Advanced Programming Techniques Using Scheme

Luc Moreau[1], Daniel Ribbens[2], and Pascal Gribomont[2]

1: *Department of Electronics and Computer Science,*
University of Southampton,
Southampton SO17 1BJ. United Kingdom.
E-mail: L.Moreau@ecs.soton.ac.uk.

2: *Institut d'Electricité Montefiore B28,*
Université de Liège,
4000 Liège, Belgium.
E-mail: (ribbens,gribomon)@montefiore.ulg.ac.be.

Résumé

There are not many non-trivial examples that can be used in a course on advanced programming concepts. In this paper, we describe an interactive reducer for lambda terms that combines first-class continuations, macros, delay, and state. We also describe the means by which we induce students to master advanced topics.

1. Introduction

Numerous articles praise the benefit of functional programming through the curriculum [14], and several textbooks on functional programming are now widely used in introductory programming courses [2, 4, 10, 24, 17]. Since the early seventies, we have based our courses on various dialects of Lisp [23], and we are now using Scheme. We believe that a Scheme-style[1] of programming relies on a few but expressive concepts that favour abstraction and ease of programming.

However, we observe that very few educational texts present difficult programming techniques and their use in non-trivial examples. Too many authors are satisfied with giving a sorting algorithm or Hanoi towers. We do not deny that one must start with the basics, but

[1]We focus more on a programming philosophy than a particular programming language. ML is a perfectly suitable candidate; in particular, the implementation of Standard ML of New-Jersey could be used to program the example described in this paper (except the macro).

we believe that an *advanced programming* course should include more difficult topics. Scheme offers a range of advanced programming concepts like continuations, lazy evaluation via delay, engines, or macros. As soon as they are grasped by students, they can be used quickly to program an operating system kernel [7, 25] or an object-oriented extension to a language [21, 1].

It is still an educational challenge to present difficult programming examples. First, there are few such examples published in the literature: most of them appear as research articles (like the widely cited references mentioned above), and very few are published in educational conferences, highlighting the pedagogical difficulties to present them. Second, even though we wish to present difficult examples, length is an important consideration, because the material should be presentable in a single lecture typically. Third, there are advanced examples focusing on a given technique, like Henderson's graphic primitives that rely on higher-order functions [15], but few combine several techniques at the same time.

In this paper, we present an advanced programming example that uses first-class continuations, macros, delay, and state in a combined way. This program, which is an interactive reducer for lambda terms, illustrates the level reached by students in the last year of Electrical Engineering and Computer Science. First, we describe the curriculum in Electrical Engineering and Computer Science at the Montefiore Institute; second, the example and its subtle aspects are presented; third, a discussion follows on the qualities of this example and on how students perceive it.

2. Curriculum in Electrical Engineering and Computer Science

In this section we first give a description of the functional programming part of the curriculum, and then comment on some aspects of the courses which should lead our students to actively master difficult notions and to use them for solving difficult programming problems.

2.1. The Curriculum in Functional Programming

Students in Electrical Engineering and Computer Science take in sequence three courses on functional programming[2]:

1. *Elements of programming*,

[2] Besides functional programming, the curriculum includes databases, complexity, parallel programming, compilers, operating systems, assembly languages, etc.

2. *Semantic Aspects of Programming*,

3. *Advanced Concepts in Programming Languages.*

There is a close collaboration between courses 1 and 2. Many concepts are introduced informally in course 1 and are revised more formally in course 2. Course 1 is the basic course of functional programming. The main target is to master abstraction, with both procedures and data. We especially emphasize recursion, modularity and higher-order functions. No explicit theory is given, except maybe a semi-formal introduction to the substitution model for evaluation.

Course 2 presents the substitution model in a formal way and illustrates its limitations. The environment model is also introduced, and used to discuss various subtle topics, like free variables, lexical scoping versus dynamic scoping, evaluation of expressions with side effects and assignments, etc. A meta-circular evaluator is given as a third semantic model. Important programming techniques are also introduced, like streams and message-passing serving as an introduction to object-oriented programming.

Course 3 introduces continuation-passing style on simple recursive procedures. Then, a meta-circular evaluator in continuation-passing style is derived. We describe *call/cc* using the intuitively-defined notion of evaluation context, and then we define it precisely in the meta-circular evaluator. Simple and more complex examples of continuations follow. Interpretation and compilation techniques are then presented, and parallelism is introduced with the **future** construct [13].

We have adopted the language Scheme in all three courses. The small number of general rules and constructs of Scheme suffices to obtain a practical programming language, that is flexible enough to support most of the major programming paradigms [22]. As the language is small and close to the lambda calculus, we can smoothly study more theoretical topics, like operational semantics or denotational semantics, without becoming too complicated.

Examination consists mainly in writing and documenting small programs. More theoretical topics, like evaluation semantics, are examined by requiring the students to simulate the evaluator behaviour. Students get acquainted with medium-sized programs through homeworks, and their graduation theses may involve a substantial effort in programming.

2.2. Building Programming Ability

The ultimate goal of programming courses is to induce the ability of solving problems by writing computer programs. As a result, we try

to avoid a common danger : students learn a lot of concepts, but tend not to use them, or to use them poorly or reluctantly, when they write programs by themselves. As a rule, we try to avoid the situation where a student has some understanding of a concept or construct, but is not sufficiently at ease with it and does not use it, or only reluctantly, in his own programs.

The problem usually occurs early, with higher-order functions. Experience shows that a *formal* introduction of procedures as arguments, and especially of procedures as values, is not an adequate way of inducing students to adopt a functional style of programming. In fact, the very claim that "functions are first-class objects" seems artificial to students; calculus courses may have induced quite the opposite idea: objects are numbers, strings, etc. and functions are simply laws, or mechanisms, by which objects are associated with, or obtained from, other objects. Our approach is to introduce elementary examples of higher-order functions, like *map* or *filter*; we delay the general claim "functions really are (first-class) objects and should be considered as such" in favour of a more modest claim like "*map* iterates on a list and performs on each element an operation specified by its functional parameter". From the theoretical point of view, we adopt a similar strategy and, in course 1, there is only a semi-formal introduction to the most elementary computation model, that is, the substitution model. More complete models are introduced in a formal way in course 2.

Our main strategy is not to avoid the explicit introduction of difficult concepts, but to delay the *formal* introduction at a time where the concept is likely to be accepted without toil. Similarly, the important concepts of functional programming should be combined into difficult design, but not to soon. This strategy of delaying the *formal* introduction of difficult notions is compatible with the student mastering difficult concepts and constructs, like continuations and *call/cc*, **delay** and *force*, macro writing, assignments and mutable data. We simply go from practice to theory; in order to demonstrate this, we describe the learning steps that may lead a (reasonably skilled) student to use these concepts and constructs in his/her own programs.

In fact, *call/cc*, **delay** and *force*, macro writing, assignments and mutable data are formally introduced in courses 2 and 3, but are "prepared" in course 1. In the sequel to this paragraph, we comment about this preparation.

2.2.1. Delayed evaluation

There is already a hint at the notion of evaluation order at the very beginning of course 1, when we show that the "natural" evaluation of

arithmetic expressions involves a partial ordering: expressions are trees, and the value of a node can be computed only when the values of its children are known.

A more significant hint is given when the **lambda** abstraction is introduced. We simply observe that, when evaluating expressions like

((**lambda** (w) (* 2 w))
 ((**lambda** (v) (− 9 ((**lambda** (u) (+ 3 u)) v))) 5))

there are several reduction orders. We point out that a Scheme evaluator always uses a call-by-value strategy, but any other reduction ordering would lead to the same result, although not necessarily consuming the same amount of time and space.

Further steps are taken in course 2, where call-by-name, normal order strategy is introduced; we show that the space and time complexity of the evaluation process may be modified substantially, and go on with a formal presentation of the normal order with a modified evaluator. At that time, students are ready to accept that the call-by-value strategy might be replaced by lazy evaluation for some applications, using constructs like **delay** to simulate this strategy in the evaluator. This allows us to solve interesting problems rather easily with the stream paradigm and to write, for instance, an interpreter for (non-deterministic) finite automata, or a solution for the Grune problem (coroutines).

2.2.2. Macro

Macro is clearly an advanced topic, which is only briefly addressed in course 2, and properly introduced in course 3. Nevertheless, the preparation and motivation work for macros begins in course 1, when discussing **let** as a syntactic but pragmatically useful variant of **lambda**; we mention that Scheme programmers may create and use their own syntactic constructs with macros. We also show early (in an elementary case) that Scheme functions can be used to produce Scheme code, by writing a function *describe*, such that (*describe* '(1 ((a) 2))) evaluates into (*cons* 1 (*cons* (*cons* (*cons* 'a '()) (*cons* 2 '())) '())).

The preparation and motivation work goes on in course 2, where object-oriented programming is introduced through message-passing. We adopt the "Tiny Object-Oriented Language" [1], a meta-circular evaluator extended for object-oriented style. At this point, we mention that macros would allow us to define such an object-oriented extension without resorting to an evaluator.

2.2.3. Continuation and *call/cc*

The *call/cc* construct is also an advanced topic we introduce only in course 3, but whose preparation and motivation begin in course 1. In fact, our first encounter with the concept of continuation occurs when we program simple functions in continuation-passing style.

A further step allows us to introduce contexts. For some recursive functions, we can encode the recursion context, i.e. the continuation, into an explicit argument. In some interesting cases, the encoding does not increase as fast as the size of the stack, which allows us to derive iterative functions using an accumulator, in a similar way to Wand [26].

Last but not least, we also address in course 1 the classical example of list multiplication when occurrences of `0` are likely, using CPS to avoid any unnecessary multiplication. This is a first hint that continuation can be a useful concept for non-local exits.

More direct examples of continuation handling can be considered in course 3, when students know more about environment and evaluation process. A nice example is the implementation of coroutines.

Non-local exits and coroutines are considered again in course 3, when *call/cc* is properly introduced. Other previously encountered concepts are revisited, and used to illustrate the usefulness of both the concept of continuation, and the new construct *call/cc*. At this time, the student is ready to assimilate *call/cc*, and to use it for more advanced notions, backtracking, and engines.

2.2.4. Assignments and Mutable Data

Course 1 is an introduction to functional programming, for students who already have written (elementary) PASCAL programs. An early introduction of **set!** and the like might impair the assimilation of the functional paradigm and of the substitution model of evaluation, but completely ignoring assignment might induce the wrong feeling that "pure functional programming" is the universal solution. We choose to mention and use assignments only twice, at the end of the course, in very specific circumstances.

One of them is when we show that a naïve approach of recursion can lead to inefficient programs and that the memoization technique can be used to improve them. The memo structure can be an a-list, but a mutable vector, updated with *vector-set!*, is also an appropriate solution.

The other use of assignment in course 1 is when the distinction between **let** and **letrec** is discussed. It seems worth mentioning that the behaviour of **letrec** can be simulated with **let** and **set!**. So assignments

are used sparingly, and only when they are really useful.

Any less "encapsulated" use of assignment is delayed to courses 2 and 3 where, as usual, concepts only "hinted" in course 1 are considered again, in more depth and also more formally. The environment model introduced in course 2 is used to explain in detail the evaluation of forms containing assignments, and problems where mutable data are the natural choice are solved. Delayed evaluation is revised by describing the implementation of **delay** using assignment.

At this point, students are ready for more advanced examples, like engines, an operating-system kernel [7, 25], or the example given below. By adopting the language Scheme, we are able to reach advanced topics rather quickly, using a single programming framework that allows us to concentrate on concepts.

3. Example

The program that we present in this section is an interactive reducer for lambda terms. It reads a lambda term (represented as an S-expression), displays the different redices of the term, waits for the user's selection, replaces the selected redex by its contractum, and repeats this process until it reaches a normal form.

3.1. Lambda terms

In this paper, we shall assume that the reader is familiar with the basic principles of the λ-calculus, and in particular of the call-by-value λ-calculus [19] that we adopt here. During course 2, we take the opportunity to introduce the syntax of the λ-calculus, the notion of (call-by-value) β and δ-reductions, and the definition of terms like redex, contractum, context, and normal form; this material is thoroughly studied in course 3.

Terms are represented by S-expressions satisfying the following grammar:

$$\begin{array}{rcll} M & ::= & (M\ M) \mid (\mathsf{lambda}\ v\ M) \mid v \mid n \mid p & \text{(Term)} \\ v & ::= & \textit{a symbol} & \text{(Variable)} \\ n & \in & \{0, 1, \ldots\} & \text{(Number)} \\ p & \in & \{+, -, /, *, \ldots\} & \text{(Primitive)} \end{array}$$

A value is a lambda term, a variable, a primitive, or a number.

```
(define value?
  (lambda (x)
    (or (symbol? x) (number? x) (and (pair? x) (eq? (car x) 'lambda)))))
```

We define[3] the function *components*, which returns the list of subterms of a term. The function *constructor-of* is a higher-order function: given a term x, it produces a function that constructs a term of the same type as x for some given subterms.

```
(define components
 (lambda (x)
  (match x
   ((? number?) '())
   ((? symbol?) '())
   (('lambda v body) (list body))
   ((a d) (list a d)))))
```

```
(define constructor-of
 (lambda (x)
  (match x
   ((? symbol?) (lambda () x))
   ((? number?) (lambda () x))
   (('lambda v body) (lambda (b)
                       (list 'lambda v b)))
   ((a b) (lambda (new-a new-b)
            (list new-a new-b))))))
```

The predicate *redex?* determines whether its argument is a redex; the function *contractum* returns the contractum of a redex. The functions implementing the beta and delta reductions are not given in this paper.

```
(define redex?
 (lambda (applic)
  (match applic
    ((('lambda x body) V) (value? V))
    (((? primitive?) V) (number? V))
    (_ #f))))
(define primitive?
  (lambda (p)
    (and (symbol? p)
         (memq p '(+ * / - add1 sub1)))))
```

```
(define contractum
 (lambda (applic)
  (match applic
   ((('lambda x body) V)
    (substitution body V x))
   (((? primitive? p) V)
    (delta-reduction p V)))))
```

3.2. Searching and Constructing

The interactive program is composed of two phases: the *search* phase finds all the redices of a lambda term, and the *construction* phase returns a new lambda term where the user-selected redex is replaced by its contractum. A naïve implementation of the program would alternate the search and constructing phases, performing a complete search for every new term and reconstructing a complete term after each reduction. The

[3] We believe that pattern matching is a useful programming technique. As the language Scheme [22] does not support it, we use Wright and Duba's [27] **match** macro. A brief explanation of **match** appears in Section A.

efficiency of such a program can be improved by taking into account the following observations:

1. As the redex and contractum appear in the same context (cf. Figure 1), the search and constructing phases for the context are the same before and after transition.

2. The search and constructing phases use the same recursion schema based on the grammar of terms.

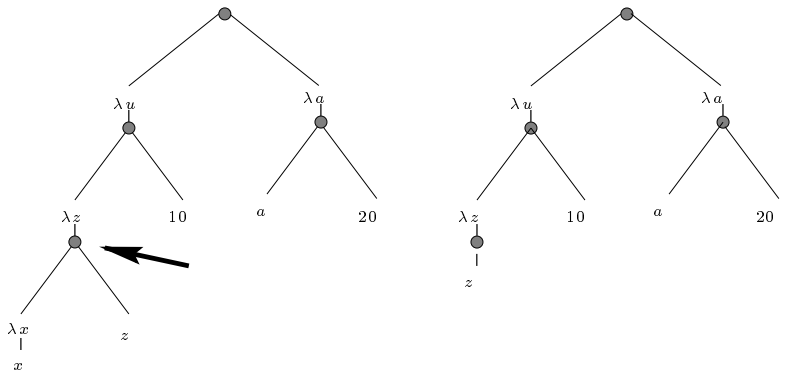

Figure 1: Redex replaced by a contractum in the same context

The function *search+construct* combines the search and construction phases. It takes a term and a predicate as arguments, and returns two results: *(i)* a copy of the term passed in argument, *(ii)* the list of subterms satisfying the predicate. Both results are combined into a datastructure *result*. In the program, we use the predicate *redex?* to select the list of redices of a term. For the time being, let us assume that the function *map!!* is defined as *map*.

```
(define search+construct
  (lambda (term pred)
    (let-values ((k term) (call/cc (lambda (k)
                                     (values k term))))
      (match (merge-results (constructor-of term)
                            (map!! (lambda (x)
                                     (search+construct x pred))
                                   (components term)))
        ((and res ($ result new-term success))
         (if (pred new-term)
             (make-result new-term (cons (list k new-term) success))
             res))))))
```

```
(define-structure (result term success))
```

Multiple values can be returned by the primitive *values* and matched using **let-values** [6]. If the Scheme implementation does not support multiple values, these constructs can be defined in terms of lists as follows:

```
(define values list)

(define-syntax let-values
  (syntax-rules ()
    (((let-values ((var vars ...) val) body1 bodyn ...)
     (apply (lambda (var vars ...) body1 bodyn ...) val))))
```

Let us observe that the recursion schema of *search+construct* is implicit as we use the generic accessor *components* and the higher-order function *constructor-of*. In the *success* slot of a structure *result*, the function *search+construct* associates each term satisfying the predicate with the continuation that existed when *search+construct* was called on that term.

The function *merge-results* combines the structures *result* obtained for the components of a term into a new structure *result*:

```
(define merge-results
  (lambda (constructor args)
    (let ((terms (map result-term args))
          (success (mapcan result-success args)))
      (make-result (apply constructor terms) success))))
```

> **Remark** In order to reduce the allocation cost, we can avoid to reconstruct a new term, and we could do some sharing in the function *constructor-of*.

3.3. Interactive Loop

The entry point of the program is the function *driver-loop*; it reads a term, displays its redices, reduces the user-selected redex, and repeats the process until a normal form is reached or until the user closes the input stream. In the latter case, a non-local exit is performed by calling the continuation **abort**, initialised by *with-abort*.

```
(define abort 'any)
(define with-abort
  (lambda (thunk)
    (call/cc (lambda (k)
               (set! abort k)
               (thunk))))))
```

The interactive loop *driver-loop* calls *search+construct* and then *return-redex*, whose role is to display the list of redices and to return the one chosen by the user.

```
(define driver-loop
  (lambda ()
    (with-abort (lambda ()
                  (match (search+construct (initial-prompt) redex?)
                    (($ result term l)
                     (display-term term)
                     (if (null? l)
                         term
                         (let ((the-redex (return-redex l)))
                           (match the-redex
                             ((n k tree)
                              (k (values k (contractum tree))))))))))))))))))))))
(define return-redex
  (lambda (l)
    (let ((el (enumerate-terms l)))
      (let ((val (let loop ((val (begin (prompt el)
                                        (read))))
                   (if (eof-object? val)
                       (abort 'end-of-computation))
                   (if (not (and (number? val)
                                 (> val 0)
                                 (<= val (length el))))
                       (loop (begin (display "Invalid number")
                                    (newline)
                                    (prompt el)
                                    (read)))
                       val))))
        (let loop ((el el))
          (cond ((null? el) (error 'driver-loop "Did not find redex " val))
                ((= (caar el) val) (car el))
                (else (loop (cdr el))))))))))
(define enumerate-terms
  (lambda (l)
    (let loop ((n 1)
               (l l))
      (if (null? l)
          '()
          (cons (cons n (car l))
                (loop (+ n 1) (cdr l)))))))
```

```
(define prompt
  (lambda (l)
    (display "Which redex do you want to reduce?")
    (newline)
    (for-each (lambda (x)
                (match x
                  ((n k tree) (display n)
                              (display " : ")
                              (pp tree))))      ;;; pretty-print
              l)))
```

Iterating is done by calling the continuation associated with a redex on the continuation and the contractum. Evaluation resumes in *search+construct* as if the *call/cc* expression was returning the continuation and the contractum. Then, the contractum obtained after reduction is recursively searched by the function *search+construct* in order to obtain its redices. A crucial point in the program is when to check when a term satisfies the predicate *pred*. If we do it on *term* when entering *search+construct*, we will not take into account that subterms might have been reduced. Instead, we call *pred* on *new-term*, the term returned as part of the result. This guarantees that we check whether the *current* term satisfies the predicate by taking into account all reductions.

With our teaching experience, we are aware that some students have difficulties to understand where the function *search+construct* is resumed when a redex is selected. We observe it is convenient to add debugging information to the function *search+construct*:

```
(define search+construct
  (lambda (term pred)
    (let-values ((k term) (call/cc ...))
      (display-trace "Searching:    " term)
      (match (merge-results ...)
        ((and res ($ result new-term success))
         (display-trace "Constructing: " new-term)
         (if ...))))))
(define display-trace
  (lambda (str term)
    (display str)
    (display term)
    (newline)))
```

This debugging information is even more helpful if the trace is indented: the expression nesting being represented by its indentation on the screen. We do not show the code in the paper, but it presents an extra difficulty as the indentation level has a dynamic nature (like dynamic binding [16]), and its definition in the presence of first-class continuations is an instance of the state-space problem [11].

3.4. Continuations and State

So far, we assumed that the function *map!!* was defined as *map*, but the program would not behave as expected. Figure 2 displays a sample execution if *map* was used. The user has successively selected the third and second redices. Then, the reducer amazingly tells us that `((lambda a a) 20)` is still a possible redex. At this point, the term to reduce should have been `((lambda u ((lambda x x) 10)) 20)`, instead of `((lambda u ((lambda x x) 10)) ((lambda a a) 20))`. Intuitively, the reducer has forgotten that `((lambda a a) 20)` had already been reduced. It is as if we wanted to *mutate* a data structure (the term to reduce), but the implementation with *map* was *stateless*.

```
Enter a lambda term
((lambda u ((lambda z ((lambda x x) z)) 10)) ((lambda a a) 20))

========================================================================
((lambda u ((lambda z ((lambda x x) z)) 10))
 ((lambda a a) 20))

Which redex do you want to reduce?
1 : ((lambda z ((lambda x x) z)) 10)
2 : ((lambda x x) z)
3 : ((lambda a a) 20)
3

========================================================================
((lambda u ((lambda z ((lambda x x) z)) 10)) 20)

Which redex do you want to reduce?
1 : ((lambda u ((lambda z ((lambda x x) z)) 10)) 20)
2 : ((lambda z ((lambda x x) z)) 10)
3 : ((lambda x x) z)
2

========================================================================
((lambda u ((lambda x x) 10)) ((lambda a a) 20))

Which redex do you want to reduce?
1 : ((lambda x x) 10)
2 : ((lambda a a) 20)    ;;;; ******* was already reduced
```

Figure 2: Stateless implementation with `map`

The *search+construct* function traverses lambda terms in the order defined by the Scheme implementation. For our purpose, let us assume that this order is from left to right. When the continuation associated with a redex is called, it resumes the search on the contractum *and on the part of the tree that appears to the right of the contractum*. This

explains, why selecting redex 2 in Figure 2 makes the redex `((lambda a a) 20)` available again: when the continuation associated with redex 2 was captured, `((lambda a a) 20)` was a subexpression that remained to be traversed.

Our first attempt to correct this problem is to define a function *map-box!!* which uses the **box!!** macro and a box datastructure, i.e. a single-field mutable structure.

```
(define map-box!!
  (lambda (f l)
    (if (null? l)
        '()
        (box!! cons (f (car l)) (map-box!! f (cdr l))))))

(define (make-box) (cons 'box '()))
(define (box-ref x) (cdr x))
(define (set-box! x v) (set-cdr! x v))
```

The **box!!** macro is defined using Dybvig's **syntax-case** [5].

```
(define-syntax box!!
 (lambda (x)
  (syntax-case x ()
    ((_ M M1 ...)
     (with-syntax (((boxM boxM1 ...)
                    (generate-temporaries (syntax (M M1 ...))))
                   ((tmpM tmpM1 ...)
                    (generate-temporaries (syntax (M M1 ...)))))
       (syntax (let ((boxM (make-box))
                     (boxM1 (make-box))
                     ...)
                 (let ((tmpM (begin
                               (set-box! boxM M)
                               (lambda (boxM1 ...)
                                 ((box-ref boxM) (box-ref boxM1) ...))))
                       (tmpM1 (begin (set-box! boxM1 M1) boxM1))
                       ...)
                   (tmpM tmpM1 ...)))))))))
```

For each operand of an application, the macro **box!!** introduces a box in which the operand value can be stored. Unfortunately, this function produces exactly the same behaviour as in Figure 2, assuming a left to right evaluation order. Indeed, in the expression (**box!!** *M M1* ...) a continuation captured in *M* also re-evaluates *M1*, whereas we want the continuation captured in *M* to use the latest value returned by *M1*.

Our solution is the following *map!!* function which uses the macro **!!** in the inductive case.

```
(define map!!
  (lambda (f l)
    (if (null? l)
        '()
        (!! cons (f (car l)) (map!! f (cdr l))))))

(define-syntax !!
 (lambda (x)
  (syntax-case x ()
   ((_ M M1 ...)
    (with-syntax (((boxM boxM1 ...)
                   (generate-temporaries (syntax (M M1 ...))))
                  ((tmpM tmpM1 ...)
                   (generate-temporaries (syntax (M M1 ...)))))
     (syntax
       (let ((boxM (make-box))
             (boxM1 (make-box))
             ...)
         (let ((tmpM (delay (begin
                              (set-box! boxM M)
                              (lambda (boxM1 ...)
                                ((box-ref boxM) (box-ref boxM1) ...)))))
               (tmpM1 (delay (begin (set-box! boxM1 M1) boxM1)))
               ...)
           ((force tmpM) (force tmpM1) ...)))))))))
```

The macro **!!** differs from **box!!** by the **delay**[4] and *force* expressions, which guarantee that if (*force tmpM1*) is evaluated several times, its value is always a box, but it evaluates *M1* only once.

> **Remark** An application "annotated" by **!!** is an embedding of a sequential version of Queinnec's semantics of applications in ICSLAS [20].

4. Discussion

Mostly-functional languages such as Scheme and ML rely on a functional core but also use imperative features for very specific purpose. Imperative features in these languages are used for a different reason than in imperative languages. This single example uses several of these programming techniques:

- continuations are used for resuming and aborting computations;
- side-effects are used explicitly to program mutable data structures;

[4] As in [22], **delay** guarantees that only one value is computed for a promise.

- the imperative behaviour of the program is abstracted in the macro **!!** so that the function *map!!* can have the same recursion schema as *map*;
- **delay** ensures that some expressions are evaluated once at most.

The most controversial issue in this solution is probably the usage of continuations. The reader might think that this example was artificially created to use such a programming technique. This is not the case. We chose this example because the problem is not trivial and its explanation is short. Could we have programmed the solution differently? We could have avoided first-class continuations by using an explicit continuation-passing style; such a solution essentially presents the same difficulty, only adding the burden of passing extra continuation arguments. The key reason for using continuations in this example is to share the search and construction of a subtree between two reduction steps. In addition, the continuation implicitly encodes various information that would have to be made explicit in their absence, such as the position of the redex in the tree, the associated list of redices that are no longer valid, i.e. those that were found when searching the redex.

If the programmer decides to abandon the use of continuation in this example, the idea of maintaining a state as reduction proceeds still remains valid. Macros could also be used to separate the state-mutating functionality from the recursion schema for data structure traversal. A state can also be represented in a functional manner by a stream [2, 3]; such a solution would further rely on the **delay** construct. The coroutining effect between the construction phase and the interactive loop would be implemented using lazy evaluation instead of continuations. This might be an interesting alternative using a purely functional style, but it also offers pedagogical difficulties as it relies on mutually recursive implicit streams. Let us mention here that our goal was not to present an efficient reduction technique for lambda terms; there exist some techniques that are more appropriate for this purpose, like abstract machines or graph reduction [18].

Our teaching experience is that there are three categories of students. Very few students were unable to understand the program; our interpretation is that they had not managed to build their own intuition of continuations, and therefore were unable to understand the key idea of this program. The vast majority of the students were able to understand the program. We believe that the interactive nature of the program, the possibility of displaying debugging traces, and the small number of functions were important reasons in their understanding. We still believe that the most difficult problem for them is to understand the use of continuations in the function *search+construct*. However, we

observed that only the best students are able to reproduce a similar style of programming. This requires the ability to deduce that a common set of operations, i.e. reconstructing a *result* for the context, can be abstracted into a continuation.

This example can also be used for a series of homework: the trace indentation mentioned above, extension of the language with other constructs like **if** or datastructures like pairs, use of a Felleisen's style of semantics for non-functional features [9], other parameter-passing conventions.

With this example, we have killed two birds with one stone: we have described a program with advanced programming techniques, but also we have illustrated the reduction semantics of the lambda-calculus. Some might argue that this example is a functional program about a functional program, and it does not deal with the real world. In fact, this program has some features that go beyond this specific application: it is interactive and it is presenting and reorganising information according to the user's choice. We are considering other alternative application to this program. Currently, we are investigating the problem of navigating a hypertext system: users are offered a choice of links to follow, and selecting a link results in displaying a new document itself containing links.

5. Conclusion

This single example highlights various advanced programming techniques: first-class continuations in a coroutine-style, non-local exits, delayed evaluation, syntactic abstraction, and mutable datastructures.

It also forces the student to understand interpretation techniques; it shows how evaluation order determines program behaviour, and it helps to distinguish control from state.

6. Acknowledgement

Luc Moreau was partially supported by EPSRC GR/K30773 and EC project reference ERB 4050 PL 930186. Thanks to the anonymous referees for their useful comments.

A. Appendix: Pattern Matching

In this paper, we are using a subset of Wright and Duba's [27] **match** macro. The syntax of a **match** expression is as follows:

(**match** *exp* (*pat* . *body*) ...)

The value of *exp* is matched against the first pattern *pat*. If the matching succeeds, then the body of the clause is evaluated in the scope of the variables appearing in the pattern. Otherwise, the process is repeated with the following pattern. If no pattern matches the value of *exp* the **match** has an unspecified value.

A pattern can be:

- (), #t, #f, strings, numbers, quoted sexpressions: these constants match themselves.
- variable: such a pattern matches any value, and results in binding the variable with the value.
- (*pat1* ... *patn*) matches a list of n elements, and each *pati* must match the ith element of the list.
- (*? predicate pat1* ... *patn*): In this pattern *predicate* must evaluate to a unary function. The pattern matches a value if the predicate applied to the value returns true *and* all *pati* also match the value.
- (*$* **struct** *pat1* ... *patn*) matches a structure **struct** defined by **define-structure**, and each component is matched by a subpattern *pati*.

Let us consider the function *redex?*:

```
(define redex?
 (lambda (applic)
  (match applic
    ((('lambda x body) V) (value? V))
    (((? primitive?) V) (number? V))
    (_ #f))))
```

The value bound to *applic* matches the first pattern if it is a list of two elements, and if the first element is itself a list of three elements starting with the symbol `lambda`. In addition, the body of the clause will be evaluated in the scope of three new bindings for x, *body*, and V. Let us assume the value of *applic* does not match the first pattern. It

matches the second one if it is a list of two elements, whose first element satisfies the predicate *primitive?*.

The structure constructor **define-structure** has the following syntax:

(**define-structure** (**struct** *arg1 ... argn*))

It defines a constructor *make-struct* and accessors *struct-argi*.

B. Auxiliary Functions

A few auxiliary functions were not defined in the core of the paper.

```
(define mapcan
  (lambda (f l)
    (if (null? l)
        '()
        (append (f (car l))
                (mapcan f (cdr l))))))
```

The function *display-term* is called after every reduction and displays the reduced term.

```
(define display-term
  (lambda (term)
    (newline)
    (display "======================")
    (newline)
    (pp term)
    (newline)))
(define initial-prompt
  (lambda ()
    (display "Enter a lambda term")
    (newline)
    (read)))
```

The substitution takes care of renaming bound variables in *exp* that belong to the set of free variables of *new*. In this definition, we look for simplicity and not efficiency.

```
(define substitution
  (lambda (exp new id)
    (match exp
      ((? symbol?) (if (eq? exp id) new exp))
      (('lambda id2 body)
       (if (eq? id2 id)
           exp
           (if (member id2 (free-variable new))
```

```
            (let ((id3 (gensym)))
              (list 'lambda
                    id3
                    (substitution (substitution body id3 id2)
                                  new
                                  id)))
            (list 'lambda id2 (substitution body new id)))))
      ((operator operand) (list (substitution operator new id)
                                (substitution operand new id)))
      ((? number?) exp))))
(define free-variable
  (lambda (exp)
    (match exp
      ((? symbol?) (list exp))
      ((? number?) '())
      (('lambda id body) (remove id (free-variable body)))
      ((operator operand) (append (free-variable operator)
                                  (free-variable operand))))))
```

Finally, we define delta reduction on the functional constants `add1` and `square`.

```
(define delta-reduction
  (lambda (p V)
    (cond ((eq? p 'add1) (+ V 1))
          ((eq? p 'square) (* V V)))))
```

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. Languages for object-oriented programming (laboratory assignment associated with MIT course 6.001, about section 4.1 of [2]). Available at `http://www-mitpress.mit.edu/sicp/psets/ps7oop/`.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[3] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Series in Computer Science. Prentice-Hall International, 1988.

[4] Jacques Chazarain. *Programmer avec Scheme. De la pratique à la théorie.* International Thomson Publishing France, Paris, 1996.

[5] Kent Dybvig. Writing Hygienic Macros in Scheme with Syntax-Case. Technical Report 356, Indiana University, Bloomington, Indiana, June 1992.

[6] R. Kent Dybvig. *The Scheme Programming Language.* Prentice-Hall, 1987.

[7] R. Kent Dybvig and Robert Hieb. Engines from Continuations. *Computer Languages*, 14(2):109–123, 1989.

[8] Michael Eisenberg. *Programming in Scheme*. The Scientific Press, 507 Seaport Court. Redwood City. CA 94063-2731, 1988.

[9] Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 2(4):235–271, 1992. Technical Report 100, Rice University, June 1989.

[10] Daniel P. Friedman and Matthias Felleisen. *The Little Schemer*. The MIT Press, fourth edition, 1996.

[11] Daniel P. Friedman and Christopher T. Haynes. Constraining Control. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, pages 245–254, New Orleans, LA., January 1985. ACM.

[12] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. McGraw-Hill, 1992.

[13] Robert H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp: Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, number 441 in Lecture Notes in Computer Science, pages 2–57. Springer-Verlag, 1990.

[14] Pieter H. Hartel and Rinus Plasmeijer, editors. *Functional Programming Languages in Education (FPLE'95)*, number 1022 in Lecture Notes in Computer Science, Nijmegen, The Netherlands, December 1995. Springer-Verlag.

[15] Peter Henderson. Functional Geometry. In *Lisp and Functional Programming*, pages 179–187. ACM, 1982.

[16] Luc Moreau. A Syntactic Theory of Dynamic Binding. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT/FASE'97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 727–741, Lille, France, April 1997. Springer-Verlag.

[17] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[18] Simon Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[19] Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[20] Christian Queinnec. A Concurrent and Distributed Extension of Scheme. In D. Etiemble and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE'92)*, number 605 in Lecture Notes in Computer Science, pages 431–446, Paris, June 1992. Springer-Verlag.

[21] Christian Queinnec. *Lisp in Small Pieces.* Cambridge University Press, 1996. ISBN 0 521 56247 3.

[22] Jonathan Rees and William Clinger. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

[23] Daniel Ribbens. *Programmation non numérique: Lisp 1.5.* Monographies d'informatique, AFCET, Dunod, 1969.

[24] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming.* MIT Press and McGraw Hill, 1989.

[25] Mitchell Wand. Continuation-Based Multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28, 1980.

[26] Mitchell Wand. Continuation-Based Program Transformation Strategies. *Journal of the Association for Computing Machinery*, 27(1):164–180, January 1980.

[27] Andrew W. Wright and Bruce F. Duba. Pattern Matching for Scheme. Technical report, Rice University, Hoston, TX 77251-1892, May 1995.