

Purely Functional Monadic Scheme

Martin Grabmüller
Technische Universität Berlin

Draft: October 14, 2003; last modification: June 15, 2004

Abstract. This paper describes a purely functional dialect of the algorithmic programming language Scheme. Mutable variables, assignment operations and impure control operations (*call-with-current-continuation*) have been omitted, while side effecting input and output operations are incorporated into the purely functional language using monads, similar to the approach taken in the lazy programming language Haskell.

1 Introduction

This paper describes an experiment in language design and implementation. The goal was to create a purely functional dialect of the programming language Scheme [1]. For this experiment, Scheme has been restricted to its purely functional parts, followed by the introduction of monadic input/output operations.

Restricting programming languages to purely functional (and therefore referential transparent) constructs has the advantage that the meaning of programs becomes independent of evaluation order (if one disregards termination [6]), because there are no side-effects which impose a specific order on the control flow of the program to be executed correctly. Thus a compiler for such a language has much more freedom in rearranging computations in order to increase efficiency, for example by parallel evaluation of expressions. Unfortunately, all programming languages need some mechanism for performing side effects, otherwise they would not be able to communicate with other programs, humans, or systems. Lazy programming languages, such as Haskell [5], therefore need to re-introduce some mechanism to express sequential, side-effecting operations for the interaction with the “external world”.

This paper is organized as follows: in Sect. 2, we briefly describe the language Scheme, pointing out the impure language components on the way. Section 3 presents how to convert Scheme into a purely functional language, first by removing all impure features, then by adding monadic side-effects in order to re-establish the most of the functionality which was removed in the first step. Section 4 reviews related work and draws a conclusion.

2 Scheme

Scheme [1] is a member of the Lisp family of programming languages. It uses fully parenthesized prefix notation both for function application and for control and declaration constructs, so-called special forms. Scheme is dynamically typed (types are associated with values, not

with variables) and provides several builtin data types, such as pairs (lists), vectors, numbers, characters, strings and booleans.

A Scheme program consists of a sequence of declarations and expressions. A declaration has the form

```
(define x exp)
```

and declares a location (named *x*) which contains the value of the expression *exp*. Expressions are variables *x*, constants 1, "text", #t, '(1 a "a") '#((1 2 3) #\b), function abstractions (**lambda** (*x*) *x*), function applications (+ 1 2), conditionals (**if** (> *x* 2) 'a 'b), assignments (**set!** *x* 1) and recursive variable bindings (**letrec**).

In addition to these expressions, Scheme defines a number of derived forms for easier writing and reading of Scheme code. These forms are defined in terms of the primitive ones and can be syntactically transformed into them.

Scheme uses eager evaluation, so that function arguments are completely evaluated before functions are applied. Variables are statically scoped and functions are first-class values which can be arguments and results of functions and get stored into data structures.

3 Purely Functional Scheme

The conversion into a pure language consists of two steps: (1) removing any imperatively side-effecting operations, and (2) adding them back in some referentially transparent way.

Taking away any harmful side-effecting operations is easy: simply remove their definitions. In most Scheme systems, this can be done by assigning a new value (e.g. #f) to them:

```
(set! set-car! #f)
(set! set-cdr! #f)
...
(set! close-output-port #f)
```

The special form **set!** is also dangerous, but it cannot be **set!** in some Scheme implementation, because it is syntax, not a procedure. Often, it is possible to define new syntax with the **set!** keyword, so we will do that.

```
(define-syntax set!
  (syntax-rules ()
    ((set! x y) (scheme-report-environment -1))))
```

This definition will make the assignment statement unusable because it will be translated into the form (*scheme-report-environment* -1). This procedure call will cause a run-time error because the procedure is not defined for the argument -1.

The second step of the transformation into purely functional Scheme is accomplished by using monads to express side-effecting operations. This approach has been taken in several purely declarative languages, such as in the functional language Haskell [5] or the logic language Escher [3].

As suggested by Wadler [8], monadic computations in Purely Functional Scheme will be represented as functions from the state of the world to a value and the changed state of the world. So the general form of such a computation will be:

```
(lambda (world) ... (cons value world') ...)
```

(The dots ... indicate some calculation, depending on the computation to perform.)

The first step to adding monadic computation to Scheme is to provide the minimal functions for handling them within the language. First, we need a function which creates a monadic action which simply returns a value. Influenced by Haskell, this is called *return*.

```
(define (return value)
  (lambda (world)
    (cons value world)))
```

The function *return* lifts a value into a computation yielding exactly that value.

Monads need another operation, called *bind*. Again, we will use the Haskell name, $>>=$ for defining this combinator. It takes a monadic action and a function taking the result of the action and returning an action, and returns an action which defines the composition of the two arguments:

```
(define (>>= m f)
  (lambda (world)
    (let ((pair (m world))) ;; perform m
      (let ((value (car pair)) ;; extract value...
            (world! (cdr pair))) ;; and changed world
        ((f value) world!)))))) ;; run f
```

In order to form a monad, the two operations *return* and $>>=$ need to fulfill the monad laws:

1. $(>>= (return\ v)\ (\lambda\ (x)\ m)) = m[x:=v]$
2. $(>>=\ m\ (\lambda\ (x)\ (return\ x))) = m$
- 3.

```
(>>= m (\lambda (x) (>>= n (\lambda (y) o)))) =
(>>= (>>= m (\lambda (x) n)) (\lambda (y) o))
```

In line 1, $m[x:=v]$ means m with each free occurrence of variable x replaced by the term v , x may appear free in m . In the last line, variable x may appear free in term n but not in term o , and variable y may appear in term o .

It is common to define at least one further monadic combinator, $>>$, which sequentially combines two monadic action, discarding the result of the first (but not its possible changes to the world!).

```
(define (>> m1 m2)
  (lambda (world)
    (>>= m1 (\lambda (-) m2)) world)))
```

Now we are ready to define a simple application of these combinators: the state monad. Its purpose is to maintain a hidden state variable and to provide operations for reading and writing it.

```
(define (get)
  (lambda (world)
    (cons (cdr world) world)))
(define (set value)
```

```
(lambda (world)
  (cons '() (make-world value))))
```

In order to test this state monad, we need the definition of the auxiliary function *make-world*, which was used in the *set* function, and a procedure which, when given a monadic action, executes it and extracts the value it returns, discarding the value representing the world at the end of the computation. We will call that function *run*.

```
(define (make-world state)
  (cons 'world state))
(define (run m)
  (let ((pair (m (make-world #t))))
    (car pair)))
```

With these definitions, we can already program with our monadic computations. The following definition declares the monadic computation *main-1*, which sets the state to the value 1, retrieves the value from the state and returns the value increased by 1.

```
(define main-1
  (>> (set 1)
    (>>= (get)
      (lambda (x)
        (return (+ x 1))))))
```

Using the expression

```
(run main-1)
```

the computation can be invoked and returns the value 2.

Note that nowhere in the definition of *main-1* the value representing the world was mentioned. This fact makes it possible to optimize the operations *set* and *get* by using (hidden) side-effecting operations.

```
(define *state* #f)
(define (get)
  (lambda (world)
    (cons *state* world)))
(define (set value)
  (lambda (world)
    (set! *state* #f)
    (cons '() world)))
```

In this version, we use **set!** again (of course, we cannot hide its definition like we did above), but we do it in a referential transparent way. The reason is that the monadic computations are composed in a way which makes sure that they happen in the correct order, without interfering with each other. By implementing the state as an array, an association list or a hash table, we could maintain more than one value in the state. For simplicity, we do not add this complication in this paper.

As a last step, we could even optimize the *world* argument away, because it is not used anywhere except for passing it from one computation to another. This requires the re-definition of our basic combinators, of course. Below all the functions defined so far are shown in the optimized variant. (The *make-world* function is not necessary anymore.)

```

(define (return value)
  (lambda ()
    value))
(define (>>= m f)
  (lambda ()
    (let ((value (m)))
      ((f value))))))
(define (>> m1 m2)
  (lambda ()
    (>>= m1 (lambda (-) m2))))
(define *state* #f)
(define (get)
  (lambda ()
    *state*))
(define (set value)
  (lambda ()
    (set! *state* #f)
    '()))
(define (run m)
  (let ((result (m)))
    result))

```

Now that we have the machinery for constructing computations and plumbing them together, we want to integrate all of Scheme's standard input/output operations into our monadic framework.

For each imperative procedure we have to construct a monadic variant. To simplify this, we write a higher-order function *monadify* which takes a procedure and returns a monadic action encapsulating this procedure.

```

(define (monadify proc)
  (lambda args
    (lambda ()
      (apply proc args))))

```

Now it is easy to create monadic counterparts of all the side-effecting procedures defined for Scheme.

```

(define m-open-input-file
  (monadify open-input-file))
(define m-close-input-port
  (monadify close-input-port))
(define m-open-output-file
  (monadify open-output-file))
(define m-close-output-port
  (monadify close-output-port))
(define m-read-char
  (monadify read-char))
(define m-read (monadify read))
(define m-display (monadify display))

```

```

(define m-write (monadify write))
(define m-newline (monadify newline))
...
(define m-vector-set! (monadify vector-set!))

```

(Note again, that these definitions must be made before the side-effecting procedures are hidden.)

We have not considered the function *call-with-current-continuation*, which captures the remainder of a program and lets the programmer store it in variables and re-use it. We simply undefine it and let the implications of integrating it into a monadic Scheme for future research.

In principle, we have everything we need for purely functional monadic programming in Scheme, but since writing down combinations of monadic computations is tedious, we make use of the macro capabilities of Scheme for defining an abbreviation form. The syntax `>>*` combines the `>>=` and `>>` functions with a binding facility for computation results. It is basically the same as the `do` notation in Haskell, but we needed to chose a new name because the keyword `do` is already used in Scheme.

```

(define-syntax >>*
  (syntax-rules (<-)
    ((>>* (<- x m) rest ...)
     (>>= m (lambda (x) (>>* rest ...))))
    ((>>* m)
     m)
    ((>>* m1 m2 ...)
     (>> m1 (>>* m2 ...))))))

```

The following test programs are intended to illustrate how to program in purely functional Scheme.

The first program opens a file called "monadic.scm", reads an expression from that file and prints it to the terminal. Only the monadic combinator `>>=` the monadic input/output procedures defined above are used.

```

(define main-2
  (>>= (m-open-input-file "monadic.scm")
    (lambda (port)
      (>>= (m-read port)
        (lambda (x)
          (m-write x))))))

```

In the following example, the syntactic form `>>*` is used for formulating an interactive program. After reading an expression from a file, the user is prompted to enter an expression. After that, the expression read from the file and the one input from the user are output. This example shows, how the declaratively formulated program is sequentially executed.

```

(define main-3
  (>>* (<- port (m-open-input-file
                "monadic.scm"))
    (<- x (m-read port))
    (m-display "Expr: "))

```

```

(<- y (m-read))
(m-close-input-port port)
(m-write x)
(m-newline)
(m-write y)
(m-newline)))

```

The last example is an adaptation of an example taken from Wadler's paper [8]. It illustrates how monadic computation can be abstracted just like values. In exactly the same way as values, computations can be bound to variables.

```

(define main-4
  (let ((x (m-display "ha")))
    (>>* x
      x
      (m-newline))))

```

4 Conclusion

In this paper, we have shown how an impure functional language can be transformed into a pure language, by restricting it to its purely functional subset and introducing the possibility to perform side effects in a referentially transparent way. The restriction poses no problems, since only one special form (**set!**) and all side-effecting standard procedures have to be taken out.

The problem with monadic computations in Scheme is that there is no type system which can help in formulating monadic combination functions. In Haskell, the type system can pinpoint very subtle programming errors, whereas in Scheme, often cryptic run-time errors result.

Kiselyov has written a usenet article [2] about the use of monadic programming techniques in Scheme for introducing some kind of laziness into the language. He considers monads as a programming technique rather than a program construct. In contrast to the present article, monads are not explicitly introduced there, but only the underlying principles are exploited.

Semmelroth and Sabry [7] use monads to integrate side effects into a purely functional subset of ML [4]. Because of the missing macro capabilities in ML, their work requires changes to the ML languages, whereas we can convert any standard Scheme implementation into a purely functional one, simply by redefining some of the standard procedures and syntax.

What remains to be investigated is what changes would be needed to transform Scheme into a lazy language, so that the freedom of evaluation order could be completely exploited. This would result in a Lisp-like language with semantics similar to languages like Haskell.

The interaction of *call-with-current-continuation* and our purely functional language needs to be investigated, too.

References

- [1] Richard Kelsey, William Clinger, Jonathan Rees, et al. Revised⁵ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(6):26–76, September 1998.

- [2] Oleg Kiselyov. Monads, scheme, and io. World Wide Web, March 1998. <http://okmij.org/ftp/Scheme/misc.html#monadic-io>, last visited: 2003-10-15.
- [3] J. W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.
- [4] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1997. Revised edition.
- [5] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, April 2003. Also available from: <http://www.haskell.org/definition/>, last visited: 2003-06-23.
- [6] Amr Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22, January 1998.
- [7] Miley Semmelroth and Amr Sabry. Monadic encapsulation in ML. In *International Conference on Functional Programming*, pages 8–17, 1999.
- [8] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):140–263, September 1997.