Scheme@33

[Extended Abstract]

William D Clinger
Northeastern University
will@ccs.neu.edu

ABSTRACT

Scheme began as a sequential implementation of the Actor model, from which Scheme acquired its proper tail recursion and first class continuations; other consequences of its origins include lexical scoping, first class procedures, uniform evaluation, and a unified environment. As Scheme developed, it spun off important new technical ideas such as delimited continuations and hygienic macros while enabling research in compilers, semantics, partial evaluation, and other areas. Dozens of implementations support a wide variety of users and aspirations, exerting pressure on the processes used to specify Scheme.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: General

General Terms

design,languages,standardization

Keywords

Lisp,Scheme,Actor model

1. INTRODUCTION

Lisp's notation for functions came from the lambda calculus. In 1975, Gerald Jay Sussman and Guy Lewis Steele Jr returned to those roots by adopting the applicative (call-by-value) lambda calculus's semantics for Scheme, their new dialect of Lisp.

Scheme's foundation in the lambda calculus provides simplicity and power. Scheme demonstrated elegant solutions to several of Lisp's outstanding problems. A few of those solutions (lexical scoping and first class procedures, also known as closures) were later adopted by Common Lisp. Scheme contributed to the maturation and growing acceptance of functional programming languages, and Scheme's influence can also be seen in languages such as Java, C#, Python, Ruby, and Javascript/ECMAscript.

This retrospective reviews the central ideas of Scheme, traces some of those ideas to their origins in the lambda calculus and the Actor model, lists some of Scheme's many technical contributions, and outlines the evolution of Scheme to the present day.

2. POWER FROM SIMPLICITY

In retrospect, Scheme is the result of fusing Lisp with a certain minimalist philosophy or attitude [12]:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

For example, Scheme solved Lisp's funarg problem [11] by replacing dynamic scoping with lexical scoping and by removing all restrictions on the dynamic extent of variables and objects.

> Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

For example, there are only five distinct kinds of expression in core Scheme:

$$
\begin{array}{rcl}
\mathit{Exp} & ::= & \mathit{Id} \\
& ::= & \texttt{(lambda (}\mathit{Id}\ \ldots\texttt{)}\ \mathit{Exp}\texttt{)} \\
& ::= & \texttt{(}\mathit{Exp}\ \ldots\ \texttt{)} \\
& ::= & \texttt{(if}\ \mathit{Exp}\ \mathit{Exp}\ \mathit{Exp}\texttt{)} \\
& ::= & \texttt{(set!}\ \mathit{Id}\ \mathit{Exp}\texttt{)}
\end{array}
$$

The first three productions are those of the lambda calculus, extended to allow functions to accept an arbitrary number of arguments.

(The last of the five productions is necessary because Scheme variables are mutable. This is often regarded as a mistake in the design of Scheme; immutable variables combined

with mutable references (one-element vectors) would simplify Scheme even further without sacrificing power.)

In addition to expressions, Scheme allows variable and macro definitions to appear in certain contexts. In R6RS Scheme, a library syntax allows programmers to export selected definitions while hiding others.

From the start, Scheme's design has followed a simple strategy:

1. Get procedures right.
2. Add macros.
3. Everything else is just a library.

Scheme's modernization of procedures and introduction of reliable and powerful macro technologies have been among its main technical contributions.

Scheme's main shortcoming has been the limited number of portable libraries. After too many years, the SRFI process and R6RS libraries, both discussed in a later section, began to address this problem.

3. GETTING PROCEDURES RIGHT

Getting procedures right in Scheme was mostly a matter of making call-by-value lambda calculus into the core of a practical programming language. That involved the abandonment of several practices that had become part of Lisp's tradition: dynamic scoping, representing functions as lambda expressions, dynamic extent for lambda-bound variables, distinct sets of rules for evaluating operators and operands of procedure calls, a separate environment for procedure-valued variables, and pushing a stack frame for every procedure call.

More subtly, getting procedures right was also a matter of treating continuation-passing style (CPS) as a normal, even fundamental, style of programming, and of regarding the more usual direct style as a kind of syntactic sugar for CPS.

This retrospective will give examples to illustrate (1) the funarg problem, whose interest is primarily historical, and (2) proper tail recursion, whose importance is only now beginning to be appreciated by the programming languages community at large.

3.1 The Funarg Problem

Prior to Scheme, most dialects of Lisp tried to represent functions as lambda expressions or other code. The values of a lambda expression's free variables were obtained from a dynamic *environment* that mapped each variable to the argument to which a formal parameter that happened to share that variable's name had most recently been bound by a call, omitting bindings created by calls to procedures that had already returned. That stack-like binding semantics was easy to implement but led to the so-called *funarg* problem, in which the free variables of a lambda expression would occasionally map to the wrong value or to no value at all.

Consider, for example, the free reference to `x` in the lambda expression for `f` below. If the function `f` were represented by its lambda expression, with the dynamic environment supplying the value of `x`, then `f`'s reference to `x` would resolve to one of the lists bound by the other lambda expression's formal parameter of the same name, instead of resolving to its intended value: the scalar argument that was passed to `add-scalar-to-lists`. Consequently, the `+` procedure would complain that it doesn't know to add that list to a number.

```
;;; Given a scalar x and a list y of lists of
;;; numbers, returns a list that is like y except x
;;; has been added to every number.  Example:
;;;
;;; (add-scalar-to-lists 10 '((1 2) (3 4)))
;;; returns ((11 12) (13 14))

(define add-scalar-to-lists
  (lambda (x y)
    (let ((f (lambda (y) (+ x y))))
      (map (lambda (x) (map f x))
           y))))
```

In Scheme, however, procedures are objects in their own right, and are distinct from lambda expressions and other S-expressions. Scheme's procedures can be regarded as *instances* of lambda expressions, and are generally represented as *closures*, which are data structures that combine a procedure's code (typically a compiled form of the original lambda expression) with its *environment*, which furnishes the values of all variables that occur free within the procedure's code. Those free variables can be regarded as the *instance variables* of the procedure object.

Scheme, like most other programming languages since Algol, is *lexically* (or *statically*) scoped, so all references to `x` in the example above resolve to the intended arguments. This provided an elegantly simple and completely general solution to the funarg problem that had bedevilled most previous dialects of Lisp.

3.2 Proper Tail Recursion

All of Scheme's control structures macro-expand into a very small core language that consists of a call-by-value lambda calculus extended with the McCarthy conditional. Scheme's viability therefore depends upon the efficiency with which loops and other common control structures can be translated into recursion.

The key to Scheme's efficiency is *proper tail recursion*, which is an unfortunate name for the asymptotic space complexity achieved by creating continuations intelligently. Proper tail recursion is a simple idea, easily implemented, but almost all implementations of almost all programming languages waste asymptotic space by allocating continuations unnecessarily.

Allocating a continuation frame for every procedure call wastes asymptotic space. Proper tail recursion is more efficient because continuations are allocated only for statically *nested* procedure calls. When a procedure's code contains a

call that is not nested within another call, and is not nested within any other syntactic construct for which a new continuation would have to be created, then the call occurs in a *tail context*[1] and is said to be a *tail call*. The continuation for a tail call is always equivalent to the continuation that was passed as an implicit argument to the procedure that contains the tail call, so it would be wasteful to allocate new continuations for tail calls.

Scheme mandates proper tail recursion, as do most other functional and higher order languages. Scheme was the first dialect of Lisp to mandate proper tail recursion, and appears to have been the first programming language for which proper tail recursion was defined with mathematical precision [9, 5].

Proper tail recursion is just as important for object-oriented programs as for functional programs, even though few (if any) object-oriented languages offer proper tail recursion.

The absence of proper tail recursion in object-oriented languages forces programmers to write ugly code. For an example, consider the problem of adding immutable lists, modelled on those of Lisp, to a language like Java. Standard design recipes would give us an abstract base class, say `FList`, with concrete subclasses `Empty` and `Pair`. To implement a `contains` method modelled after the `member` procedure of Lisp/Scheme, we would declare `contains` as an abstract method within `FList` and add the following definitions to the `Empty` and `Pair` classes:

```
class Empty extends FList {
    ....
    boolean contains (Object x) { return false; }
}

class Pair extends FList {
    Object first;
    FList rest;
    ...
    boolean contains (Object x) {
        return first.equals(x) || rest.contains(x);
    }
}
```

That simple and elegant implementation would also be efficient if Java were properly tail recursive. Since Java currently makes no provision for proper tail recursion, however, Java programmers must rewrite the tail-recursive definition of `contains` into something like

```
boolean contains (Object x) {
    FList lst = this;
    while (lst instanceof Pair) {
        if (((Pair) lst).first.equals(x))
            return true;
        else lst = ((Pair) lst).rest;
    }
    return false;
}
```

As can be seen, Java's failure to guarantee proper tail recursion forces programmers to write ugly code that violates the Law of Demeter [10] and other generally accepted rules of style.

4. ORIGIN OF SCHEME

Several of Scheme's main ideas come from Carl Hewitt's Actor model of concurrent computation, which made use of continuation-passing style, proper tail recursion, and other ideas taken from research on formal semantics and lambda calculus.

In the fall of 1975, Guy Steele was a new graduate student at the MIT Artificial Intelligence Laboratory, where he became interested in Hewitt's Actor model. Unfortunately, the syntactic complexity of Hewitt's Plasma language seemed to obscure rather than to clarify the model's main ideas. Professor Gerry Sussman suggested that he and Steele could improve their understanding of the Actor model by implementing a simple interpreter for sequential actors, using a more Lisp-like syntax. They did so. When they were done, they noticed that the code for creating an actor looked just like the code for lambda expressions, and the code for invoking an actor looked just like the code for calling a closure. They concluded that actors are just procedures, and that actor scripts are just lambda expressions.

Why hadn't that been obvious before? Because contemporary dialects of Lisp evaluated operators differently from operands and did not guarantee proper tail recursion. Had Steele and Sussman followed Lisp tradition instead of allowing their interpreter's code to be influenced by the Actor model, the code for creating an actor would have been different from the code for lambda expressions, and the code for invoking an actor would have been different from the code for calling a closure.

Hence Steele and Sussman's interpreter was not just Lisp: It was Lisp done right. Uniform evaluation of operators and operands implied a unified (Lisp1) environment. The independence of computational actors, which Hewitt had stressed for reasons of concurrency, implied lexical (static) scoping. The Actor model's treatment of direct style as syntactic sugar for continuation-passing style, with syntactic operators for converting between the two styles, implied proper tail recursion.

Following in the tradition of Planner and Conniver, Steele and Sussman decided to call their new language *Schemer*. The "r" was soon dropped because the MIT AI Lab's Incompatible Timesharing System (ITS) limited file names to six characters.

[1] In a conditional expression, for example, the test's continuation is not the same as the continuation for the entire conditional expression, so the test's context is not a tail context. The "then" and "else" expressions have the same continuation as the entire conditional expression, however, so their contexts are tail contexts if and only if the entire conditional expression is in a tail context.

5. TECHNICAL CONTRIBUTIONS

Beginning with their original description of Scheme in December 1975, Steele and Sussman co-authored a set of about ten "lambda papers" that explained what they had learned from inventing, implementing, and improving the Scheme language. These lambda papers include a revised report on Scheme in January 1978 [8] and Steele's master's thesis in May 1978, which described his innovative RABBIT compiler that used CPS as its primary intermediate language [7].

The simplicity and power of Scheme have made it extremely useful for research, and hundreds of technical papers that use or describe Scheme have now been written. Jim Bender's excellent online bibliography of research related to Scheme [3] lists the following major categories: the original lambda papers by Steele and Sussman, language features and semantics, macros, object-oriented programming, modules and component-oriented programming, continuations and continuation-passing style, XML and web programming, applications, compiler technology and implementation techniques, distributed, parallel, and concurrent programming, partial evaluation, and reflection.

Those research results include the discovery of delimited continuations, efficient strategies for implementing first class continuations, formal specifications of Scheme's core semantics, rigorous proofs of compiler correctness, development and practical applications of CPS and A-normal form, higher-order and object-oriented flow analysis (kCFA), soft typing, contracts, accurate conversions between binary and decimal, and hygienic macro expansion.

6. EVOLUTION OF SCHEME

Scheme became popular among educators and researchers in the early 1980s, and was being used as a general purpose programming language by the late 1980s. Scheme's simplicity soon contributed to an abundance of implementations, creating portability issues that have been addressed, with varying degrees of success, by the standardization efforts outlined below.

Standardization has been impeded by the competing needs and desires of researchers (who insisted upon keeping the core simple), educators (who advocated a wide variety of features that were believed to have pedagogical value), and programmers, whose own needs were quite diverse. Programmers of embedded systems wanted an extremely small language in which even floating point arithmetic would be optional; those who were using Scheme for prototyping wanted fancy programming environments and large libraries; those who were using Scheme for scripting prized access to external resources; and those who used Scheme for systems programming often valued efficiency to a degree that mystified prototypers and script-writers.

6.1 The RnRS Authors

By the early 1980s, Hal Abelson and Gerry Sussman were using Scheme to teach the freshman CS course at MIT. Their textbook [2] and the MIT implementation of Scheme had already diverged from Steele and Sussman's revised report of January 1978 [8].

Meanwhile at least five other implementations of Scheme were well under way: T, Scheme 84, PC Scheme, Mac-Scheme, and Chez Scheme. As these implementations continued to diverge, their implementors and users began to find it difficult even to read Scheme code written for other implementations. In October 1984, fifteen representatives of the then-major implementations met at Brandeis University to work toward a better and more widely accepted standard for Scheme. Their report, the *Revised Revised Report on Scheme*, was published in the summer of 1985 [6]. Another round of revision resulted in the *Revised*^{3} *Report on the Algorithmic Language Scheme* in 1986 [12].

The authors of these reports, known as the R*RS (or RnRS) authors, continued to propose and to debate potential changes by electronic mail and at occasional meetings. In 1988, the R*RS authors met again at the ACM Conference on Lisp and Functional Programming, just a few days before the first meeting of the working group that developed the IEEE-1178 standard discussed below. This and other meetings resulted in the *Revised*^{4} *Report* [4], which was developed in parallel with the IEEE standard.

The rule of consensus that had been agreed upon at Brandeis soon morphed into a rule of unanimity: no significant changes could be made unless all of the self-selected and self-perpetuating R*RS authors agreed to the change. By the time the *Revised*^{5} *Report* was published in 1998 [9], it was so hard to secure unanimous agreement for new proposals that few authors felt it was worthwhile to agitate for change.

6.2 IEEE Standard 1178-1990

The Scheme Working Group of the Microprocessor and Microcomputer Standards Subcommittee of the IEEE began its work in 1988. The working group was chaired by Professor Chris Haynes of Indiana University. IEEE Standard 1178-1990 was edited by David H Bartley, Chris Hanson, and Jim Miller, and "drew heavily" on the *Revised*^{3} *Report* and drafts of the *Revised*^{4} *Report* [1].

IEEE Standard 1178-1990 became an ANSI standard via a routine process whose announcement in an obscure ANSI publication went entirely unnoticed.

6.3 Recent Developments

A new standardization process began in 2003, under a Scheme Language Steering Committee whose original members were Alan Bawden, Guy Steele, and Mitch Wand. That committee selected a committee of seven editors (later reduced to five, and then temporarily reduced to four) that began work on two documents that were ratified in August 2007 by 65.7% of an electorate consisting of about 100 voters.

These new standards are the *Revised*^{6} *Report on the Algorithmic Language Scheme* and the *Revised*^{6} *Report on the Algorithmic Language Scheme — Standard Libraries* [13]. Two auxiliary documents were produced by the editors but were not subject to ratification.

R6RS Scheme bears approximately the same relation to R5RS (and IEEE/ANSI) Scheme that Algol 68 bears to Algol 60: the two languages have almost the same expressions, but there is no overlap between programs. The semantics of R5RS/IEEE/ANSI Scheme is designed to support interac-

tive read/eval/print loops with dynamic loading of libraries. R6RS Scheme is designed for batch execution of whole programs, and forbids interactive read/eval/print loops by requiring the entire program to pass certain static checks before any part of the program is allowed to begin its execution.

Within a year after ratification of the R6RS, two implementations of R5RS Scheme (Larceny and PLT) had added R6RS modes to their R5RS modes of operation. Four entirely new implementations of R6RS had been developed, with no support for the R5RS or previous standards; one of those (Ypsilon) had been completed and two others (Ikarus and IronScheme) were complete enough to run some R6RS programs. On the other hand, eleven implementations of Scheme had announced their intention to remain within the R3RS/R4RS/R5RS/IEEE/ANSI tradition. Although portability of Scheme programs was an explicit goal of the R6RS, it is not yet clear whether its net effect will be to improve or to degrade portability.

6.4 Libraries

Scheme's main weakness has been a shortage of portable libraries. Although several implementations have accumulated impressive sets of libraries that work with only one implementation, only three collections of libraries have been successful enough to promote portability between implementations:

- SLIB: `http://people.csail.mit.edu/jaffer/SLIB`
- SRFI: `http://srfi.schemers.org/`
- R6RS: `http://www.r6rs.org/`

Of these three, the SRFI (Scheme Request For Implementation) project has specified the most widely accepted libraries, but implementations vary widely in their support for SRFI libraries. All of the core SLIB libraries are supported by every implementation that supports SLIB, and all of the standard R6RS libraries are supported by the three completed implementations of the R6RS.

7. CONCLUSIONS

Many of Scheme's once-radical ideas have become influential, and Scheme remains a significant advance over several of the languages it has influenced.

As a research language, Scheme has been wildly successful. As a teaching language, Scheme endures. As a programming language, Scheme has been hampered by weak standards and inadequate libraries, and by the incompatibilities that proliferated as each major implementation promoted its own extensions and libraries. Despite its problems, Scheme has been more successful and retains more vitality than most other 33-year-old programming languages.

8. REFERENCES

[1] *IEEE Standard for the Scheme Programming Language*. IEEE Standard 1178-1990. IEEE, 1991.

[2] H. Abelson and G. J. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[3] J. Bender. Bibliography of Scheme-related Research. `http://library.readscheme.org/`

[4] W. Clinger and J. Rees [editors]. The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3):1–55, 1991.

[5] W. D. Clinger. Proper tail recursion and space efficiency. In *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, pages 174–185. ACM, June 1998.

[6] W. Clinger [editor]. The revised revised report on Scheme, or an uncommon Lisp. *MIT AI Memo 848*, August 1985.

[7] G. L. Steele Jr. Rabbit: a compiler for Scheme. *MIT AI Memo 474*, May 1978.

[8] G. L. Steele Jr and G. J. Sussman. The revised report on Scheme, a dialect of Lisp. *MIT AI Memo 452*, January 1978.

[9] R. Kelsey, W. Clinger, and J. Rees [editors]. The revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998.

[10] K. J. Lieberherr. Formulations and benefits of the Law of Demeter. *ACM SIGPLAN Notices*, 24(3):67–78, March 1989.

[11] J. Moses. The function of FUNCTION in Lisp or why the funarg problem should be called the environment problem. *ACM SIGSAM Bulletin*, 15:13–27, July 1970.

[12] J. Rees and W. Clinger [editors]. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.

[13] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten [editors]. Revised^6 report on the algorithmic language Scheme. `http://www.r6rs.org/`