A Newcomer's Impressions of Scheme

Gregory V. Wilson
Department of Mathematics and Computer Science
Vrije Universiteit, Amsterdam
gvw@cs.vu.nl

In July and November of 1993 I posted a summary of my first impressions of Scheme to the Internet news group `comp.lang.scheme`. The articles engendered a great deal of follow-up, from which I learned a great deal more about the language. I am grateful to everyone who commented on the original postings, and hope that this article will stimulate as much discussion as they did.

Throughout this work, I have relied primarily on Aubrey Jaffer's SCM. Aubrey has been extremely helpful in answering questions, and in adding engine-like interrupts to SCM. I would also like to thank Matthias Blume for several informative discussions, and for adding engine-like interrupts to VSCM.

1 What I Used Scheme For

I used Scheme between October 1992 and December 1993 to write a compiler and simulator for parallel dialects of an Algol-like language. The aim of this work was to produce tools with which to teach parallel programming. This project was my introduction to Scheme, and, except for six intensive months of Prolog while doing an M.Sc. in Artificial Intelligence in 1986, my first significant use of a non-Backus language (i.e. one outside the Fortran and Algol families).

I chose to simulate an Algol-like language so that my example programs would not appear too foreign to programmers trained in conventional imperative languages. I did consider using Scheme as the base language for this work, but felt that trying to teach engineers list-oriented programming and parallelism at the same time would be a bit much.

My two major design criteria at the beginning of this project were:

1. Execution had to be reproducible: if a program generated an error, it had to generate the same error each time it was run. I wanted students to be able to spend most of their time thinking about parallelism, rather than waiting for a timing-dependent bug to manifest itself again.

2. The whole system had to be small enough to run comfortably on PCs

and Macintoshes as well as Unix workstations, so that it could be used in colleges and by students studying at home.

1.1 The Compiler

My base language contained the usual loop, conditional, and procedure-call constructs, and has strongly-typed scalar variables (Boolean, fixed-point, floating point, and string), arrays, and record structures. The parallel dialects I implemented were:

data parallelism: whole-array operations, parallel conditionals (the "where" construct), and reduction/scan operations;

"traditional" shared variables: spawn (rather than fork), semaphores, and barriers;

futures: including competing evaluators to support speculative parallelism;

generative communication: the Linda model;

CSP: Occam-style channels, guarded input, and parallel blocks; and

procedural message-passing in the spirit of PVM, Express, CHIMP, etc.

The compiler consisted of a tokenizer, a parser, a checker (which resolved variable references, checked type consistency, etc), and a code generator. Rather than trying to maintain dialect-specific versions of the compiler (see Section 2.3), I controlled its behavior using conditional flags.

1.2 The Run-Time System

The compiler's output was Scheme, and made calls to a generic run-time library and one of several dialect-specific run-time libraries. The generic library's functions emulated multi-tasking, managed variables, performed some minimal tracing, and supported the language's control structures. Some of its salient features were:

- Processes were represented as continuations. There were managed by a single process management object using `call/cc`. The fact that Scheme has `call/cc` was, in fact, one of the main reasons I chose to use it: the alternative would have been to write a simple virtual machine, and compile code for it, but I felt that most of what I wanted was already to be found in Scheme.
- Scalar variables were represented as pairs containing a property (used to indicate constants, or variables with futures in progress) and an atomic value. Such "boxing" was needed so that variables could be write-shared

between processes, or copied to create separates workspaces. Arrays and records were represented as vectors of other variables, with appropriate headers (i.e. the record's type or the array's dimensions and dope vector).

- Functions and variables were defined directly in Scheme, rather than being stored in association lists. I used the latter method in earlier versions in order to support symbolic debugging. However, the run-time overheads proved unacceptable; this is a point to which I return in Section 2.7.

- The checking phase of the compiler determined when arguments to intrinsic operators needed to be unwrapped, and inserted appropriate code. Thus, an expression like:

```
a[3] := b * c * cos(d) + e;
```

was translated into:

```
(scl:set! (arr:elt-ref a 3)
  (+ (scl:ref e)
    (* (scl:ref b) (* (scl:ref c) (cos (scl:ref c))))))
```

Earlier versions of my system always worked with boxed variables, i.e. boxed the output of every operator, and expected the arguments of every operator to be boxed. Eliminating this dramatically improved performance (not least by reducing the frequency of garbage collection), at a cost of some extra complexity in the compiler.

2 Scheme Itself

2.1 I Got Used to the Parentheses

Like most programmers trained in von Neumann languages, I initially found Scheme's parenthesization confusing. It took me several weeks to adjust, but I no longer find it difficult to read (my own) Scheme code. I still disagree strongly with the claim that Scheme's syntax is simple ("it's all just lists"). I believe that Scheme actually provides less syntactic support for programmers than most other languages, since many things which are not usually thought of as lists (such as `let` statements) appear to be. Scheme's `do` construct is even uglier than C's `for`, and special forms such as `let` could benefit from some of the extra infix syntax found in many other functional and dataflow languages:

```
(let (a 1)
     (b 2)
 in
  (foo a b)
  (bar b a))
```

A precedent already exists for this, in the special => form of **cond** statements.

2.2 Special Forms Should Be Syntactically Distinct

While on the subject of special forms, I understand that:

```
(apply or list-of-booleans)
```

is illegal because **or** is a special (short-circuiting) form, but I still think of **or** as a function, and routinely find myself trying to **apply** it. Similarly, when first learning Scheme I told myself that a **define** statement looks like a function, so:

```
(map define '(a b c) '(1 2 3))
```

ought to be legal. Making keywords that indicate special forms syntactically distinct from other keywords in some way may offend purists, but it might save beginners (such as myself) some head-scratching.

2.3 Version Management

I found it difficult to manage the different, but related, versions of my software. In C, I use **#if/#else/#endif** to conditionally include or exclude code sections; in Scheme, I had to choose between:

```
(if (eq? (SYSTEM) 'Unix) ...)
```

and finding a hackaround using **cpp(1)** or something similar. The former proved uncomfortable, since many of the things I wish to make conditional are most naturally expressed as case branches:

```
(case property
  ((no-prop)    #t)
  ((constant)   'error)
  ((array)      map-array) ;-- only in data-parallel dialect!!
  (else         #f))
```

I can of course translate this into:

```
(cond
  ((eq? property 'no-prop)              #t)
  ((eq? property 'constant)             'error)
  ((and (eq? *Dialect* 'datapar)
        (eq? property 'array))          map-array)
  (else                                 #f))
```

but I resent having to pay the run-time cost of checking the dialect each time, particularly when the operation is a low-level one, such as fetching the value of a scalar.

Aubrey Jaffer suggested that Common Lisp-style read macros could easily be added to SCM, but at the time I did not want to become reliant on any particular implementation of Scheme. (I have since decided that I will have to rely on a particular non-standard feature, namely engines, in order to get reasonable speed. *C'est la vie...*) I therefore wrote a simple pre-processor using `cpp(1)` and `sh(1)` to handle C-style directives such as:

```
(case property
  ((no-prop)            #t)
  ((constant)           'error)
#if DIALECT_DATAPAR
  ((array)              map-array)
#endif
  (else                 #f))
```

I feel that Scheme's lack of support for multi-version programming is its single greatest weaknesses. While some respondents fell that small languages, intended for small projects, do not need (or should not have) such facilities, I have found them extremely useful. I have, for example, both functional and macro definitions of low-level utilities in a single file, and can switch between them safely with minimal effort.

One could also argue (and indeed several people did, when this article was posted to `comp.lang.scheme`) that conditional compilation or inclusion is a Bad Thing, and that it would be better to isolate variant cases as separate functions. I agree that directives such as

```
#if DATAPAR && !NESTED && (!VMS || (SVR4 && !SUN))
```

are a bit much, but trapping the array case in the previous example using a separate conditional isn't very elegant either.

2.4 Name Hiding and Modularization

Scheme is only the second language I have worked with which does not support any level of naming between local and global; the first was FORTRAN-66. Since my first posting of an early version of this article to `comp.lang.scheme`, I have become aware of the contention that surrounds the issue of packages, modules, objects, and types in Scheme.

2.5 Continuations Aren't Quite First Class

Most of the Scheme textbooks and papers I have read to date state that Scheme has first-class continuations, the implication being that they are "things" to

which all reasonable operations (e.g. inclusion in a list) can be applied. I regard I/O as a reasonable operation, and was frequently frustrated when debugging by being unable to write out a continuation and then read it back in. Aubrey Jaffer and Mikael Pettersson (among others) pointed out one of the many difficulties associated with this, using example such as:

```
(let ((counter 0))
  (let ((inc1 (lambda ()                ; (fetch&add counter 1)
                (set! counter (+ counter 1))
                (- counter 1)))
        (inc2 (lambda ()                ; (fetch&add counter 2)
                (set! counter (+ counter 2))
                (- counter 2))))
    (write-arbitrary file inc1)         ; save inc1 function
    (write-arbitrary file inc2)         ; save inc2 function
    (reset-file file 0 'read)           ; close and reopen
    (let ((new-inc1 (read-arbitrary file))
          (new-inc2 (read-arbitrary file)))
      ...operations using new-inc1 and new-inc2...)))
```

Here, operations using `inc1` and `inc2` affect one another, since they share the same counter. But what about `new-inc1` and `new-inc2`? My intuition is that they should share a counter as well, but arranging for this to happen would be very difficult. On the other hand, if I alias a list within a vector using:

```
(define the-list '(a b))
(define the-vector (vector the-list the-list))
```

and then write it out, and read it back in with:

```
(with-output-to-file "some-file"
  (lambda () (write the-vector)))
(define the-result
  (with-input-from-file "some-file" read))
```

then the sharing in the original has also been lost. In this situation, I would be happy with a New Jersey solution that did 90% of what I wanted, 90% of the time. After all, what I can't display, I can't debug.

2.6 Vectors and Strings Aren't Quite First-Class Either

Scheme is not even-handed in its treatment of its aggregate data types. In fact, those about whose structure implementations know the most — vectors and strings — are not as well supported as lists, badly-formed instances of which are very easy to create. While `vector-for-each` and `string-for-each`

were easy enough to write, they would clearly be more efficient if they were implemented as intrinsics.

And while I'm on my soapbox, why does Scheme include vectors, rather than multi-dimensional arrays? The former are just a special case of the latter, and in most other areas Scheme has provided general mechanisms in preference to restricted ones. When I first made this comment, several respondents replied that a multi-dimensional array (MDA) is just a vector of vectors (of vectors...). A "real" MDA allows any plane (e.g. any row or column in a 2-dimensional array) to be selected with equal ease; clearly, this is not the case with a vector-of-vectors structure.

2.7 Profiling and Debugging

The first complete version of my compiler handled approximately 15 lines per second; the last version was an order of magnitude faster. I believe there was still room for much more improvement, but finding out where was a problem. I implemented a simple profiling system using macros which re-bind function names and record entry and exit times. I am not satisfied with it, as it cannot record time spent in anonymous functions, and the recording overheads involved undoubtedly distort the results a great deal. I realize that profiling a language which supports anonymous functions, and which relies on tail recursion, is more difficult than profiling one in which all functions are named as well as declared, but I'd be willing to settle for another New Jersey solution. After all, what I can't profile, I can't optimize.

Debugging raises many of the same naming issues, and is complicated by Scheme's tail-recursiveness. I very much want to provide a symbolic debugger for use with my system, but am not sure how to go about constructing one now that I am using `let` and `define` directly to create functions and variables, rather than building assoc lists.

2.8 Engines

In order to emulate the race conditions which bedevil multi-tasking systems, my system has to be able to switch contexts at a very fine- grained level, e.g. after reading the values of `a` and `b`, but before dividing or reading `c` in:

```
a := a/b + c;
```

Inserting calls to a count-and-interrupt function after each statement was therefore not enough; intrinsic operations themselves had to be instrumented, as in:

```
(define scl:ref (lambda (scl) (tick!) (cdr scl)))
```

or (more efficiently):

```
(defmacro scl:+ (x y)
 `(begin (tick!) (+ (scl:ref ,x) (scl:ref ,y))))
```

where `tick!` is given by:

```
(define tick! (lambda ()
  (set! *Step-Counter* (- *Step-Counter* 1))
  (if (= 0 *Step-Counter*)(begin
    (set! *Step-Counter* (random-value))
    (call/cc (lambda (continuation)
      (enqueue! *Process-Queue* continuation)
      ((dequeue! *Process-Queue*) 'dummy-argument)))))))
```

As Aubrey pointed out, this greatly increases execution time, as every operation involved a Scheme-level function lookup. At my request, Matthias Blume kindly added an intrinsic counter to VSCM, which counts s-expression evaluations and invokes a user-defined handler when the count is exhausted. Aubrey has added similar functionality to SCM, and I am very grateful to them both. I would be very interested to hear of other uses of such ticking.

2.9 A Final Plea

Please, can we have constants and enumerations? Several Scheme implementations support `define-integrable`, or something similar; other languages have found these useful, and I'm sure I'm not the only one who'd like to see constants as part of the standard. I'd also like to be able to enumerate a family of related constants for use as vector and list indices using:

```
(enum color red green blue)
```

rather than:

```
(define color-red 0)
(define color-green 1)
(define color-blue 2)
```

3 Closing Remarks

Developing the compiler and simulator described in Section 1 was certainly easier in Scheme than it would have been in C, C++, or something similar. I have been pleasantly surprised by the performance of the Scheme systems I have used; like most programmers used to von Neumann languages, I had believed that higher-level languages were very slow, and I am glad to learn that this is not necessarily the case. However, I believe the language is weakened unnecessarily by its lack of support for multi-version programming, name-hiding, and performance optimization: in short, by its lack of support for software engineering.