if the result of the evaluation is also a *future* that too is forced until the result is not a *future*.

10.2 Linda

The `eulinda` module implements a simplistic version of the Linda model. Points of note include

1. multiple pools, so the in/out/read functions take an extra argument
2. matching is always literal on first element of tuple
3. function names are prefixed by `linda-`

The module exports the following functions:

- `(make-linda-pool)` to make a pool.
- `(linda-out pool tag val1 val2 ...)` places the values in the pool under the tag.
- `(linda-in pool tag pat1 pat2 ...)` does an "in" from the pool under the tag, using the patterns.
- `(linda-read pool tag pat1 pat2 ...)` does a "read".
- `(linda-eval fun arg1 arg2 ...)` starts a new thread running the function with the arguments.

A pattern is

- `?` matches any value.
- `(? var)` matches any value, and sets the *var* to that value.
- anything else is matched literally.

Tags are also *always* matched literally. Thus, if `y` has value 22, then
`(linda-in pool 'foo ? (? x) (+ 1 y))`
searches for a tuple in the pool under the tag `foo` that has anything in the 1st or 2nd slot, and 23 in the 3rd slot. When such a tuple is found, `x` is set to the value in the 2nd slot; the call suspends (with an implicit `thread-reschedule`) until then.

A small example:

```
(defmodule pc (standard eulinda) ()

  (deflocal pc-pool (make-linda-pool))

  (defun producer (tag val)
    (format t "producer: out ~a to ~a~%" val tag)
    (linda-out pc-pool tag val))
```

```
  (defun consumer (tag)
    (let ((x ()))
      (linda-in pc-pool tag (? x))
      (format t "consumer ~a: got ~a~%" tag x))
    (consumer tag))

  (linda-eval consumer 'one)
  (linda-eval consumer 'two)

)
```

Then, e.g., `(producer 'one 23)` will place a tuple in the pool for the thread `one` to fetch out.

Minor debugging tools are `(tril t)` to turn on a trace of the internals of the matching process; `(tril ())` to turn it off. Also `(print-linda-pool p)` will print the values in the pool `p`.