LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1993

EuLisp in Education

RUSSELL BRADFORD (*rjb@maths.bath.ac.uk*)
University of Bath, UK
DAVID DE ROURE (*D.C.DeRoure@soton.ac.uk*)
University of Southampton, UK

Keywords: Education, Lisp, object-oriented programming, concurrent processing, language design

Abstract. We present our experience with EuLisp as a teaching language, focussing on the level of the language which was specifically designed for this purpose (level-0). EuLisp has been used in undergraduate and postgraduate teaching since 1990, in lectures and laboratories, where in many cases it has replaced Scheme or Common Lisp. It has been used extensively in programming courses, parallelism courses, as a vehicle for advanced courses in symbolic computing and programming language design; it has also been used as a platform for final year undergraduate projects. This experience has demonstrated that EuLisp is well suited to teaching and far reaching in its capabilities: it supports the relevant concepts in a consistent and versatile framework, so that the language serves to facilitate the educational process. The discussion is illustrated with examples, and where appropriate we draw a comparison with the Lisp dialects used previously in these courses.

1. Introduction

The EuLisp language design process has been progressing for several years, and during this time the reference implementation developed at the University of Bath, UK, has tracked the evolving definition [4]. The availability of the interpreter has enabled the language to be evaluated as a platform for teaching, as well as for research and applications development; this experience has then been fed back into the design process.

Lisp frequently features in undergraduate courses, often as a vehicle for teaching other material. The authors have been involved in many such courses. Prior to the availability of a EuLisp interpreter, several Lisp dialects had been used; these included Standard Lisp, LISP/VM, Scheme and Common Lisp. Level-0 of EuLisp was specifically designed with a view to teaching, and was adopted experimentally by the authors when an implementation became available in 1990. This level is roughly comparable to Scheme with the addition of an integrated object system (single

inheritance),[1] modules and support for concurrency.

EuLisp has now been used across a spectrum of courses in four institutions, including courses on programming, parallelism, symbolic computing, programming language design and artificial intelligence. This paper presents our experience of using EuLisp in teaching, and compares it with our experience of Scheme [10] and Common Lisp [12]. Section 2 describes a 'programming paradigms' course which was originally based on Scheme, section 3 presents a course on parallelism using the 'concurrency toolbox' of EuLisp, section 4 describes a programming language design course; other courses are discussed in section 5. We conclude with some comments on implementations in section 6 and some final remarks.

2. Programming

In a course designed to introduce a variety of programming paradigms, it is convenient to use a single language which provides natural support for the various styles; it can be argued that nearly any language could be used, but clearly some languages are more suitable than others and will facilitate rather than impede the educational process. Scheme is chosen for this role in the classic text *Structure and Interpretation of Computer Programs* [1], which provides an excellent introduction to programming in models supporting referential transparency, data abstraction, mutable state, message passing and streams, as well as issues in language design and implementation.

EuLisp has also been used in this role, supporting a programming course which follows up an introductory programme based on Standard ML. The primary objective of the course is to introduce techniques for managing the complexity of programs which are larger in size than the introductory examples, reflecting the typical programming task of the individual programmer. An adjacent course addresses methods of large-scale software design and engineering.

The course was originally based on [1] but has evolved to accommodate the addition of new items to the syllabus. The major addition is object-oriented programming, while the other significant changes result from a 'spiral approach' to teaching programming techniques for window systems and for distributed systems; we believe these classes of problems to be sufficiently important to warrant first-class status in the core syllabus for event-driven programming and programming with threads. This philosophy leads to a further, more pervasive, change because concurrency issues are

[1] One of the authors has been heard to call it `Scheme++-`, because it is `Scheme++` without full continuations.

now discussed throughout the course.

The current course syllabus is:

1. Programming in a functional style;
2. Handling state and changes to state;
3. Object-oriented programming;
4. Stream-oriented programming;
5. Event-driven programming;
6. Programming with threads.

The course makes full use of EuLisp's features at level-0 of the language definition. To address software systems of significant complexity, the module system is used to construct the components of an evolving system; this approach enables existing components to be provided to the students, who can then create their own modules to integrate with these. Ultimately, the software consists of the modules provided plus those programmed by the students using the different techniques introduced during the course, forming a relatively sophisticated software system. These techniques are discussed in turn in the following sections.

2.1. Programming without side effects

Since the students have already worked in SML, this section of the course doubles as an introduction to the syntax of EuLisp and the use of the EuLisp environment. Initially all data values are numbers, and the concepts of generic functions and multimethods are introduced. This is then extended to user-defined classes (structures) and inheritance.

The typical example for this material is the introduction of a new class which can be handled by generic arithmetic functions, for example rational numbers (which we call *fractions* to distinguish from the existing rational number class). One module provides the implementation of fractions, exporting the constructor and accessor functions for the `<fraction>` class; alternative implementations are possible, where the fraction may be reduced to its lowest terms at construction time or at access time. This module is imported by a second *fraction arithmetic* module which provides methods on the addition, subtraction, multiplication, division, equality and comparison operations (see Figure 1). An application requiring fractions can then simply import the fraction arithmetic module.

The use of a **Lisp**$_1$ after a course based on SML provides a natural progression, hence EuLisp and Scheme are attractive choices. The strength

```
(defmodule fraction-arithmetic
  (import (fraction-implementation1 eulisp-level-0)
   syntax (eulisp-level-0))

  (defmethod = ((a <fraction>) (b <fraction>))
     (and (= (num a) (num b)) (= (denom a) (denom b))))

  (defmethod binary* ((a <fraction>) (b <fraction>))
     (make-fraction (* (num a) (num b)) (* (denom a) (denom b))))
)
```

Figure 1: Part of the fraction arithmetic module

of EuLisp in this context is the provision of a data abstraction mechanism whereby user-defined 'types' (classes) are identical in use to those provided by the system; this simplicity is not evident in Scheme or CLOS.

2.2. Introducing state

The slots in the structures introduced above are immutable: either objects are created with the slots initialised, or the constructor function assigns values to the slots—in both cases the slot values do not change once set. To provide the facility to modify their values the sole function that need be introduced is `setter` (this function is also used to update structures based on cons cells). Figure 2 illustrates part of the classic bank account program, where an account is created by `(make <account>)` and has an initial balance of zero; `deposit` increases the balance, returning the new value.

There is a particular elegance to the Scheme solution, where closures provide a primitive mechanism for capturing state and controlling access to it; this style of solution is viable in EuLisp too, but for simplicity and consistency we adopt objects as the sole repositories of state—the object-system is seen as the mechanism for handling the intellectual complexity of state in our programs.

2.3. Object-oriented programming

At this stage the major language features to support object-oriented programming have already been introduced. To establish the concepts, the inheritance mechanism is discussed (this is single inheritance at level-0) and examples making more sophisticated use of inheritance are presented.

```
(defmodule bank-account
  (import (eulisp-level-0) syntax (eulisp-level-0))

  (defstruct <account> () ((balance accessor balance initform 0)))

  (defgeneric deposit ((ac <account>) (x <number>)))

  (defmethod deposit ((ac <account>) x)
    (let ((new-balance (+ (balance ac) x)))
         ((setter balance) ac new-balance)
         new-balance))
)
```

Figure 2: Part of the bank account module

Figure 3 illustrates part of an example based on a blocks world, with an abstract class `<block>` and methods added to `generic-write` to illustrate class precedence.

```
  (defstruct <block> () ((colour)))              ;abstract class

  (defstruct <sphere> <block>
    ((diameter accessor sphere-diameter)))

  (defstruct <cube> <block> ((side accessor cube-side)))

  (defgeneric volume ((b <block>)))
  (defmethod volume ((b <cube>)) (expt (cube-side b) 3))

  (defmethod generic-write ((b <block>) s)
    (print 'block s))

  (defmethod generic-write ((b <cube>) s)
    (print 'cube s)
    (call-next-method))
```

Figure 3: Part of the blocks module

Message-passing as a paradigm is also discussed here, where it can be described in terms of messages between real-world objects. A correspondence between messages and generic function calls is established, and this would be an appropriate point to introduce closures as repositories of state, and the `setq` special form to mutate bindings; these techniques can be used to control access to shared bindings, providing a mechanism for creating

contours within the module boundary.

2.4. Stream-oriented programming

The powerful combination of standard interfaces and a flexible means of composition of components is exemplified by UNIX.[2] The illusion of individual data items flowing along channels between black boxes (c.f., UNIX pipes) can be implemented using lists, or using streams as in Scheme (where the second argument to `cons` is delayed and the evaluation of stream elements is memoized). It is possible to write generic code which will work with various implementations of streams; Figure 4 illustrates part of the definition of generic `head` and `tail` functions.

```
(defstruct <stream> () ())

(defstruct <empty-stream> <stream> ())

(defstruct <stream-pair> <stream>
  ((head accessor stream-head)
   (tail accessor stream-tail)))

(defgeneric head (s))
(defmethod head ((s <list>)) (car s))
(defmethod head ((s <stream-pair>)) (stream-head s))

(defgeneric tail (s))
(defmethod tail ((s <list>)) (cdr s))
(defmethod tail ((s <stream-pair>)) (force (stream-tail s)))
```

Figure 4: Part of a generic streams implementation

The implementation of the dataflow illusion using lists is flawed in that it requires the stream to be finite in length and no data appears at the output until all the input data has arrived. Delayed evaluation is demand-driven in nature and overcomes these deficiencies, accommodating infinite streams (and streams which may contain elements which cannot be computed); however, this implementation is still problematic when interfacing to real data streams. Therefore actual input–output streams are also introduced in this section; we adopt the EULISP I/O model which supports user-defined streams, and enables stream processing elements to be plugged together in an arbitrary manner to construct more complex streams.

The multiple inheritance mechanism of CLOS provides a powerful means

[2] UNIX is a trademark of Unix Systems Laboratories, in the USA and other countries.

of constructing a versatile input–output streams system [8]. At EuLisp level-0 we are restricted to single inheritance. We can use this to simulate multiple inheritance by performing a second generic-function-call on prototypical stream properties stored in the slots of stream objects. This serves as a good example of single versus multiple inheritance.

2.5. Handling exceptions

Some problems contain 'exceptional' situations and it is appropriate that the programmer writes code which reflects this, using EuLisp's condition system with user-defined conditions; in these programs, the programmer will arrange both to signal and handle the conditions. In general, the program may also handle conditions raised by the EuLisp system itself in response to the occurrence of exceptional situations. In the absence of user-defined handlers, a default handler will take appropriate action.

Figure 5 shows a simple handler for a *read-eval-print* loop in an embedded interpreter. The application of the user's `generic-eval` function (see section 4) to the input expression occurs within the dynamic scope of the enclosing `with-handler` form; should any conditions be raised during that application, the handler accepts them and performs a non-local exit from the enclosing `block` form. The other options for the handler would be to call the `resume` continuation, or simply to return and thus decline to handle the condition.

```
(defun rep (v)
  (format t "~a~%> " v)
  (rep (block k
         (with-handler
            (lambda (condition resume) (return-from k 'error))
            (generic-eval (read (standard-input-stream))
                          global-environment)))))
```

Figure 5: *Read-eval-print* loop with trivial error handler

This style of condition handling is not provided in Scheme, but can be supported.

2.6. Event-driven programming

Use of the condition system introduces the notions of dynamic extent and the single thread of control being interrupted by exceptional events occurring asynchronously. Building on these ideas, we can introduce the idea of a program whose normal mode of operation is to respond to asynchronous

events in its environment.

To do this, we introduce the `wait` construct, which provides a generic interface to blocking operations. `wait` is provided as a standard abstraction for blocking on asynchronous events; the user can add methods for any objects, though the EuLisp definition only specifies `wait` methods for streams and threads. Given an input stream as argument, `wait` returns true as soon as data becomes available on that stream (i.e., an input operation will not block), else if the timeout period expires it returns false. Figure 6 illustrates the use of `wait` in an event loop, where the object system can be used to control the relationships between different types of events.

```
(while (wait stream timeout)
  (generic-event-handler (read-event stream)))

(defmethod generic-event-handler ((k <keypress-event>)) ...)
```

Figure 6: Fragment of an event-driven program

This construct can support non-deterministic input–output operations when provided with a *collection* of streams; however, collections are not supported in level-0 of EuLisp.

2.7. Threads

The final paradigm is programming with multiple threads of control. This is a natural extension of the previous material, where the idea of external events now includes events associated with other threads executing asynchronously. The use of EuLisp to teach parallelism is discussed in the following section.

Neither Scheme nor Common Lisp support multiple threads of control as part of the standard languages; however, the EuLisp model can be simulated in Scheme, and Common Lisp implementations often provide appropriate facilities (such as stack groups).

3. Parallelism

EuLisp has been used successfully as a component of both undergraduate and postgraduate courses on parallel processing. A course on parallelism should ideally contain some practical exercises—actually trying to program in parallel is the best way of getting across to the student the difficulties inherent in the subject. However, not many classes have access to a true parallel machine, thus we must simulate the process in some fashion by

scheduling threads on a single processor. The threads mechanism in EuLisp is designed so that the user is unaware how many actual processors are running, or how the threads are scheduled on the available processors (some weak promises are made in the definition, but also see [6]). So the user must write code that has no such built-in assumptions, encouraging portability of code, and flexibility of programming style.

In using EuLisp we have the opportunity to program using threads and binary semaphores (named *locks*). From these low-level constructs student assignments are to implement higher level abstractions (see [2] for examples). To make a counting semaphore from a binary semaphore involves the student recognising that there are several issues at stake: the mutual exclusion on the counter; the problem of how and when to suspend a process that fails to decrement the counter; the question of non-determinism in reawakening of processes, and so on.

Students must program without using the knowledge of the particular system the code may run on: it may be a multiprocessor or a uniprocessor, and the latter may be using run-to-completion or time-slicing to schedule threads. This ensures that they fully appreciate the more subtle problems that may arise, particularly due to interleaving of control. For instance, does it make any difference if the two starred lines in the code for the counting semaphore (Figure 7) are swapped? To determine the answer requires some detailed thought.

Building on this, it is a natural step to use counting semaphores (packaged as a module) to implement a bounded buffer producer–consumer system, or as a way to break the deadlock in the Dining Philosophers problem. The student soon discovers that using bare semaphores is not a very convenient or easy way to program, and that a higher-level way of programming (for example, using monitors) is much better, as well as being less error-prone.

After having written the Dining Philosophers problem using semaphores (Figure 8) the student can consider rewriting the code in terms of, say, monitors, at which point the student can appreciate the distinction between synchronisation and mutual exclusion (these are well separated in a monitor, but are generally implemented in identical ways when using just semaphores, leading to confusion of the issue at stake). Also, using monitors, the student can see their relative grain of control in comparison with semaphores: the trivial solution of Dining Philosophers using one or two large monitors does not work, or at best over-constrains the actions of the philosophers (typically, such attempted solutions have a philosopher being stopped from picking up a fork if *any* other philosopher is trying to get a fork). The fine-grained nature easily available from a semaphore must be balanced against the clarity of use of a monitor. This, again, illustrates

```
(defmodule csem (import (eulisp-level-0) syntax (eulisp-level-0))

  (defstruct csem ()
    ((sem initform (make-lock) ;access to the csem
          reader csem-sem)
     (count initform 0 initarg count ;the count
            accessor csem-count)
     (suspend-sem initform (lock (make-lock)) ;block on this
                  reader csem-suspend-sem)
     (suspend-count initform 0 ;number waiting
                    accessor csem-suspend-count))
    constructor (make-csem count))

  (defun cwait (csem)
    (let ((sem (csem-sem csem)))
      (lock sem)                ;get control of the csem
      (let ((s (csem-count csem)))
        (cond ((> s 0)          ;room to move
               ((setter csem-count) csem (- s 1))
               (unlock sem))    ;release the csem
              (t                ;must suspend yourself
               ((setter csem-suspend-count) csem ;one more
                (+ (csem-suspend-count csem) 1))
               (unlock sem)     ;release counting sem
               (lock (csem-suspend-sem csem)))))))) ;suspend yourself

  (defun csignal (csem)
    (let ((sem (csem-sem csem)))
      (lock sem)                ;get the counting semaphore
      (cond ((> (csem-suspend-count csem) 0) ;someone is waiting
             ((setter csem-suspend-count) csem ;one less now
              (- (csem-suspend-count csem) 1))
             (unlock (csem-suspend-sem csem)) ;unblock someone *
             (unlock sem))      ;release the csem           *
            (t                  ;no-one waiting
             ((setter csem-count) csem
              (+ (csem-count csem) 1))
             (unlock sem)))))   ;release the csem

  (export make-csem cwait csignal)

)
```

Figure 7: Counting Semaphores using Binary Semaphores

```
(defmodule dinphilsem (import (eulisp-level-0 csem)
                       syntax (eulisp-level-0))

  (defconstant no_phils 5)
  (defconstant count 20)  ;each phil to eat 20 times

  (defun phil (p n)       ;phil p eating for the nth time
    (let ((left p) (right (remainder (+ p 1) no_phils)))
      (when (< n count)
        (think p)
        (get-ticket p) (get-fork left p) (get-fork right p)
        (eat p n)
        (drop-fork left p) (drop-fork right p) (return-ticket p)
        (phil p (+ n 1)))))

  (defconstant room-ticket (make-csem (- no_phils 1)))
  (defconstant forks (make-vector no_phils))

  (defun init-forks (n)
    (when (>= n 0)
      ((setter vector-ref) forks n (make-lock))
      (init-forks (- n 1))))

  (defun think (p)
    (format t "~s thinking " p)
    (thread-reschedule))  ;allow some other action while thinking

  (defun eat (p n)
    (format t "~s eating (~a) " p n)
    (thread-reschedule))  ;similarly while eating

  (defun get-ticket (p)   ;issue up to 4 tickets
    (cwait room-ticket)   ;controlled by the counting semaphore
    (format t "~a enters room " p))

  (defun return-ticket (p)
    (format t "~a exits room " p)
    (csignal room-ticket))

  ...

)
```

Figure 8: Dining Philosophers using Counting Semaphores

important points to be considered when writing parallel programs.

3.1. Data Parallel Processing

If writing MIMD programs as part of the undergraduate programme is unusual, writing SIMD programs is relatively exotic. It is certainly a subject that is not dealt with in great detail, if at all, in many undergraduate parallel processing textbooks. While FORTRAN-90 is an important case-study under this heading, we also explore symbolic data-parallel problems and the abstractions that have been developed, mainly in the Lisp community, such as Connection Machine Lisp [13] and Paralation Lisp [11]. Although we have developed a simpler model [9] and implementations of both CM-Lisp and Paralation Lisp on top of that running on a MasPar, the emphasis in teaching is on the Paralation model using a serial simulation.

Probably the most difficult notion to get across is that it is harmless to compute a value that is not used, because it is not wasted computation since it was carried out at the same time as useful values were computed. An easier issue is the need for virtualization of the array once problem sizes exceed physical resources. This is readily appreciated as a tedious matter to implement, but a powerful abstraction once done.

A number of exercises have been worked through in this context, beginning with numerical topics, such as relaxation (see Figure 9), conjugate gradient method, matrix multiplication and moving on to non-numeric problems such as polynomial representation, (univariate) polynomial multiplication, connectionist simulations and classification.

The combined coverage of MIMD and SIMD within a single language also provides a good springboard to the discussion of the simulation of each paradigm in the other and a motivation for architecture independent programming.

3.2. General Parallel Programming

The fact that we can employ all of these parallel models in a single language, using modules to import those constructs we need, is particularly convenient. For example, we have modules that implement Linda, monitors, paralations, a Future-like class of objects, and an occam or CSP-style (message passing on channels) language, and to use any of these is simply a matter of importing the module. This lets us contemplate the question of choosing the appropriate method for the solution of a particular problem, and also opens the way to mixed-paradigm solutions. For example, if we decide that, say, the first phase of a problem would work best using Linda, while the second would benefit from the use of paralations, we can do this

```
(defun relax (grid eps)
  (let* ((index (index grid))
         ;;boolean field indicating whether element is on boundary
         (boundary (elwise (index) ...)))
    (labels
      ((loop (old new)
         (if (vref (lambda (x y) (or x y))
                   (elwise (old new) (< (abs (- old new)) eps)))
             ;;some points not yet converged
             (let ((n (get N new ())) (s (get S new ()))
                   (e (get E new ())) (w (get W new ())))
               (loop new
                 (elwise (new boundary n s e w)
                   (if boundary new (/ (+ n s e w) 4.0)))))
             ;;all points have converged
             new)))
      ;;compute first iterate
      (let ((n (get N grid ())) (s (get S grid ()))
            (e (get E grid ())) (w (get W grid ())))
        (loop grid
          (elwise (grid boundary n s e w)
            (if boundary grid (/ (+ n s e w) 4.0)))))))))
```

Figure 9: Relaxation using paralations

in a single language, in a single program. Further, we are able to compare solutions in different models directly, and transform from one to another. We believe that the subject should be seen as a whole, not a series of compartmentalised ideas that stand in isolation, and the design of EuLisp lets us do this.

4. Language design

EuLisp has also been used in a final-year course in programming language design. The students already know the language from earlier courses, and we can build on this to use EuLisp in three new roles:

1. EuLisp is a contemporary general-purpose language and as such provides a valuable case-study in language design, capturing a number of important concepts in a consistent framework (for example, modules, the object system, exceptions, threads and streams).

2. Like other Lisp dialects, EuLisp can be used to illustrate the traditional 'metacircular' Lisp interpreter, and a correspondence estab-

lished between this and a simplified denotational semantics of the language. As part of the case study, this approach is used to describe the evolution of LISP (as discussed in [14]). EULISP affords two variations on the traditional approach: the interpreter can be presented as a generic function, and it can be extended to include aspects of the EULISP language, notably continuations and threads [5]. Part of the interpreter is illustrated in Figure 10, where the `if` special form has been parsed into a structure `<if>` with three slots: `if-test`, `if-true` and `if-false`.

3. Building on the previous point, EULISP provides a powerful vehicle for teaching issues in language design ([7] adopts a similar approach), as well as a symbolic platform for prototyping little application-oriented language systems. In conjunction with interpreter techniques, the module system is used extensively in assembling a complete language system: each component of the system generates an intermediate program in a language L in the form of a module which imports the implementation of L.

```
(defgeneric generic-eval (k exp env))

(defmethod generic-eval (k (exp <number>) env) (send k exp))

(defmethod generic-eval (k (exp <quotation>) env)
  (send k (value exp)))

(defmethod generic-eval (k (exp <if>) env)
  (generic-eval (lambda (v)
                  (generic-eval k
                    (if v (if-true exp) (if-false exp))
                    env))
                (if-test exp)
                env))
```

Figure 10: Fragment of generic `eval`

5. Other courses

5.1. Object-Oriented programming (advanced)

EULISP level-1 is used in an advanced object-oriented programming course to illustrate a metaobject protocol, where it has replaced CLOS. The TELOS

protocol is simultaneously at the heart of EuLisp, and has a separate, independent, identity. There are many constructs in EuLisp that rely on the object system, but it is possible to extract that system and consider it as an entity in its own right (this has been done through the reimplementation of Telos in other languages, including Scheme and Common Lisp). This orthogonality of design has allowed the development of a powerful MOP independently of the rest of the language: the advantage of having language-independent implementations enables us to discuss the protocol in isolation, not relying on any particular language feature.

5.2. Computer Algebra

One of the classic applications of Lisp is computer algebra, and so EuLisp has been used in a computer algebra course. The object system again plays an important part, as it can abstract away distracting details of some of the algorithms. For example, a traditional way of implementing a polynomial is to use a recursive list structure. We use a collection of macro definitions for accessors to the polynomial structures, and many `cond`s to determine what part of the polynomial we are considering as we work our way down the list (for example, leading coefficient, the degree in the first variable, the reductum of the polynomial, and so on). In contrast, using Telos we can use classes for each different part of the polynomial, and use methods on generic functions to isolate their functionalities. As a way of implementing fast polynomial arithmetic the use of generic functions and structures may not be the most efficient, but certainly they are an excellent way to *explain* the algorithms. Again, the integration of system and user types in Telos provides a uniform interface for the user: the user can add appropriate methods to `+` (via the generic function `binary+`), facilitating natural expressions such as `(+ p q 1)`, where `p` and `q` are of class `<polynomial>`.

5.3. Projects

A number of final-year projects have been based on the EuLisp platform, including the following:

- *Visual programming for* EuLisp. This is an X Windows System application which enables the user to construct programs in a dataflow notation with a graphical editor; the notation is loosely compatible with ProGraph [15], which is an icon-based visual programming environment with a programming model that shares much with EuLisp. The tool generates executable EuLisp code. The consistent treatment of objects in EuLisp permits a very simple graphical syntax.

- EuLisp *editing environment.* An application for OS/2 Presentation Manager, this is an editor which has knowledge about standard EuLisp syntax but also accepts simple extensions to the standard syntax to incorporate additional class information for bindings, arguments to generic functions and some return values. The environment includes a tool (nicknamed *EuLint*) for checking for consistency within a EuLisp application and another for filtering the extended syntax into standard EuLisp code.

- *Score representation languages.* EuLisp has been used to implement simple languages for representing musical scores. These have been used as the target of composition tools, and as the input to tools which generate either MIDI data or source text for typesetting utilities. EuLisp provides a rich environment for construction and combination of such tools, and this application is a good test of its novel event-oriented features.

- *User Guide to* Telos. A project to write a document that firstly gave an overview of the Telos MOP, and secondly showed how it could (and in some ways, could not) be used to emulate the `C++` style of class definition and inheritance. The purpose behind this project was twofold. On the one hand, the student was led by experimentation to understand the MOP; on the other hand, the student was doing real research by investigating the flexibility and extensibility of Telos.

- *A module browser.* The first stage of this project involved using the interface to `gdbm` implemented in FEEL to build a module database. The second stage built a mode for GNU Emacs to browse the module structures stored in the database communicating via a EuLisp process. This presented the user with a display modelled after the GNU Emacs directory browser where each line corresponded to an expression in the module. Selection of an item lead to its display in a new buffer for editing and later storage. A similar mode was provided to edit the module directives. This environment also supported consistency checking in a similar manner to the EuLint program mentioned above.

6. Hardware platforms

The courses described above have been taught on four main platforms: Sun3 and SPARC under SunOS; 386 and 486 PCs under MS-DOS and OS2/V2; Amiga. The primary EuLisp implementation was FEEL [4]; we have also used Aubrey Jaffer's SCM and Scheme-to-C [3], with basic extensions for EuLisp level-0 compatibility.

Inevitably the implementation of level-0 is more complex than Scheme, but one of the goals of is that it should be implementable by students on a reasonably short timescale. So far we have had good experiences with implementing and porting level-0. The provision of interpreters for the popular 'home computers' has enabled students to work on a wide variety of platforms without suffering the constraints of larger Lisp systems.

7. Conclusion

One of the objectives of EuLisp is to provide a teaching platform. We have demonstrated through teaching a number of courses over a three year period that EuLisp performs this function well. Respect for the Scheme principles of simplicity and orthogonality enables EuLisp to be used unobtrusively in teaching non-Lisp related material, and it lends itself equally to advanced Lisp-based courses. We anticipate that EuLisp will continue to be used and expanded in this role, aided by an increasingly rich environment as more tools are created.

In comparison with Scheme, the most significant increase in functionality provided by EuLisp is the object system; modules, conditions and threads are also used extensively in our teaching. These mechanisms are provided without introducing excessive complexity which would impede the educational process. We favour EuLisp over Common Lisp for its layered approach (providing level-0 as a small, well-defined and portable kernel), its natural progression from functional languages (through being a $\mathbf{Lisp}_1$) and the provision of the 'concurrency toolbox', and over CLOS for its simple, integrated classes. We miss first-class continuations and multiple inheritance, though these are needed only in advanced courses where we can achieve them through embedded interpreters and EuLisp level-1 respectively.

Acknowledgements

The authors wish to thank colleagues and students involved in the following computer science courses for allowing us to use them to test the evolving design of the EuLisp language: C7 *Lisp*, C80 *Applications of Logic*, C85 *Concurrent Programming*, MSc in Computer Algebra (Bath); CM203 *Computational Systems*, CM333 *Programming Language Design*, CM358 *Advanced Lisp*, CM364 *Object-Oriented Programming and Systems* CM366 *Symbolic Computation*, CM367 *Models of Programming* (Southampton); Concurrent Processing, Ph.D. programme (UPC Barcelona); Data Structures (Warwick).

Considerable thanks to the authors of *Structure and Interpretation* for their influential work. Keith Playford and Pete Broadbery implemented FEEL. Neil Berrington performed the 386 port, and provided EuLisp level-0 support on a variety of platforms.

References

1. Abelson, H and Sussman, G J with Sussman, J. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts (1985).

2. Andrews, G A. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, California (1991).

3. Bartlett, J F. *Scheme->C a Portable Scheme-to-C Compiler.* Research Report 89 1, DEC Western Research Laboratory, Palo Alto, California (January 1989).

4. Broadbery, P., et al. FEEL. Available by anonymous FTP from `pub/eulisp` on `ftp.bath.ac.uk` (1992).

5. De Roure, D C. *QPL3—Continuations, Concurrency and Communication.* Technical Report CSTR 90–20, Department of Electronics and Computer Science, University of Southampton (1990).

6. De Roure, D C and Padget, J. Guaranteeing Unpredictability. To appear (1993).

7. Friedman, D P, Wand, M, and Haynes, C T. *Essentials of Programming Languages.* MIT Press, Cambridge, Massachusetts (1992).

8. Keene, Sonya E. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley, Reading, Massachusetts (1989).

9. Merrall, S and Padget, J A. Plural EuLisp: A Primitive Symbolic Data Parallel Model. *LASC*, 6, 1/2 (September 1993).

10. Rees, J A and Clinger, W. The revised^3 report on the algorithmic language Scheme. *ACM Sigplan Notices*, 21, 12 (December 1986) 37–79.

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

12. Steele Jr, G L, Fahlman, S E, Gabriel, R P, Moon, D A, Weinreb, D L, Bobrow, D G, DeMichiel, L G, Keene, S E, Kiczales, G, Perdue, C, Pitman, K M, Waters, R C, and White, J L. *Common Lisp: The Language (Second Edition)*. Digital Press, Bedford, Massachusetts (1990).

13. Steele Jr, G L and Hillis, W D. Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing. In *ACM Conference on Lisp and Functional Programming* (1986) 279–297.

14. Steele Jr, G L and Sussman, G J. *The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two)*. AI Memo 453, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts (May 1978).

15. Szpakowski, M, Pietrzykowski, T, Laskey, J, Kilshaw, T, Eyre, P, and Cox, P. *Prograph Reference: A very high-level, pictorial, object-oriented programming environment*. TGS Systems, Halifax, Canada (1989).