A Security Kernel Based on the Lambda-Calculus

by

Jonathan Allen Rees

S.M. Comp. Sci., Massachusetts Institute of Technology (1989)
B.S. Comp. Sci., Yale College (1981)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Signature of Author ....
Department of Electrical Engineering and Computer Science
January 31, 1995

Certified by ....
Gerald Jay Sussman
Matsushita Professor of Electrical Engineering
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

A Security Kernel Based on the Lambda-Calculus

by

Jonathan Allen Rees

Revised version of a thesis submitted to the
Department of Electrical Engineering and Computer Science
on January 31, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Cooperation between independent agents depends upon establishing a degree of security. Each of the cooperating agents needs assurance that the cooperation will not endanger resources of value to that agent. In a computer system, a computational mechanism can assure safe cooperation among the system's users by mediating resource access according to desired security policy. Such a mechanism, which is called a *security kernel*, lies at the heart of many operating systems and programming environments.

The dissertation describes Scheme 48, a programming environment whose design is guided by established principles of operating system security. Scheme 48's security kernel is small, consisting of the call-by-value λ-calculus with a few simple extensions to support abstract data types, object mutation, and access to hardware resources. Each agent (user or subsystem) has a separate evaluation environment that holds objects representing privileges granted to that agent. Because environments ultimately determine availability of object references, protection and sharing can be controlled largely by the way in which environments are constructed.

I will describe experience with Scheme 48 that shows how it serves as a robust and flexible experimental platform. Two successful applications of Scheme 48 are the programming environment for the Cornell mobile robots, where Scheme 48 runs with no (other) operating system support; and a secure multi-user environment that runs on workstations.

Thesis Supervisor: Gerald Jay Sussman
Title: Matsushita Professor of Electrical Engineering

Acknowledgements

Joseph Weizenbaum, for generously allowing me the use of his office during the long final stretch.

Gerry Sussman of 4AI, Bruce Donald of Cornell, and Gregor Kiczales of Xerox PARC, for moral and financial support, guidance, and ideas.

Hal Abelson and Tom Knight, for serving on my committee.

Richard Kelsey, for hard work and heroic deeds that helped to bring Scheme 48 to life.

Norman Adams and Will Clinger, for productive collaborations.

Alan Bawden, David Espinosa, Dave Gifford, Philip Greenspun, Ian Horswill, Kleanthes Koniaris, Dave McAllester, Jim O'Toole, Brian Reistad, Bill Rozas, Mark Sheldon, Olin Shivers, Franklyn Turbak, and many others at Tech Square, for making life at MIT not just tolerable but interesting and even amusing.

Russell Brown, Jim Jennings, Daniela Rus, and others at the Cornell Computer Science Robotics and Vision Laboratory, for their good humor and courageous robotry.

Kathleen Akins, Mike Dixon, John Lamping, Brian Smith, and others at Xerox PARC, for creating an environment in which peculiar ideas flourish.

Butler Lampson, for his dazzlingly clear explanations.

Hal Abelson, Oakley Hoerth, Frans Kaashoek, Tom Knight, Mark Sheldon, Gerry Sussman, and Franklyn Turbak, for their comments and suggestions on drafts of the document.

Marilyn Pierce, for making the administrative nightmares go away.

Rebecca Bisbee, for making everything work.

Albert Meyer, for urging expedience.

Tom Collett, for giving me a reason to finish.

Laura Burns, Katy Heine, Kris Jacobson, and Geoff Linburn, for their generosity and emotional support in strange times.

ARPA and NSF, for grants to the projects that employed me.

Others too numerous to mention.

Finally, Oakley Hoerth, for the diverse array of model arthropods: glow-in-the-dark cicada, wind-up beetle, pop-up and puppet ants, cricket stickers, and many others.

Contents

Chapter 1

Introduction

Cooperation between independent agents depends upon establishing a degree of security. Each of the cooperating agents needs assurance that the cooperation will not endanger resources of value to that agent. In a computer system, a computational mechanism can assure safe cooperation among the system's users by mediating resource access according to desired security policy. Such a mechanism, which is called a *security kernel*, lies at the heart of many operating systems and programming environments.

My thesis is that the λ-calculus can serve as the central component of a simple and flexible security kernel. The dissertation supports this thesis by motivating and describing such a λ-calculus-based security kernel and by giving several lines of evidence of the kernel's effectiveness.

The W7 security kernel consists of the call-by-value λ-calculus with a few simple extensions to support abstract data types, object mutation, and access to hardware resources. Within W7, each agent (user or subsystem) has a separate evaluation environment that holds objects representing privileges granted to that agent. Because environments ultimately determine availability of object references, protection and sharing can be controlled largely by the way in which environments are constructed.

The effectiveness of W7 as a security kernel is demonstrated through three lines of evidence:

1. its ability to address certain fundamental security problems that are important for cooperation (Sections 2.2–2.3);

2. a structural correspondence with familiar operating system kernels (Section 2.4); and

3. the success of Scheme 48, a complete implementation of the Scheme programming language built on W7, as a basis for secure, robust, and flexible programming systems (Chapter 3 and the Appendixes).

1.1 Security Kernel Based on λ-calculus

The λ-calculus is a calculus of functions, and is concerned with how computations are abstracted and instantiated and how names come to have meanings. W7 is a λ-calculus of *procedures,* which are generalized functions capable of performing side effects. Procedures correspond to what in an operating system would be programs and servers. Side effects include access to input and output devices and to memory cells. It is through side effects that communication, and therefore cooperation, is possible. However, side effects can be harmful. For example, a computer-controlled robot arm can easily hurt a person who gets in its way.

The purpose of a security kernel is to allow control over access to objects. The challenge in designing a security kernel is not to support sharing or protection per se, but rather to allow flexible control over the extent to which an object is shared or protected. One way in which the λ-calculus and W7 provide protection is through *closure*: a procedure is not just a program but a program coupled with its environment of origin. A procedure cannot access the environment of its call, and its caller cannot access the procedure's environment of origin. The caller and callee are therefore protected from one another. Sharing is accomplished through shared portions of environments, which may include procedures that allow still other objects to be shared.

To address a number of authentication and certification problems, W7 includes an abstract data type facility. (Usually this term refers to type abstraction enforced through compile-time type checking, but here it means a dynamic information hiding mechanism.) The facility is akin to digital signatures: a subsystem may sign an object in such a way that the signed object may be recognized as having been definitely signed by that subsystem. In particular, a compiler might use a particular signature to mean that the signed procedure is one that is "harmless" (in a technical sense) and is therefore safe to apply to fragile arguments.

1.2 Scheme 48

Scheme 48 is a complete Scheme system that tests W7's capacity to support safe cooperation. Scheme 48 was a major design and implementation effort and therefore constitutes the heart of the thesis project. Chapter 3 gives an overview of Scheme 48, but most of the information on it is to be found in the appendixes.

A large amount of engineering goes into making a practical programming environment, and in Scheme 48 security has been a concern in nearly every component. Major facilities whose design has been shaped by security concerns include the following:

- The module system (Appendix C). Modules are truly encapsulated, just as procedures are, allowing them to be shared safely.

- The macro facility (Appendix D). Macros are also closed, like procedures. This allows a form of compile-time security in which a module may export a macro while protecting objects used in the macro's implementation.

- Dynamic variables. Information can be communicated from caller to callee through an implicit dynamic environment. However, a dynamic variable must be accessed via a key, and such keys can be protected.

A major theme running through the design of Scheme 48 is avoidance or minimization of shared global state. For example, the virtual machine (byte-code interpreter) has only an essential minimum set of registers; there are no registers that hold global symbol tables or environment structure as there is in most Lisp and Scheme implementations. Another example is in the run-time system modules, which never alter global state (in Scheme terms, no top-level variable is ever assigned with `set!`). Data structures manipulated by these modules can be instantiated multiple times to avoid conflict over their use.

Finally, the success of Scheme 48 (and therefore of W7) is demonstrated by its use in a number of applications. These include the programming environment for the Cornell mobile robots, where Scheme 48 runs with no (other) operating system support (Appendix B), and a secure multi-user environment that runs on workstations.

Chapter 2

The Security Kernel

This chapter is an exposition, starting from first principles, of the problem of secure cooperation between independent agents. It describes a simple idealized security kernel that addresses this problem. The presentation is intended to show the essential unity of security concerns in operating systems and programming languages.

The idealized kernel is based on a graph of encapsulated objects. Accessibility of one object from another is constrained by the connectivity of the object graph and further limited by object-specific gatekeeper programs. The kernel enables the natural construction of a variety of mechanisms that support secure cooperation between agents. In particular, an agent can securely call an untrusted agent's program on a sensitive input.

The kernel is similar to those of the capability-based operating systems of the 1970's [38, 70]. The main differences are that this kernel is simpler, more abstract, and more clearly connected with programming language concepts than are classical capability systems.

2.1 Safe Computer-Mediated Cooperation

The participants in a cooperative interaction carry out a joint activity that uses or combines the resources that each provides. Resources might include energy, information, skills, or equipment.

Each agent relinquishes control to some extent over the resources that the agent brings. If the agent values a resource, relinquishing control over it is dangerous, because the resource may come to harm or may be used to cause harm.

To assure a resource's safety when control over it is relinquished, it is desirable for an agent to be able to dictate precisely the extent to which control is relinquished. A trusted intermediary can be helpful in this situation. The resource is handed over to the intermediary, who performs actions as specified by the recipient subject to restrictions imposed by the source.

A computer system may be useful in a cooperation not only because of the resources it may provide (programmable processor, memory, network communications, programs, and so on), but also because of its potential to act as a trusted intermediary. The agent providing a resource can dictate use restrictions in the form of a program, and the resource's recipient can specify actions to be performed with the resource in the form of a program.

Program invocation therefore plays a critical role in computer-mediated cooperation. But when a program is untrusted (as most programs should be!), invoking it is fraught with peril. The invoking agent puts at risk resources of value, such as inputs and the use of parts of the computer system. Similarly, the agent who supplied the program may have endowed it with (access to) resources of value to that agent, so those resources are also put at risk when the program is invoked.

Consider the following scenario:

> Bart writes a program that sorts a list of numbers. Being a generous fellow, he gives this useful program to Lisa, who has expressed an interest in using such a program. But Lisa is hesitant to use Bart's program, since Bart may have arranged for the program to do sneaky things. For example, the program might covertly send the list to be sorted back to Bart via electronic mail. That would be unfortunate, since Lisa wants to sort a list of credit card numbers, and she would like to keep these secret. What should Bart and Lisa do?

There are two distinct approaches to solving the safe invocation problem — that is, the general situation in which each agent in a cooperation has resources that should be protected when one agent's program is invoked in a context (inputs and computer system) given by another agent. In the first approach, the program runs in a limited computing environment, one in which dangerous operations (such as sending mail) are prohibited. In the second approach, the second agent rejects the program unless it bears a recognized certificate of safety. The next two sections look at the two approaches.

2.1.1 Limited Environments

One way to avoid disclosure of the numbers (an instance of the *confinement problem;* see [36]) would be for Lisa to isolate a computer running Bart's program, making it physically impossible for the program to do any harm. This would require detaching the computer from the network, removing disks that should not be read or written, setting up a special network or disk for transmitting the inputs and results, and so on.

These measures are inconvenient, time-consuming, and expensive. They are also crude; for example, it should be permissible for Bart's program to fetch information from the network or disks, but not write information. This selective restriction would be impossible using hardware configurations.

The basic idea is sound, however. The trick is to ensure that all accesses to sensitive resources are intercepted by a special supervisor program. The supervisor program checks the validity of the access attempt and allows it to go through only if it should.

The most straightforward method by which this is achieved is through the use of an *emulator*. An emulator is a program that mimics the role that the processor hardware usually plays in decoding and executing programs. For instructions that are always safe, the emulator simulates the hardware precisely; for dangerous instructions, the supervisor program is invoked.

Use of an emulator makes programs run slowly, so a more common alternative is the use of a special hardware feature available on many computers called "user mode". When the processor is in user mode, all dangerous operations are preempted, causing the supervisor program to be invoked. The supervisor program runs in an ordinary (non-user or "supervisor") mode, so it may implement whatever protection policy it likes.

Here is how safe program invocation might be accomplished using an architecture with user mode support:

1. The invoker runs the program in user mode, telling the supervisor program which operations are to be permitted.

2. The supervisor program monitors all potentially dangerous operations, refusing to perform operations that are not permitted.

3. When the program is done, it executes a special operation to return to supervisor mode.

2.1.2 Screening and Certification

Another way to avoid disclosure of her list of numbers would be for Lisa to screen Bart's program to verify that it contains no dubious operations (such as calls to the send-email program). This would probably require her receiving the program in source form, not as a binary; the program would have to be of modest size and written in a language familiar to Lisa. She would also need access to a compiler and time on her hands to run the compiler. But even if Lisa were willing and able to go to this trouble, Bart might be unwilling to give the source program to her because it contains information that he wants to keep secret, such as passwords, material subject to copyright, trade secrets, or personal information.

This dilemma could be solved with the help of a trusted third party. There are many different ways in which a third party might help; the following is a variant of the screening method suggested above that doesn't require Lisa to obtain Bart's source program.

> Suppose that Bart and Lisa both trust Ned. Bart gives his source program to Ned with the instructions that Ned is not to give the source program to Lisa, but he may answer the question of whether the program is obviously harmless to run. ("Obviously harmless" could mean that the program uses no potentially harmful operations, or it could be a more sophisticated test. Because harmlessness is uncomputable, any such test will be an approximation.) Lisa obtains the machine program as before, and asks Ned whether the machine program is obviously harmless to run; if he says it is, she runs it with assurance.

In the screening approach, the agent invoking the program rejects it unless it bears a recognized certificate of safety. Following this initial check, invoking the program is just as simple as invoking any fully trusted program. There is no need for an emulator program or user mode hardware support.

Here is how safe program invocation might be accomplished using the screening approach:

1. The invoker checks to see whether the program is definitely restricted to operations permitted by the invoker.

2. The invoker refuses to run the program if it appears to use any unpermitted operations.

3. If not, the invoker runs the program on the real machine. Harmful operations needn't be prevented, since they won't occur (if the certifier is correct).

Screening has the drawback that some cooperation is required from the person supplying the program. He must go to the trouble of obtaining certification, and perhaps even change the program so that it is certifiable.

2.2 A Simple Kernel

A *security kernel* is the innermost layer of an operating system or programming language, the fundamental component responsible for protecting valued resources from unwanted disclosure or manipulation. Isolating the security kernel from the rest of the system helps to ensure reliability and trustworthiness. Simplicity is important in a kernel because the simpler the kernel, the more easily it can be tested and verified.

What follows is just one of many ways to define a security kernel. This design, which I'll call W7,[1] is intended to be suitable for use in either an operating system or a programming language. W7 might also be seen as a theory of security that can be used to describe security in existing programming languages and operating systems.

W7 organizes the computer's resources into a network of logical entities that I'll call *objects*.[2] Objects are the basic units of protection in W7. A link in the network from one object to another means that the second is accessible from the first. The objects to which a given object has access are its *successors,* and the objects that have access to a given object are its *predecessors.*

Objects may be thought of as representing particular privileges, capabilities, or resources. For example, an object might represent the ability to draw pictures on a display, to transmit messages over a communications line, to read information from a data file, or to run programs on a processing element.

New nodes enter the object network when a running program requests the creation of a new object. In order to avoid the dangers of dangling references (links to deleted

[1]The name "W7" was chosen with the help of a pseudo-random number generator.

[2]Unfortunately, "object" has become a loaded term in computer science. To many people it implies complicated phenomena such as methods, classes and inheritance. I don't mean to suggest any of this baggage. It may be best to take it as an undefined term, as is "point" in geometry.

nodes) and memory leaks, object deletion is left in the hands of the kernel. The kernel may delete an object as soon as there is no path through the network to it from the current computation. Absence of such a path is detected by an automatic mechanism such as reference counts or garbage collection.

There are several different kinds of objects. These will be introduced as they are needed in the presentation.

2.2.1 On the Choice of Notation

In order to elucidate the kernel design and describe its consequences, it will be necessary to present specimen programs, and for this purpose I must choose a notation — that is, a programming language. The choice is in principle arbitrary, but is important because it affects the exposition.

If I were most interested in comparing W7 to currently popular and familiar security kernels in operating systems, I would present my examples as programs that perform kernel operations in the traditional way, using system calls. Programs would be written in an Algol-like language, or perhaps in a machine-level language such as C. This would be a viable approach, since there is nothing in the theory that precludes it (see Section 4.3.3).

However, this approach makes some of the examples awkward to write and to understand. Particularly awkward is the frequent case where in creating a new object, a program must specify a second program. Most traditional languages don't have a good way for a program to specify another entire program. Even given such a notation, the division of labor between the language and the security kernel might be difficult to tease apart.

For this reason I prefer to use a notation in which kernel operations have natural forms of expression. The language of the Unix "shell" approaches this goal since program invocation has a natural syntax, and operations on programs and processes are more concise than they would be in Algol or C.

However, I would like to go a step further in the direction of languages with integrated kernel support, and use a version of Scheme [11] in which all Scheme values, including procedures, are identified with objects known to the W7 kernel. ML [42] would have been another reasonable choice of language.

2.2.2 On the Notation

To avoid confusion with other Scheme dialects, I'll refer to the small dialect of Scheme to be used in examples as "Scheme$^-$." The following grammar summarizes Scheme$^-$. E stands for an expression.

```
E ::= var
    | (lambda (var ...) E)
    | (E E ...)
```

```
    | constant
    | (if E E E)
    | (begin E ...)
    | (let ((var E) ...) E)
    | (let var ((var E) ...) E)

    | (arith E E)
    | (cons E E) | (car E) | (cdr E)
    | (null? E) | (pair? E) | (list E ...)
    | (symbol? E) | (eq? E E)
    | (new-cell) | (cell-ref E) | (cell-set!\ E E)
    | (enclose E E) | (control E E)
arith ::= + | - | * | / | < | = | >
```

The meaning of most of the constructs is as in full Scheme. The first group of constructs is Scheme$^-$'s λ-calculus core: variable reference, procedure abstraction (`lambda`), and application. Each of the second group of constructs (`if`, etc.) has a special evaluation rule.

The third group is a set of primitive operators. In every case all of the operand expressions are evaluated before the operator is applied. Most of these are familiar, but a few require explanation:

A *cell* is a mutable object with a single outgoing access link (field). `(new-cell)` creates a new, uninitialized cell, `(cell-ref` *cell*`)` returns the object to which *cell* currently has a link, and `(cell-set!` *cell obj*`)` redirects *cell*'s outgoing link to *obj*.

The `enclose` operator takes a program and a specification of a successor set and converts them to a procedure. `enclose` might be called on the output of a Scheme$^-$ or Algol compiler. Its details are unimportant. See section 2.2.4.

The `control` operator controls hardware devices. For example,

```
(control keyboard 'read-char)
```

might read a character from a particular keyboard. Its arguments are interpreted differently for different kinds of devices.

To reduce the complexity of the exposition, pairs will be assumed to be immutable, and `eq?` will be assumed to be applicable only to symbols and cells.

I'll use the term *value* to describe integers, booleans, symbols, and other entities that carry only information, not privileges. Whether or not values are considered to be objects is unimportant.

2.2.3 Procedures

The W7 kernel is principally concerned with *procedures.* Every procedure has an associated program. The procedure's program can access the procedure's successors and use them in various operations. Access is not transitive, however: access to a procedure does not imply the ability to access the procedure's successors. Any such access is necessarily controlled by the procedure's program. In this sense, the procedure's program is a "gatekeeper" that protects the procedure's successors.

The primary operation on a procedure is application to an argument sequence. When a procedure is applied to arguments (which may include other objects or values, such as integers), its program is run. In addition to the procedure's successors, the program has access to the arguments. It may use any of these objects, but no others, in making new objects or in performing further applications. Applications are written in Scheme$^-$ using the syntax

(*procedure* *argument* ...).

For example, if `f` names a link to a procedure, then `(f x y)` denotes an application of that procedure to `x` and `y`.

The essential source of security in W7 is that a procedure's program is *absolutely limited* to using the procedure's successors and the objects that are in the application's argument sequence.

Scheme$^-$ `lambda`-expressions are a concise notation for specifying procedures. When a program evaluates a `lambda`-expression, a new procedure is created. The new procedure's successors are the objects that are the values of the `lambda`-expression's free variables, and its program is specified by the body of the `lambda`-expression.

Here is an example illustrating `lambda` and application.

```
(lambda (x) (g (f x)))
```

specifies a procedure with two successors, which it knows as `f` and `g`. When the resulting procedure (say, h) is applied to an argument, it applies `f` to that argument, and then applies `g` to the result obtained from applying `f`. Finally, `g`'s result becomes the result of the call to h. h therefore acts as the functional composition of `f` and `g`.

An procedure's program could be written in a language other than Scheme$^-$, for example, Algol, C, or a machine language. I will consider this possibility later (Section 4.3.3).

2.2.4 Initial program

Suppose that someone — let's call her Marge — obtains a brand-new machine running W7 as its operating system kernel. She turns it on, and the machine begins executing an initial program. The initial program is an ordinary program with no special relation to the kernel other than that it is executed on power-up. In principle, the initial program is arbitrary; it is whatever the manufacturer has chosen to provide.

The initial program is given access to objects that represent all of the computer system's attached hardware resources, which might include keyboards, displays, facsimile machines, traffic lights, etc. (Hardware may be manipulated with the `control` primitive operator, whose details are not important here.) Because these resources are objects, any given program will only have access to those hardware resources to which it has been explicitly granted access by the initial program (perhaps indirectly through a series of other programs).

In order to make the computer as generally useful as possible, the manufacturer has installed an initial program that executes commands entered from the keyboard. One kind of command is simply a Scheme$^-$ expression. When an expression is seen, the initial program compiles and executes the expression and displays its result. For example, the command

```
(+ 2 3)
```

causes the number 5 to be displayed.

A second kind of command is a *definition*, which looks like

(define *var* *E*).

Definitions give a way for a user to extend the environment so that new objects can be given names that subsequent commands can see. For example, Marge can write

```
(define square (lambda (x) (* x x)))
```

to define a squaring procedure, and

```
(square 17)
```

to invoke it and display the result.

To be able to use existing resources, commands need to have access to them. For this reason they are processed relative to an *environment,* or set of variable bindings, allowing objects to be accessed by name. The environment initially includes two kinds of resources: I/O devices (with names like `the-display` and `the-fax-machine`), and handy utilities that the user could have written but the manufacturer has thoughtfully supplied. The utilities are the usual Scheme procedures such as `assoc`, `write`, `read`, and `string-append`, as well as a few others to be described below.

The initial program just described, including utilities and command processor, might have been written entirely in Scheme$^-$, in which case it would resemble the following:

```
(define command-processor
  (lambda (env source sink)
    (let loop ((env env))
      (let ((command (read source)))
        (if (definition? command)
            (loop (bind (cadr command)
                        (eval (caddr command) env)
                        env))
            (begin (write (eval command env) sink)
                   (loop env)))))))

(define definition?
  (lambda (command)
    (if (pair? command)
        (eq? (car command) 'define)
        #f)))
```

`eval` is a utility that executes a Scheme$^-$ expression, of which a representation has been obtained from the user or otherwise received or computed. It makes use of a compiler that translates the expression into a form acceptable to the kernel's primitive `enclose` operator. The translated expression is given access to a specified environment, and executed.

```
(define eval
  (lambda (expression env)
    ((enclose (compile-scheme- expression
                               (map car env))
              (map cdr env)))))
```

Definitions are handled using `bind`, which extends a given environment by adding a binding of a variable to a value. With environments represented as association lists, `bind` could be defined with

```
(define bind
  (lambda (name value env)
    (cons (cons name value) env))).
```

2.2.5 Administration

Now we have enough mechanism at our disposal to consider some examples.

Marge intends to set up her machine so that Lisa and Bart can use it and its resources. As a means for the machine's users to share objects with one another, she defines a simple object repository:

```
(define *repository* (new-cell))
(cell-set! *repository* '())

(define lookup
  (lambda (name)
    (let ((probe (assoc name *repository*)))
      (if probe (cdr probe) #f))))

(define publish!
  (lambda (name object)
    (cell-set! *repository*
      (cons (cons name object) *repository*))))
```

Anyone with access to Marge's `lookup` procedure can obtain values from the repository, while anyone with access to `publish!` can store values in the repository.[3] (`assoc` is assumed to be Lisp's traditional association list search function.)

[3] For simplicity, it is assumed that concurrency is not an issue. In a multiprocessing or multitasking system, access to the `*repository*` cell would have to be controlled by a lock of some kind.

Next, Marge makes an environment for Bart's command processor. It must include all of the objects that Bart is to be able to access initially. There is no harm in including all of the system's utility procedures (`assoc`, `read`, etc.), since no harm can come from his using them. However, she needs to be careful about the I/O devices. For the sake of safety, she includes only Bart's own I/O devices in `Bart-env`:

```
(define make-user-env
  (lambda (from-user to-user)
    (bind 'standard-input from-user
          (bind 'standard-output to-user
                (bind 'lookup lookup
                      (bind 'publish! publish!
                            utilities-env))))))
(define Bart-env
  (make-user-env from-Bart to-Bart))
```

Bart's command processor can now be initiated with

```
(command-processor Bart-env from-Bart to-Bart)
```

Similarly for Lisa's:

```
(command-processor Lisa-env from-Lisa to-Lisa)
```

Security results from the omission of vulnerable objects from users' initial environments. For example, Lisa's I/O devices (`from-Lisa` and `to-Lisa`) aren't accessible from Bart's environment, so Bart won't be able to cause any mischief by reading or writing them.

Bart and Lisa are able to share objects with each other through the repository:

```
Bart: (define really-sort
        (lambda (list-of-numbers)
          (if (null? list-of-numbers)
              '()
              (insert (car list-of-numbers)
                      (really-sort
                         (cdr list-of-numbers))))))
      (define insert
        (lambda (x l)
          (let recur ((l l))
            (if (null? l)
                (list x)
                (if (< x (car l))
                    (cons x l)
                    (cons (car l)
                          (recur (cdr l)))))))))
```

```
       (publish! 'sort sort)

Lisa:  (define Bart-sort (lookup 'sort))
       (Bart-sort '(9 2 7))  ⟶  '(2 7 9)
```

Of course, Bart and Lisa must both trust Marge, who still has full control over all resources. They must treat her administrative structure as part of the machine's trusted substrate. (Trust cannot be avoided. Bart and Lisa must trust Marge's software and machine just as Marge trusts the manufacturer's installed software, the manufacturer trusts the semiconductor chip factories, the chip factories trust the solid state physicists, and so on.)

2.2.6 Trusted Third Party Establishes Safety

There is already a great deal of safety built in to the structure of the W7 kernel. When a procedure is invoked, the only privileges the invoked procedure's program has are (1) those that it was given when created ("installed"), and (2) those that are passed as arguments. If the two privilege sets contain no dangerous privileges, then there is no way that any harm can come from the invocation. However, sometimes there is reason to pass privileges or sensitive information to an unknown program. This can be a problem if when installed the program was given access to channels over which the information might be transmitted, or to a cell in which a privilege or information may be saved for later unexpected use.

To return to the main example from the introduction: Lisa wants to call Bart's `sort` program without risking disclosure of the input, which may contain sensitive information. Specifically, the risk she runs is that Bart has written something like

```
(define *list-of-numbers* (new-cell))

(define sort
  (lambda (list-of-numbers)
    (begin (cell-set! *list-of-numbers* list-of-numbers)
           (really-sort list-of-numbers))))

(define really-sort
  (lambda (list-of-numbers)
    (if (null? list-of-numbers)
        '()
        (insert (car list-of-numbers)
                (really-sort (cdr list-of-numbers))))))
(publish! 'sort sort)
```

This deceit would allow Bart to extract the most recent input passed to `sort` by evaluating `(cell-ref *list-of-numbers*)`.

Sketch of a rudimentary solution:

1. Ned, who both Bart and Lisa trust, sets up a "safe compilation" service.

2. Bart submits the source code for his `sort` program to Ned's service.

3. Ned's service analyzes Bart's source code to determine whether the program might divulge its input to Bart or to anyone else. (For a definition of "might divulge," see below.)

4. Ned evaluates (for initialization) Bart's program in a fresh environment and extracts access to the resulting `sort` procedure.

5. Ned makes `sort` available to Lisa.

6. Lisa obtains the program from Ned, and runs it with assurance that the input will not be divulged.

Whether a program "might divulge" its input is undecidable, so any test for this property is necessarily conservative: such a test will reject perfectly benign, useful programs that can't easily be cast into a recognizably safe form.

There are many possible tests. One simple test is the following one: a program might divulge its input if the program is not obviously applicative; and a program is obviously applicative iff none of its `lambda`-expressions contains a `(cell-set! ...)` expression.

One can clearly do better than this; some methods will be discussed later (in the context of the module system, section 3.2).

To see how this safe compilation service might work, let's continue with the scenario begun above in which Marge has established a public object repository. The first problem is that Lisa needs a way to determine authorship of a repository entry, to distinguish Bart's entries from Ned's. Assuming that all objects published by Ned are safe, she must make sure that the `sort` procedure she uses is published by Ned, not by Bart.

This is a fundamental flaw in the repository, so Marge must fix it.

```
(define make-publish!
  (lambda (submitter)
    (lambda (name object)
      (publish! name (cons submitter object)))))

(define make-user-env
  (lambda (user-name from-user to-user)
    (bind 'standard-input from-user
          (bind 'standard-output to-user
                (bind 'lookup lookup
                      (bind 'publish!
                            (make-publish! user-name)
                            utilities-env))))))
```

```
(define Bart-env (make-user-env 'Bart from-Bart to-Bart))
(define Lisa-env (make-user-env 'Lisa from-Lisa to-Lisa))
(define Ned-env (make-user-env 'Ned from-Ned to-Ned))
```

Each object in the repository is now accompanied by the name of the user who put it there, and each user's environment now has its own `publish!` procedure. Bart's `publish!` procedure, for example, will store entries of the form `(cons 'Bart` *object*`)` into the repository.

Here is the complete scenario:

```
Ned:  (define publish-if-safe!
        (lambda (name program)
          (if (safe? program)
              (publish! name (eval program))
              'not-obviously-safe)))
      (define safe?
        (lambda (program) ...))
      (publish! 'publish-if-safe! publish-if-safe!)

Bart: (define sort-program
        '(begin (define sort ...)
                (define insert ...)
                sort))
      (define publish-if-safe!
        (cdr (lookup 'publish-if-safe!)))
      (publish-if-safe! 'safe-sort sort-program)

Lisa: (define safe-sort-entry (lookup 'safe-sort))
      (define safe-sort
        (if (eq? (car safe-sort-entry) 'Ned)
            (cdr safe-sort-entry)
            "safe-sort not published by Ned"))
      (safe-sort '(9 2 7))
```

There is nothing Bart can do to trick Lisa. If he tries to publish an unsafe `safe-sort` procedure, he will have to use his own `publish!` procedure to do so, since Ned's `publish!` isn't accessible to him. In this case, when Lisa does `(lookup 'safe-sort)`, the result will be marked with Bart's name, not Ned's, and Lisa will discard it.

2.3 Authentication

This section treats the problem of *authentication* and considers its solution in W7. Broadly speaking, authentication is any procedure or test that determines whether an object is trustworthy or genuine. For example, checking a student identification card authenticates the person presenting the card as being a student as indicated on the card. Having been found authentic, the person may be granted privileges appropriate to that status, such as permission to use an ice rink.

Authentication is an important capability of secure computer systems. Some examples of authentication in such systems are:

- A request received from an untrusted source such as a public communications network must be authenticated as originating from an agent that has the right to perform the action specified by the request.

- In a dynamically typed programming language such as Lisp or Snobol, a value must be authenticated as being of the correct type for an operator receiving it as an operand.

- The solution to the safe invocation example of Section 2.2 involves a test for the authenticity of a putatively safe or trustworthy object (Bart's program).

Authentication is necessary for reliable transmission of an object using an untrusted messenger (or channel). To authenticate an object transmitted from a sender to receiver, the sender must label or package the object so that it can be recognized (authenticated) by the receiver, and this labeling or packaging must be done in an unforgeable and tamper-proof manner. I'll use the word *capsule* for an object so labeled or packaged. A physical analogy for a capsule would be a locked box to which only the receiver has a key.

When the object is digital information, such as a sequence of characters or numbers, well-known digital cryptographic techniques can be used to implement authenticated transmission [50]. However, object transmission in a security kernel that limits object access according to an accessibility graph requires a different mechanism. With such a kernel, transmitting a name for an object, whether encrypted or not, is never the same as transmitting access to that object, since interpretation of names — and therefore accessibility of named objects — is local to each object. Access can only be transmitted through approved channels such as argument and result transmission in a procedure invocation.

2.3.1 Abstract Data Types

Authentication is required whenever the results of one procedure are intended to constitute the only valid inputs to another procedure. These objects, the results and valid inputs for a related set of procedures, are called the instances of a *type* or *abstract data type*. The set of related procedures is called a *module* or *cluster* in the context of programming languages, or a *protected subsystem* or *server* in classical operating systems parlance.

For example, consider a central accounting office that issues account objects to clients. Clients may create new accounts using a `new-account` procedure and transfer money between accounts using a `transfer` procedure. The integrity of accounts is of great importance to the accounting office; it would be unfortunate if a client created a counterfeit account and then transferred money from it into a valid account. Therefore, `transfer` needs to authenticate the accounts it's asked to manipulate as genuine — that is, created by `new-account`.

From the clients' point of view, accounts are objects that clients may manipulate a certain set of operators. From the accounting office's point of view, accounts are capsules transmitted from `new-account` to `transfer` via an untrusted network of interacting clients.

2.3.2 Key-based Authentication

Section 2.2.6's solution to the safe invocation problem relies on the exchange of object *names.* Bart tells Lisa the name of his object in the trusted repository (`safe-sort`), and Lisa obtains from the repository (1) the fact that the object is authentically safe (i.e. placed there by Ned), and (2) the object itself, which Lisa can then use.

If Bart wishes to keep his object a secret between him and Lisa, he may give it a secret key in the form of an unguessable name, i.e. a password, and transmit that name to Lisa. Passwords can be made long enough and random enough to make the probability of unauthorized discovery arbitrarily small. (Of course, this assumes that there is no way for untrusted agents to obtain a list of all names defined in the repository.)

Instead of passwords, it is possible to use cells (Section 2.2.2) as keys, as suggested by Morris [44]. Like passwords, cells are objects with unique, recognizable identities. Unlike passwords, they have all the security advantages of objects: access to them can only be transferred through kernel-supported operations.

There is no need to use a single repository, such as Marge's, to store all objects that might need to be authenticated. Each separate abstract data type can have its own mapping from keys to objects.

Below is an implementation of a general authentication facility (essentially the same as in [44]). Each call to `new-seal` returns a new triple of objects ⟨*seal, unseal, sealed?*⟩. Each such triple represents a new abstract data type. *seal* encloses an object in a capsule that can be authenticated; *unseal* is an extraction operator that reveals the encapsulated object iff the capsule is authentic (i.e. was made by this *seal*); and *sealed?* is a predicate that returns true iff the capsule is authentic.

It is assumed that `assq` (association list lookup) can determine cell identity. Cells (other than the one holding the association list) are used only for their identities, not for holding references to objects.

```
(define (new-seal)
  (let ((instances (new-cell)))
    (cell-set! instances '())
    (let ((seal
           (lambda (rep)
             (let ((abs (new-cell)))
               (cell-set! instances
                          (cons (cons abs rep)
                                (cell-ref instances)))
               abs)))
          (unseal
           (lambda (abs)
             (let ((probe
                    (assq abs (cell-ref instances))))
               (if probe
                   (cdr probe)
                   (error "invalid argument" abs)))))
          (sealed?
           (lambda (x)
             (if (assq x (cell-ref instances))
                 #t
                 #f))))
      (list seal unseal sealed?))))
```

As an example, here is an implementation of the accounting system described above:

```
(define account-operators (new-seal))
(define make-account (car account-operators))
(define account-cell (cadr account-operators))
(define account? (caddr account-operators))

(define new-account
  (lambda ()
    (make-account (new-cell 0))))
```

```
(define transfer
  (lambda (amount from to)
    (let ((from-cell (account-cell from))
          (to-cell   (account-cell to)))
      (let ((new-balance
              (- (cell-ref from-cell) amount)))
        (if (>= new-balance 0)
            (begin
              (cell-set! to-cell
                         (+ (cell-ref to-cell) amount))
              (cell-set! from-cell new-balance))
            (error (error "insufficient funds")))))))

(publish! 'new-account new-account)
(publish! 'transfer transfer)
```

It's very important *not* to publish `account-cell`, since doing so would allow clients of the accounting system to set account balances to arbitrary values.

2.3.3 The Case for Kernel Support

As seen above, existing kernel mechanisms (cells, procedures) can be used to implement authentication. However, I would like to argue in favor of a direct authentication mechanism implemented in the W7 kernel. An argument in favor of this is required due to the stated goal of keeping the kernel as simple as possible.

A key-based authentication mechanism has several practical problems.

- Performance: It's inefficient to have to search the central table on every use of an instance.
- Memory management: How can the kernel know whether it is safe to delete an object? Straightforward garbage collection techniques don't work because the central table holds links to all of the type's instances, and the garbage collector is ignorant of the association between the key and the object.
- Semantic obscurity: It would be unfortunate if creating an object of an abstract data type were necessarily a side effect, as it would be if a key had to be generated. The side effect defeats optimizations such as subgraph sharing.

A direct approach, not involving a keyed table, requires kernel support. The following argument shows that in the W7 kernel without abstract data type support, a recipient can be fooled no matter what packaging method is used.

> Consider a candidate encapsulation and authentication technique. Without loss of generality, we can assume that capsules are procedures. In order to authenticate a capsule, the recipient has no choice but to apply it to some argument sequence, since there is nothing else that can

> be done with a procedure. This application can appropriately be called a "challenge," since examining its result distinguishes authentic capsules from nonauthentic ones. If the challenge is correctly met, then an additional application of the capsule (or of some related object returned by the challenge) will then either return or perform some operation on the underlying object.
>
> But given an authentic capsule, an untrusted messenger may easily make a counterfeit, as follows: the counterfeit answers challenges by consulting the authentic capsule and returning what it returns, and handles other applications in any way it likes.

This argument doesn't even consider the problem that the challenge might diverge, signal an error, or cause troublesome side effects. The annoyance of handling these situations (see section 4.3.1) would argue against using procedures, even were there a way to do so.

Kernel support can be provided in W7 by adding `new-seal` as a new primitive operator:

E `::= (new-seal)`

Capsules are a new kind of object that can be implemented efficiently. A capsule can be represented as a record with two fields, one containing the unique seal, and the other holding the encapsulated object.[4]

2.4 Protection Domains

This section attempts to explain the correspondence between W7 and more conventional operating system kernels.

In a typical secure operating system, the objects that are immediately accessible to a running program constitute its *protection domain* (or simply *domain*). These objects are of various sorts, including the following (there may be others):

1. Mapped memory segments. These contain both programs and the data stuctures that programs directly manipulate.

2. Descriptors for hardware devices and files. Descriptors control access to devices and the file system, and may include information such as buffers or position markers.

---

[4]It is an inelegant aspect of this design that there are two distinct encapsulation mechanisms (procedures and capsules). There are various ways to remedy this; for example, instead of introducing capsules as a separate kind of object, `lambda` and application could be extended to take unique markers, with the requirement that a procedure's marker must match the marker specified in an application of the procedure:

`(lambda` *marker* `(`*var* `...)` *body*`)`
`(applicate` *marker procedure arg* `...)`

See Appendix F for a different unification.

3. In some operating systems, references to procedures residing in other domains (variously called *gateways, inter-process communication handles,* or *remote procedure handles*); I will write simply *handle.* A handle consists of a domain together with an address within that domain specifying the procedure's entry point.

The program refers to objects using short names (numbers). For example, a load or store machine instruction uses a memory address to access a memory segment object, while a system call instruction requesting that information be read from a file uses a file descriptor number to specify the file.

The domain determines how these names are to be interpreted. The domain contains a segment table or page table for use by the hardware in interpreting memory addresses, and tables of descriptors and handles for use by system call handlers in interpreting I/O and cross-domain procedure call requests.

Just as a domain maps a name (address or descriptor number) to an accessible object (memory segment or descriptor), an environment in W7 maps a name (identifier) to its denotation (value or object). The various object types are parallel in the two frameworks as well (see figures 2-1 and 2-2):

1. Mapped memory segments correspond to lists and cells. (Full Scheme also has vector and string data types, which are better approximations to memory segments than are lists and cells. A more complete version W7 might support memory segments directly.)

2. Device and file descriptors correspond to W7 devices.

3. Handles correspond to W7 procedures. Just as a handle is a domain coupled with an entry point, a procedure is an environment coupled with executable code.

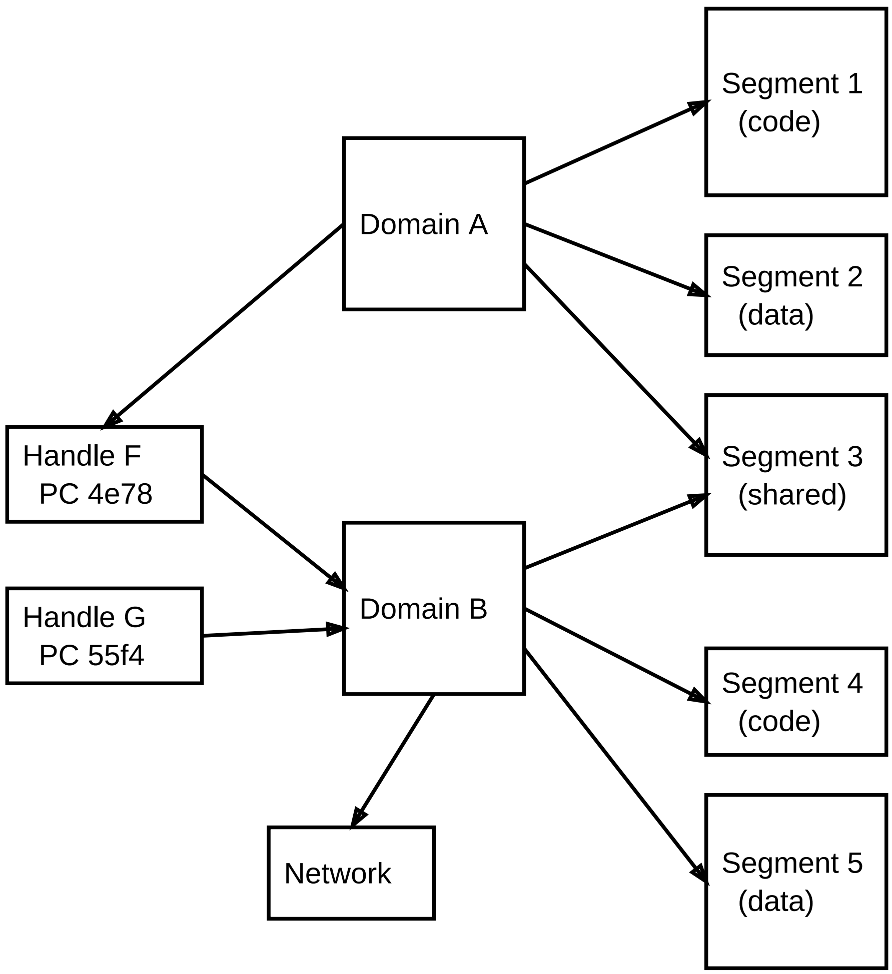

Figure 2-1: In an operating system, accessibility of resources is controlled by kernel data structures called *protection domains* (or simply *domains*). A domain includes a segment table or page table that specifies which memory segments are directly accessible by a program running in that domain. A domain also specifies availability of other resources, such as I/O devices (here represented by a network) and procedures residing in other domains. (References to such procedures are called *remote procedure handles* or *inter-process communication handles* in the literature.)

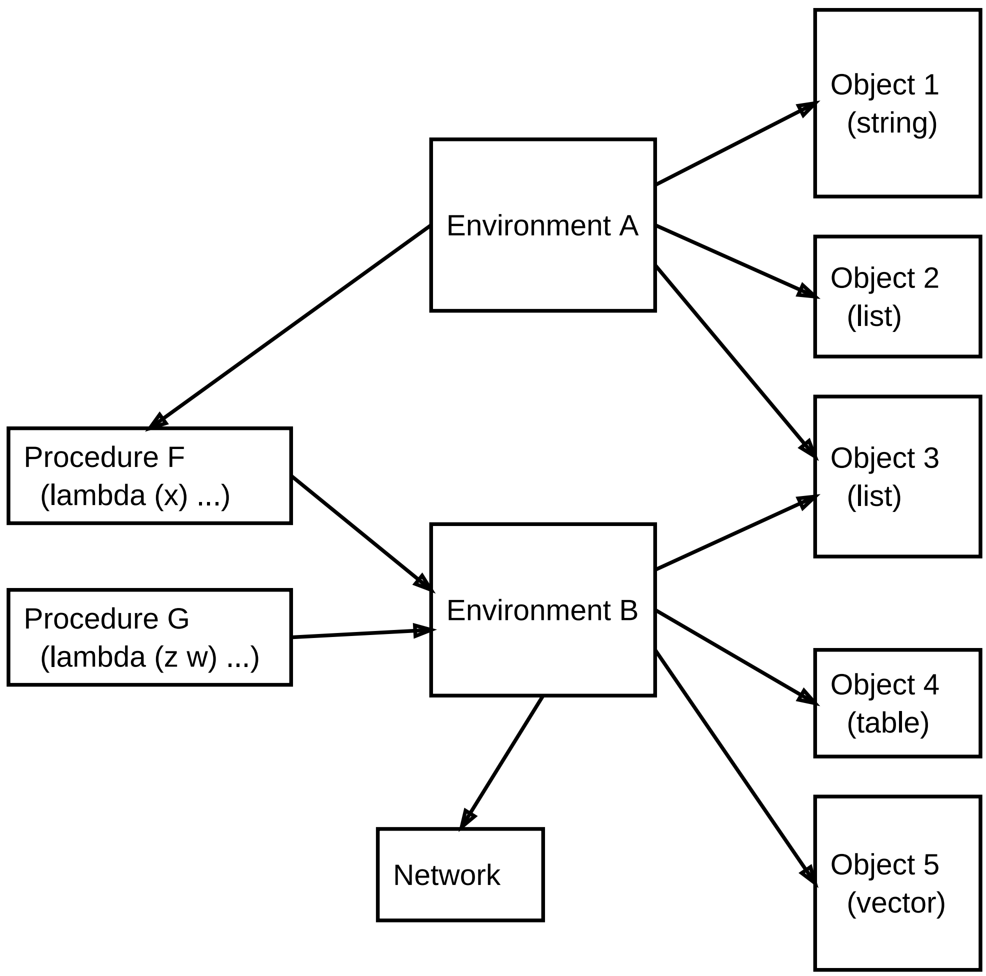

Figure 2-2: In a λ-calculus interpreter, accessibility of resources is controlled by interpreter data structures called *environments*. An environment contains references to data structures, such as lists, that are directly accessible to a program running in that environment. An environment also specifies availability of other resources, such as I/O devices (here represented by a network) and procedures connected to other environments.

Chapter 3

Implementation Experience

A variant of the W7 kernel forms the basis of a complete Scheme implementation called Scheme 48. This chapter describes Scheme 48, analyzes it from a security standpoint, and discusses experience with its use in various applications in which security is important.

The implementation of Scheme 48 consists of two parts: a virtual machine that manages memory and executes a byte-code instruction set, and a set of modules that are executed by the virtual machine. Virtual machine code could in principle be replaced by real machine code, but this would require a significant compiler construction effort, and that is beyond the scope of the project at this time. The virtual machine approach has the virtues of robustness, small size, and ease of portability.

(See Appendix A for a more thorough presentation of Scheme 48 as a Scheme system.)

3.1 Scheme 48 Security Kernel

Scheme 48's security kernel is implemented in part by the virtual machine and in part by a set of run-time system modules defined using privileged instructions supplied by the virtual machine. The VM component includes instructions such as `car`, which is secure because the VM raises an exception if it is applied to anything other than a pair, and application, which raises an exception when given a non-procedure. The run-time modules include the exception system and the byte-code compiler. Both of these export interfaces whose use is unrestricted, but they are defined in terms of lower-level operations that are restricted.

Type safety for instructions such as `car` and procedure application rely on an assiduously followed tagging discipline. All values, both immediate values such as small integers and access links to objects stored in memory, are represented as *descriptors.* A descriptor has a tag that distinguishes immediate values from links. Arithmetic operations require immediate-tagged operands and affix immediate tags to results. The non-tag portion of a descriptor for a link holds the hardware address of the stored object. A stored object is represented as a contiguous series of memory locations. The first memory location holds a header specifying the object's size and type (procedure, capsule, cell, etc.), while subsequent locations hold descriptors (which include the links to the object's successors in the accessibility network).

The tagging discipline guarantees that object addresses are never seen by the programmer. Not only does this promote security, it also gives the memory manager freedom to move objects from one place to another, guaranteeing that such relocations will not affect any running computations.

The security kernel does not provide direct access to the byte-code interpreter. Instead, new programs must be given as Scheme programs that are byte-compiled by `eval`. The compiler translates the program to a byte-code instruction stream, and then uses a privileged `make-closure` instruction to make a procedure. Use of the `make-closure` instruction is restricted for two reasons:

- An agent could defeat security by constructing and executing code streams containing privileged instructions. For example, the `closure-env` instruction allows unrestricted access to all the successors of an arbitrary procedure.

- An agent could execute an object access instruction with an index specifying a location beyond the end of the object being accessed. This might fetch a reference to an object that shouldn't be seen. (For performance reasons, some instructions perform no bounds checking. They rely on the byte-code compiler to ensure that indexes are valid.)

Abstract data types (Section 2.3.2) are used heavily in the Scheme 48 run-time system. They are provided in the form of a record facility, which is more convenient and efficient than the theoretically equivalent but more elegant `new-seal` would be. In the common case where a capsule is a tuple (that is, has several fields), only a single object (a record) is created for both the tuple and the capsule. Not only is less memory consumed, but field references require only one indirection instead of two.

The features of Scheme 48's security kernel beyond what W7 defines are the following:

- The basic features of standard Scheme [27]: characters, vectors, the full suite of numeric operators, strings, string/symbol conversion. None of these has security implications; they belong to the kernel for the sake of efficiency.

- Exception handling. It is important for the agent initiating a computation to be able to gain control when the computation leads to an exceptional event such as division by zero. A special construct allows all exceptions or exceptions of certain types to be intercepted.

- Fluid variables. Every application implicitly passes an environment that maps unique tokens (called *fluid variables*) to cells. A special construct extends the environment with a new binding. The interesting aspect from a security perspective is that if the fluid variable (unique token) is protected, then bindings of the fluid variable will be protected. Thus if X calls Y calls Z, then it can be the case that Z can access fluid bindings established by X, while Y can't. This contrasts with a similar facility in Unix (process environments), which has no such protection.

- Multiple threads of control. A new thread of control may be started with `(spawn` *thunk*`)`. Synchronization mechanisms include locks and condition variables.

Each thread has its own separate exception context and fluid environment. (Threads are not really secure; see Section 4.3.1.)

- Immutability. This feature is somewhat of a frill, but useful. A pair, vector, or string can be made read-only, as if it had been created by `(quote ...)`. Immutable objects may be passed to untrusted procedures without worry that they might be altered, since an attempted mutation (`set-car!`, `vector-set!`, etc.) will raise an exception.

Variables and pairs are mutable, as in Scheme.

3.2 Module System

Scheme 48 provides operators for constructing and querying environments. Environment control is as important for determining what is in an environment as it is for advertizing what is *not* in an environment. Roughly speaking, an environment that excludes dangerous things can't be dangerous.

The environments that are manipulated as objects and passed to the compiler are called *top-level* environments or *packages.* Packages have both static and dynamic components. The static component defines information important for compilation: syntactic keywords, type information, and early binding information such as definitions of in-line procedures. The dynamic component of a package determines how variables obtain their values at run time.

Bindings move between packages via another kind of environment-like object called a *structure*. A structure is a particular package's implementation of a particular *interface*; and an interface is a set of names, with optional types attached:

`(make-simple-interface` *names*`)` $\longrightarrow$ *interface*
`(make-structure` *package interface*`)` $\longrightarrow$ *structure*

Dually, a package receives bindings from other packages via their structures:

`(make-package` *structures*`)` $\longrightarrow$ *package*

Borrowing terminology from Standard ML, I'll say that a package *opens* the structures from which it receives bindings.

Here is a simple example. Assume that `scheme` is defined to be a structure holding useful bindings such as `define`, `lambda`, and so forth.

```
(define repository-package
  (make-package (list scheme)))
(eval '(begin
         (define *repository* ...)
         (define lookup ...)
         (define publish! ...))
      repository-package)
(define repository-interface
  '(lookup publish!))
(define repository
  (make-structure repository-package
                  repository-interface))
...
(define repository-client
  (make-package (list scheme repository)))
```

(The Scheme 48 module system is described in more detail in Appendix C.)

3.2.1 Checking Structures for Safety

As seen in Section 2.2.6, it is useful to be able to determine whether a procedure is safe. In particular, it is useful to know whether a procedure has access to any resource that could be used as a communication channel to the outside world. If it were possible to traverse the access graph starting from the procedure, it would be possible to determine the answer to this question. However, such a traversal cannot be done outside the security kernel.

In Scheme 48, information necessary to perform certain safety checks is contained in the network of structures and packages. This network can be considered a quotient (summary) of the true access network. A procedure is considered safe when the structure that exports it is safe; a structure is safe if its underlying package is safe; and a package is safe if every structure that it opens is safe. This definition of "safe" is recursive, so it can be made relative to a basis set of structures that are assumed safe. Following is a crude implementation of such a safety predicate:

```
(define (safe? struct assumed-safe)
  (cond ((memq struct assumed-safe) #t)
        ((structure? struct)
         (every (lambda (o)
                  (safe? o assumed-safe))
                (package-opens (structure-package struct))))
        (else #f)))
```

This isn't a precise test, but it has the virtue of being simple and intuitive. Rather than rely on sophisticated techniques such as types, effects, verification, or other kinds of code analysis, it only involves visibility of values.

The usual Scheme 48 run-time system provides a number of structures, some of which may be considered safe for the purpose of guaranteeing the absence of communication channels. The structure implementing the standard Scheme dialect is not one of these because it is so easy to use it to create such channels. A different `safe-scheme` structure is defined that eliminates the possibility of such channels, as described following:

One source of channels is assignments to top-level variables. For example:

```
(define *last-sorted-list* #f)
(define (sort l)
  (set! *last-sorted-list* l)
  (really-sort l))
```

We cannot get rid of top-level definitions, so to prevent this, `set!` must be disallowed on top-level variables (except possibly during initialization). The current solution is to exclude `set!` entirely from `safe-scheme`, since otherwise the compiler would have to be modified, and project time constraints didn't allow this.

Mutations to data structures reachable from top-level variables also must be prohibited, but now instead of excluding mutations entirely (as suggested in Section 2.2.6), we exclude top-level variables that hold mutable data structures. This is done by excluding normal Scheme `define` and replacing it with a variant that makes the right assurances:

- `(define (`*var var* `...)` *body*`)` is allowed.
- `(define` *var exp*`)` is allowed iff *exp* evaluates to a verifiably immutable object.

Verifiably immutable objects include scalars such as numbers and characters, immutable strings, and immutable pairs, vectors, and capsules that have verifiably immutable components. Note that the category doesn't include procedures, since procedures can easily hide channels.

Of course, `safe-scheme` also excludes all operations that access the file system. It includes I/O routines that take explicit port arguments (such as `(display` *x port*`)`) but excludes their implicit-port variants (`(display` *x*`)`).

3.2.2 Static Privileges

The fact that static bindings of syntactic keywords, macros, and primitive operators are all determined relative to a package means that any such entity may be considered a privilege to be granted or not. For example, a particular agent (user or subsystem) might be denied access to all operators that have side effects just by excluding those operators from environments available to the agent. Scoping of all syntactic and static entities effectively allows an equation between languages and modules, with the derivative ideas of safe languages and modules that are safe because they are written in safe languages.

Macros are secure in a precise sense: names that a macro introduces into expanded program text are resolved in the environment of the macro's definition, not in the environment of its use. For example, a package might define `letrec` in terms of `set!` and then export `letrec` without exporting `set!`. The structure with `letrec` and its clients can be considered applicative (or safe) even though the process of compiling it involves source to source rewrites containing imperative operators.

(For further explanation of the problem of lexically scoped macros and the algorithm used by Scheme 48 to implement them, see Appendix D.)

3.3 Deployed Configurations

The Scheme 48 configurations described here are the more interesting ones with respect to protection problems. Other configurations include the single-user development environment (by far the most evolved), a multiprocessor version written at MIT by Bob Brown, and a distributed version developed by Richard Kelsey and others at NEC.

3.3.1 Mobile Robot System

Scheme 48 is the operating system running on four mobile robots at the Cornell Computer Science Robotics and Vision Lab (CSRVL). The main on-board computer system is a 16 MHz MC68000 with 500K RAM and 250K EPROM.

User code is isolated from operating system internals by Scheme's type and bounds checking; this means that there's nothing that a user program can do to harm the robot. This is important because the robots are programmed by undergraduates taking the robotics course, many of whom are relatively novice programmers.

The Scheme 48 virtual machine running on the robot communicates with a development environment running in either Scheme 48 or Common Lisp on a workstation. The byte-code compiler runs on the workstation and sends byte codes to be executed on the robot. This "teledebugging" link offloads to the workstation all debugging and programming environment support, including the byte-code compiler. This division of labor frees up precious memory on the robot. The tether, which can be a physical encumbrance, can be detached without affecting the virtual machine, and reattached at any time for debugging or downloading.

(For a more detailed description of the mobile robot system, see Appendix B.)

3.3.2 Multi-User System

The Scheme 48 development environment can be configured to be accessed over the Internet by multiple users, with each user given a separate initial evaluation environment and thread of control. This configuration, developed by Franklyn Turbak and Dan Winship and called Museme, is a multi-user simulation environment (MUSE) similar to LambdaMOO [12].

3.3.3 WWW Evaluation Server

Scheme 48 can be configured to be run by a World-Wide Web server (`httpd`) in response to an appropriate request arriving over the Internet. This service aims to promote Scheme by giving the general network public a chance to experiment with it easily. A request contains a Scheme program and a Scheme expression, which execute on the server. The server replies with the printed representation of the result of evaluating the expression. The environment in which the programs and expressions run has been explicitly reduced from the default environment in order to limit the capabilities of programs, which, because they come from anyone on the Internet, shouldn't be trusted.

3.4 Security in Standard Scheme

Scheme 48 implements the Scheme standard [27]. In order to maintain security, however, users are given different instantiations of some of the built-in procedures, much as a conventional time sharing system would provides users with different address spaces. If shared between agents, the following procedures would spell trouble for safe cooperation:

- Any built-in procedure that opens a file, such as `open-output-file`. A buggy or malevolent program could overwrite important files or read sensitive information from files.
- Anything that binds or accesses the current input and output ports, such as `display` with no explicit port argument. Use of such procedures would allow "spoofing" — misleading output that could be mistaken for valid messages coming from legitimate source such as an error handler or command processor.
- `load`, which not only accesses the file system, but also reads and writes a "current" interaction environment.

Multi-agent Scheme 48 systems need to ensure nonconflicting use of the external file system (assuming there is an external file system). Each agent has her own instantiations of file-opening procedures; the private versions implement file system access specific to that agent.

Because file opening procedures have access to an agent's files, a utility that opens files must be given not file *names,* but rather *access* to the files in the form of procedures, ports, or some appropriate data abstraction. This transition from name-oriented to value-oriented protection is exactly what is necessary in order to implement the principle of least privilege [51], which is also behind the analogous shift from Lisp 1.5's dynamic scoping to Scheme's lexical scoping.

As with files, one agent's `current-output-port` procedure is not allowed to access an output port that matters to another agent. Each agent has his own version of `current-output-port`, `display`, `write-char`, `read`, etc.

The `call-with-current-continuation` procedure raises a fundamental question: is the ability to invoke one's continuation multiple times a privilege that should be available to all agents? The answer isn't obvious; see Section 4.3.2.

The following features accepted by the Scheme report authors for inclusion in a Revised^5 report [45] are also troubling:

- `(interaction-environment)`. Given `eval`, this has the same problem as `load`.

- `dynamic-wind`. This operator is supposed to establish a dynamic context such that all entries into and exits from the context are guarded by specified actions. A mischievous program could initiate long-running or continuation-invoking computations in an unwind action, and/or set up an arbitrarily large number of nested `dynamic-wind`s, perhaps defeating time-sharing or mechanisms established for regaining control after an error or abort request.

Chapter 4

Conclusion

The basic premise of this work is that program exchange and interaction are powerful modes of cooperation, and should be encouraged by a computational infrastructure that makes them as risk-free as possible. Program exchange and interaction are becoming easier and more common, thanks to advances in hardware infrastructure (increasing numbers of computers and growing network connectivity), but they are hindered because most computer systems provide little protection against the danger that might be inflicted by unknown programs.

The techniques proposed here to attain safety are:

1. Employ a security kernel with a simple but powerful semantics that allows fine-grained control over privileges.

2. Grant an invoked program only the privileges it needs to do its job. Rights to secondary storage and I/O devices shouldn't be implicitly inherited from the invoking agents' privilege set.

3. Certify that the program has passed a reliable safety test.

4. Label or "seal" objects so that they can be authenticated later.

None of these ideas is particularly new. Some can be derived from Saltzer and Schroeder's 1975 paper on protection in computer systems [51], which lays down desiderata for secure systems. (1) is their principle of economy of mechanism, (2) is the principle of least privilege, and (4) is the principle of separation of privilege.

Rather, the main novelty in the present work is the demonstration that a spare security kernel, derived from first principles and spanning the operating system / programming language gulf, can be simultaneously secure and practical.

4.1 Previous Work

4.1.1 Actors

The connection between lambda-calculus and message passing or "actor" semantics is well known [59]. Application in lambda-calculus corresponds to message passing in actor systems, with argument sequences playing the role of messages.

The actor languages (such as [2]) and Scheme were both developed as programming language frameworks, with little explicit attention to security concerns or cooperation between mutually suspicious agents. Security is of course implicit in the fact that an actor (procedure) can only be sent a message, not inspected, and that on a transfer of control only the argument sequence is transmitted, not any other part of the caller's environment.

There is no provision for authentication (actor recognition or abstract data types) either in the actor languages or in Scheme.

4.1.2 MUSEs

A "multi-user simulation environment" (MUSE, also MUD (multi-user dungeon) or MOO (MUD object-oriented)) simulates a world with a number of places (rooms) and inhabited by users (players or characters). A MUSE typically runs on a network and accepts connections from many users at once. Users move from place to place, communicate with each other, and manipulate simulated physical objects. Because objects may have value to users and users and their programs may attempt actions that can harm objects, and because of their generally open door policy, security is an important issue in MUSE design and administration.

Programming languages for MUSEs (in particular, that for LambdaMOO [12]) have the feature that a caller's privileges aren't implicitly passed on to a called program. This is certainly an improvement over the behavior of mainstream operating systems, which give all of the caller's privileges to the callee. The correct behavior is dictated by the demands of the environment: users encounter strange objects and do things to them; this causes invocation of programs belonging to the objects (or rather their creators).

MUSEs are generally based on ad hoc programming languages with many peculiar features that lead to security problems [3], inflexibility, or both. An overall lack of security in MUSEs is betrayed by the fact that it is a special privilege to be a "programmer." New members of a MUSE must convince an administrator that they are worthy before they are allowed to write new programs.

4.1.3 Oooz

Strandh's Oooz system [61] resembles Scheme 48 in aspiring to be a simple, multi-user, Scheme-based programming environment and operating system. The similarity ends there. Oooz extends Scheme with a hierarchical, globally accessible namespace and an object-oriented programming framework. The global namespace is analogous to a conventional file system, with access controlled by access control lists attached to objects. There is no obvious way in Oooz to implement abstract data types or authentication.

4.2 Discussion

4.2.1 Schroeder's Thesis

Schroeder studied the problem of cooperation between mutually suspicious principals in his 1972 dissertation [52]. He describes an extension to the Multics operating system in which caller and callee in a cross-domain call may protect resources from one another.

He was aware of the problem of protecting parameters from unauthorized transmission, but had nothing in particular to say about it:

> In some cases it may be desirable to guarantee that a protected subsystem invoked by another cannot remember and later divulge the values of input parameters. This problem appears to be very difficult to solve in a purely technical way and will not be considered in this thesis. ([52], page 16)

4.2.2 Operating System Security

Status quo operating systems (VMS, Unix, DOS Windows, etc.) don't effectively address safe invocation problems. When a program is invoked, it inherits all of the privileges of the invoker. The assumption is that every program that a user runs is absolutely trustworthy.

Safe invocation could be implemented in some versions of Unix, but it would require some heavy machinery. One way would be to have a privileged program that invokes a given program with the privileges of a specially unprivileged user. Such a program is not standard and cannot be created by an ordinary user. A different implementation would be to have a network server dedicated to the purpose, but the result would be strangely limited and awkward to use (file descriptors couldn't be passed, interrupts might act strangely, etc.).

Unix does have a facility whereby a program's owner can mark the program in such a way that when the program is invoked, it runs with the program's owner's privileges instead of with the invoker's (except for arguments). This sounds very similar to safe invocation of procedures, until one reads the fine print, which points out that such a program can obtain all of the invoker's privileges simply by doing a `setuid` system call.

Most common operating systems distinguish persistent objects (files) from volatile objects (in Unix, file descriptors). An `open` system call coerces a persistent object to a volatile one. Persistent objects have global names, while volatile objects have short numeric indexes that are interpreted locally to a running program. This "scoping" makes them resemble a procedure's successor links. Volatile objects can be passed from one program (process) to another that it invokes, but not in any other way.

Many operating system designs support secure program invocation in ways similar to what I describe , but they aren't deployed.

It is remarkable that we get by without secure cooperation. We do so only because people who use programs place such a high level of trust in the people who write those programs. The basic reason for this high level of trust is that computer systems are isolated from one another. Without communication, there can be no theft, since stolen goods must be communicated back to the thief. Any harm that arises is either vandalism or accident. Vandalism (e.g. the current epidemic of computer viruses) sophisticated enough to be untraceable is difficult to carry off and has little payoff for the perpetrator, while really harmful accidents (as when a commercial software product accidentally erases a disk) are mercifully rare.

4.3 Future Work

This section points out various shortcomings of W7 and Scheme 48, and attempts to suggest ways of fixing them. One of the dreams driving this work is to extend Scheme 48 to cover all operating system services, including device drivers. At that point it should be possible to dispense with the host operating system and run Scheme 48 stand-alone on a workstation, as it does on the mobile robots.

4.3.1 Preemption

A means to preempt a running process is necessary in any operating system. An agent, in invoking an unknown object, runs the risk that the invocation will be non-terminating; therefore the agent must have some way to request that the invocation be halted on some condition, such as receipt of special input from outside the processor (an abort key or button) or the passage of a predetermined amount of time.

Preemption is also desirable in that it is necessary and, together with support for first-class continuations or coroutines, sufficient for constructing a scheduler that simulates multiple hardware processing elements in software. One particularly elegant design for a timed preemption facility is a mechanism known as *engines*, of which Dybvig and Hieb have published a general implementation [26, 17]. Engines abstract over timed preemption by providing a way to run a computation for a specified amount of time. They are sufficient to construct a user mode task scheduler.

Dybvig and Hieb's engine implementation has two problems for a system in which caller and callee are mutually suspicious:

1. Response time is sensitive to engine nesting depth; thus a malevolent callee could pile up a very deeply nested sequence of engines, making response to an outer engine (which should have priority) arbitrarily sluggish.

2. No design is given specifying how engines should interact with waits and interrupts associated with concurrent activities (such as I/O).

If these problems can be solved, and I believe they can, then engines should serve well as part of W7.

(Scheme 48 supports a multitasking scheduler via a threads system, but threads are inferior to engines in several ways. Threads are less secure than engines, since one's share of the processor is proportional to how many threads you have; if you want to take over a processor, you need only create lots of threads. Scheme 48's threads give no reliable way to monitor the execution time of a supervised untrusted computation, since the computation can spawn new threads. And there is no way to limit or even monitor the amount of processor time allocated to a thread, since the threads system doesn't keep track of processor time used per thread.)

4.3.2 Continuations

First-class continuations are troublesome when caller and callee are mutually untrusting. With Scheme's `call-with-current-continuation` operator, the callee can obtain the continuation and invoke it twice. An unwary caller who has continuation code that performs a side effect is then vulnerable to having the side effect happen twice, which may be unexpected and undesired. Must all code that invokes untrusted objects be prepared for this possibility? To deal with the contingency requires code similar to the following:

```
(let ((returned? (new-cell)))
  (cell-set! returned? #f)
  (let ((result (untrusted arg ...)))
    (if (cell-ref returned?)
        (error "I didn't expect this")
        (begin (cell-set! returned? #t)
               (side-effect-to-be-done-only-once!)
               ...))))
```

An unwillingly retained continuation is also undesirable from a resource allocation standpoint, since the continuation's creator might be penalized for tying down resources (space) consumed by the continuation.

4.3.3 Machine Programs

Supporting execution of compiler-generated machine code programs is important for two reasons: it would vastly improve performance relative to Scheme 48's current byte-code interpreter; and, by using existing compilers, Scheme 48 could be made to run useful programs written in a variety of languages (such as Pascal and C).

The main difficulty in supporting machine programs is ensuring that the kernel's security policy is followed; the machine program must not obtain access to any resources that it shouldn't have access to. This could very easily happen, since machine programs can construct arbitrary addresses and attempt to dereference them, or even construct arbitrary machine code sequences and jump to them. Any object that uses memory in the machine program's address space, or that is accessible via any sequence of machine instructions, is at risk.

There are several approaches to eliminating the risks associated with machine programs:

- Limitation: switch into a limited hardware-provided "user mode" when invoking the machine program. This is the approach used by most operating systems.

In user mode, memory not belonging to the program is protected from access, and no hardware I/O is permitted. Some instructions, such as those that alter memory protection registers, are disabled. A protected environment is thus established, and all instructions are either permitted or actively prohibited.

- Verification: a trusted program scans the machine program to make sure that it doesn't do anything that it shouldn't. Unverified programs are rejected.

- Sandbagging: similar to verification, except that extra code is inserted around all troublesome instructions to dynamically ensure that security policy is respected [65]. Some programs may still be rejected, but fewer are than would be with verification.

- Trusted compilation: a program generated by a compiler may respect security policy by construction; if we trust the compiler, we will trust its output.

The type and array bounds safety of many compilers, such as many Scheme and Pascal compilers, are sufficient to guarantee that the kernel's security policy is respected. Such compilers may be used without change.

Limitation may be the only option if one has little influence over the compiler and program being compiled. For example, most C programs use pointers in ways that are difficult to guarantee safe by verification or sandbagging, and few C compilers generate object code that checks validity of pointers. Limitation is to be avoided because transfers into and out of user mode are expensive in many hardware architectures. On the other hand, for programs that do few control transfers to other programs, use of memory protection hardware may be the most efficient way to detect invalid pointers and array indexes (that is, to implement kernel security policy).

An important part of the implementation of machine program support is the interface between machine programs and the kernel. That is, how does a machine program perform a kernel operation, such as creating or invoking a procedure, and how are the operands — generally object references — specified? This will be answered differently depending on the extent to which the machine program can be trusted.

When the machine program is trusted, object references can be represented as pointers to data structures representing objects, and kernel operations can be implemented either as subroutine calls (e.g. object creation can be accomplished by a call to an allocation routine) or as code sequences occurring directly in the program (e.g. invocation of a W7 procedure might be compiled similarly to local procedure call).

When the machine program is untrusted, transfers of control outside of the program must be generally accomplished by a trap or system call. (A few hardware architectures support general calls across protection domains). Object references must be amenable to validation on use, since the program may present an arbitrary bit pattern as a putative object reference. The method used by most operating systems in similar circumstances is to associate, with each activation of an untrusted machine program, a table mapping small integer indexes to objects. (In Unix, the indexes are known as "file descriptors.") The program presents an index, which the kernel interprets as specifying the object at that position in the table. Validation consists of a simple range check to determine that the index is within the table.

Alternatively, objects may be given unique names (perhaps their addresses in memory), and these names can be used by untrusted machine programs. When the program presents a name in a kernel operation, the kernel validates the name by determining whether it occurs in an object table specific to the program's activation (as above). This approach is similar to the "capability-based" approach to protection [38]. For the W7 kernel, there is little reason to prefer this over the small-index approach, since table searches and unique name maintenance are likely to be complicated and inefficient relative to use of indexes, which can be determined when a program is compiled or installed.

4.3.4 Reflection

Reflection is reasoning about self [55]. In a computing system, the concept of reflection comprehends examining the internal structure of objects and continuations for purposes of debugging, analysis, or optimization. Reflection interacts with security issues in ways that to my knowledge have not been researched, much less resolved.

For example, Scheme 48's debugger has access to special reflective operators that break all protection boundaries; it can examine the internals of procedures, records, and continuations. This is very useful, but unfortunately the debugger is egregiously insecure, since it allows any user access to any value transitively accessible from an accessible object. It is easy enough to achieve security in Scheme 48 by disabling the debugger, but a better solution would be for the kernel to provide a simple, safe way to examine continuations and procedures, so that an unprivileged debugger could be built. I believe that this can be done in such a way that a user is able to see what she ought to be entitled to see, but nothing else. In particular, a user should be able to see any of the information that exists in object representations, as long as that information might have been available to her had she included extra debugging hooks in her own programs.

4.3.5 Other Issues

Persistence. Some support is needed for keeping objects in secondary storage efficiently, and there should be some guarantees about which objects will necessarily survive crashes. Currently Scheme 48 relies on an external file system to store information persistently; this is not integrated with the internal protection system beyond the fact that access to the file system as a whole can be limited.

Quotas. Limits should be imposed on the amount of memory agents should be allowed to use. Without this, a malevolent or buggy program can consume all available space, making the system unuseable. Memory limitation is more difficult than execution time limitation; the allowed space decreases as memory is allocated, but must increase as memory is reclaimed by garbage collection. With each object, then, must be associated an "account" to be credited when the object is reclaimed. This could be done either by putting an extra field in each object indicating the account, or by maintaining distinct regions of memory for different accounts. The latter is reminiscent of multi-stage garbage collection , and could perhaps be unified with it.

Accountability. When something goes wrong, it's nice to know who is responsible, if not exactly how it went wrong. But responsibility is difficult to assign in cooperative enterprises. In particular, if something goes wrong when one agent invokes another agent's program, who is responsible? If a server receives a request from a client, is it the server or the client who is responsible for ensuring that the request is a reasonable and safe one?

Revocation. There is no way in W7 or Scheme 48 to revoke access to an object. Objects simulating revokable links could be defined using cells, but it's not obvious that this would be sufficient. One would have to decide when giving out access to an object. the times when one would want to revoke a link might be precisely the times when one didn't anticipate that one would want to.

Distribution. A network of encapsulated objects lends itself to distribution over a network of computer systems. One major component of a distributed operating system that is missing from W7/Scheme 48 is a remote invocation mechanism (classically called RPC, or remote procedure call). Such a mechanism should have the following properties:

- Objects can be passed as arguments and returned as results. This requires automatic creation of "stub" or delegate objects that forward calls over the network.
- Calls are properly tail recursive: in a call from node A to node B, B can make a call to node C specifying that the result is to be sent to A, not B.
- Exceptions are distributed transparently, in that an exception on one machine can find the correct exception handler even if it's on another machine.

Alan Bawden has described and implemented something relevant in his PhD thesis [4].

4.4 Contributions

The success of Scheme 48 shows that a spare security kernel can provide flexible and solid security without becoming difficult to use or to program.

Is W7 simpler than all other security kernels? Much of the complexity in operating system kernels arises from performance concerns that W7 has yet to address, so a rational comparison is difficult. It is difficult to see how it could be smaller, since all of the services it provides are (as is shown) necessary: program (object) creation and invocation; object marking and authentication; primitive access to I/O devices.

There are some minimalists in the Scheme community who believe that procedures, perhaps together with some primitive data types such as symbols and cells, serve as a basis for the construction of all other useful programming constructs. I hope that the discussion of authentication and abstract data types (Section 2.3.3) will be seen to refute this position and to show the need for a built-in authentication mechanism in minimal programming languages.

The ease with which Trojan horses and viruses may infiltrate computer systems is appalling, as is the extent to which users must blindly trust software provided by vendors. I hope I have contributed a bit to the currently unpopular cause of principled solutions to security problems.

I hope the document helps to break down the artificial distinction between programming languages and operating systems. Progress in both of these areas is hindered by a failure to recognize that the concerns of both are fundamentally the same. What is needed is operating systems that provide services that are useful to languages, and languages that give convenient access to operating system services.

Appendix A

A Tractable Scheme Implementation

This paper reports on joint work with Richard Kelsey, who is co-author. Previously published in Journal of Lisp and Symbolic Computation *7(4):315–335, 1995. Reproduced here with permission of Kluwer Publishing, Inc.*

Abstract

Scheme 48 is an implementation of the Scheme programming language constructed with tractability and reliability as its primary design goals. It has the structural properties of large, compiler-based Lisp implementations: it is written entirely in Scheme, is bootstrapped via its compiler, and provides numerous language extensions. It controls the complexity that ordinarily attends such large Lisp implementations through clear articulation of internal modularity and by the exclusion of features, optimizations, and generalizations that are of only marginal value.

A.1 Introduction

Scheme 48 is an implementation of the Scheme programming language constructed with tractability and reliability as its primary design goals. By tractability we mean the ease with which the system can be understood and changed. Although Lisp dialects, including Scheme, are relatively simple languages, implementation tractability is often threatened by the demands of providing high performance and extended functionality. The Scheme 48 project was initiated in order to experiment with techniques for maintaining implementation tractability in the face of countervailing pressures and to find out what tradeoffs were involved in doing so (the project was originally an experiment to see if a Scheme implementation could be written in a single weekend; the 48 refers to forty-eight hours).

Small Lisp implementations are usually tractable merely by virtue of being small; it is usually possible for an experienced programmer to read and understand the entire source program in a few days. However, Scheme 48 specifically attempts to tackle the structural problems posed by large Lisp implementations. Although it is impossible to define the terms small or large precisely in this context, by a small Lisp system we generally mean one that is less than about 15,000 lines of source code, is based on an interpreter or simple translator not written in Lisp, and provides few, if any, features beyond a small core language such as IEEE standard Scheme [27]. By a large Lisp system we mean one that is more than about 15,000 lines and implements a large language such as Common Lisp [57] or a substantially extended dialect of Scheme.

We arrived at the number 15,000 by examining the source code for a collection of Lisp implementations. We found seven small Lisp systems in public file archives on the Internet, none of which contained more than 14,000 lines of source code, and three large systems of 25,000 to 120,000 lines. All of the large systems and two of the small ones included native code compilers. The line counts are those for the interpreters and run-time code and do not include the compilers.

Size is not the only reason that large Lisps tend towards incomprehensibility. Large implementation size is a result of additional functionality. Usually difficulties of comprehension are compounded by the authors' use of the additional features within the implementation itself. Even worse, once a system has been bootstrapped, features are often re-implemented in terms of themselves, and the original, non-circular code is deleted from the sources. Circular definitions are hard to understand and even harder to modify.

Scheme 48 is meant to be a substantial yet comprehensible system. The current Scheme 48 system includes many extensions to standard Scheme and has the basic structural characteristics of the large systems we surveyed. Tractability is difficult to measure, but our claim that Scheme 48 is tractable is based on the following features:

- explicit interfaces are used to divide the code into modules;
- these modules are organized in a hierarchical fashion: the major components of the system can be understood independently;
- the modules are largely independent: Scheme 48 can be built from the available sources, in a mix and match fashion, as anything ranging from a minimal implementation of a Scheme subset to an extended development environment;
- there are multiple implementations of many of the interfaces;
- all of the code is written in Scheme;
- major parts of the system, including the code to build a working image from the source files, can be run stand-alone using any IEEE Scheme implementation.

In addition, when writing the code for Scheme 48 we took the unusual step of giving simplicity priority over execution speed, unless measurements demonstrated that the increase in speed was significant. Implemented features not meeting this criterion were removed. The system's overall simplicity helped in performing such experiments by making it easy to isolate and evaluate different implementation choices.

A.2 System Organization

We used two main strategies to keep Scheme 48 comprehensible: the system is written entirely in Scheme, using a variety of dialects, and is divided into modules with explicit

Table A.1: Scheme Dialects

Dialect	Use within Scheme 48
Pre-Scheme	Virtual machine implementation
Primitive Scheme	Minimal compiled subset
Scheme	General, standard
Big Scheme	General
Configuration language	Module descriptions

interfaces and dependencies. For each module we used the simplest adequate dialect and tried to minimize the number of dependencies on other modules. The Scheme dialects used are listed in table A.1. Primitive Scheme and Big Scheme have the same semantics as Scheme, with the deletion or addition of procedures and syntax but without significant change to the underlying semantics. Pre-Scheme is more of a departure, as it is designed to allow compilation to idiomatic C code (integers are `int`s and so forth). This requirement results in some significant restrictions on Pre-Scheme programs, although they can still be run as Scheme programs. Pre-Scheme is discussed in section A.4.

The Scheme 48 system is based on a virtual machine consisting of a byte-code interpreter and a memory manager. Using a virtual machine raised many of the same organizational issues as using a native code compiler, such as how bootstrapping is accomplished, without forcing us to deal with actually generating machine code or with other details of hardware architecture. The virtual machine architecture gives a well-defined interface to the execution hardware. The run-time code can use that interface and ignore the question of whether the underlying machine is real or virtual. Using a virtual machine also has the immediate and practical benefit of making Scheme 48 easy to port to new platforms.

The Scheme 48 implementation has four main parts: a realization of the virtual machine, a Scheme compiler that produces code for the virtual machine, a static linker used to build executable images, and implementations of both standard and non-standard Scheme library routines. The virtual machine is written in Pre-Scheme, so that it can be compiled to efficient C or native code, and then run as a stand-alone program. The byte-code compiler provides an implementation of Primitive Scheme, and the rest of Scheme is implemented using code written in Primitive Scheme. The compiler, linker, utilities, and the extensions that make up Big Scheme are written in Scheme. Some of the extensions, such as multitasking, make use of functionality provided by the virtual machine that is not part of standard Scheme, and these modules cannot be run using other Scheme implementations. The virtual machine, compiler, and linker can be (and are) run using Scheme implementations other than Scheme 48.

Scheme 48's module system is designed to support both static linking and rapid turnaround during program development. The design was influenced by Standard ML modules [41] and the module system for Scheme Xerox [13]. Each module has its own isolated namespace, with visibility of bindings controlled by module descriptions written in a module configuration language. The module system bears some similarity

Table A.2: Virtual machine interfaces

External interface	
`architecture`	Instruction set description
Data structures	
`memory`	Pointers and pointer arithmetic
`descriptors`	Typed descriptors for Scheme objects
`fixnum-arithmetic`	Small integer arithmetic with overflow checks
`stored-objects`	Manipulating heap objects
`data-types`	Particular stored object types
`ports`	Scheme I/O ports
Storage management	
`heap`	Heap (including garbage collector)
`environments`	Lexical environments
`stacks`	Operand stack and continuations
Byte-code interpreter	
`interpreter`	Instruction dispatch; error detection
`resume`	Initialize, read a heap image, and start interpreting

to Common Lisp's package system [57], although it controls the mapping of names to denotations instead of the mapping of strings to symbols.

A.3 The Virtual Machine

The Scheme 48 virtual machine is a stack based machine similar to that described in [9]. It is written in Pre-Scheme, a Scheme subset that is described in section A.4. The implementation of the virtual machine is organized according to the internal interfaces listed in Table A.2. The following sections describe the interfaces and their implementations.

A.3.1 Architecture

The Scheme 48 architecture description is the interface between the virtual machine and the rest of the Scheme 48 system. It contains descriptions of the virtual machine's instruction set and data structures, and the kinds of possible interrupts.

Three kinds of data structures are described in the interface: fixed size objects containing other Scheme objects, vectors of Scheme objects, and vectors containing untagged data. For objects of fixed size, such as pairs, the architecture description lists the names of the various components (`car`, `cdr`) in the order in which they are stored in memory. Vectors of Scheme objects include Scheme vectors and records, and the templates described below. Vectors of untagged data include strings and vectors of byte codes.

A.3.2 Data Representations

The basic data structure representing Scheme objects is a descriptor. A descriptor is the size of a pointer and contains a tag in the low-order two bits. (This particular representation is tuned for use on byte-addressed machines where a pointer is four bytes.) The tag is one of:

- Fixnum: the non-tag portion of the descriptor represents a small integer. We use zero as the fixnum tag to simplify arithmetic operations.
- Immediate: the low-order byte gives the type of the object; the rest contains any other necessary information. Characters, the empty list, `#t` and `#f` are all represented as immediate values. Another use of immediate values is as illegal data values, allowing the virtual machine to detect and report attempts to refer to the value of uninitialized variables and other errors.
- Stored object: a pointer to a stored object. The first descriptor in the object is a header describing the object. The pointer actually points to the first non-header descriptor in the object. Most Scheme objects, such as strings, pairs, vectors, and so on are represented as stored objects. Stored objects either contain descriptors or untagged data depending on their type.
- Header: description of a stored object. The descriptor includes the type of the object, its size, and an immutability flag. If the immutability flag is set, attempts to modify the object result in an exception being raised.

The garbage collector and other heap manipulation routines are written to allow headers within stored objects. An object that normally contains only tagged data, such as a record, can also contain untagged data, as long as an appropriate untagged data header is placed before the untagged data. The system currently makes no use of this facility.

Having headers on all stored objects allows the contents of memory to be scanned. Starting at the head of any stored object, it is possible to scan through memory determining the type and contents of every subsequent stored object. The virtual machine makes use of this in a number of ways.

There are a number of alternatives to this particular data layout, some of which have been used in other implementations [37]. Headers can use either a fixnum or an immediate tag, freeing up one tag value. This tag can be used to denote a pointer to a commonly used stored object type, typically pairs, eliminating the need for headers on objects of that type. Or the fourth tag can be unused, allowing some tag tests to be done using bit-test instructions. We chose the current data representations because they are simple and contain a certain amount of redundancy, and because saving one or two machine instructions per type check or `cons` is not likely to be important for an interpreted system. The modular nature of the entire system would make it easy to change the data representations, should the need arise.

There are five aggregate data structures used by the virtual machine: templates, closures, continuations, locations, and run-time environments. These are all stored objects. A template is a piece of executable code. Each contains a vector of instructions and a vector of values that includes constants, locations, and other templates. A closure is the representation of a procedure; it contains a template and an environment. A continuation contains the information needed to restart the machine when a procedure call returns, specifically the operand stack and contents of some of the machine's registers. A location contains the value of a top-level variable. Locations are distinct from symbols to allow for multiple top-level environments.

Run-time environments contain the values of lexically bound variables. These are implemented as Scheme vectors, with the first value in each vector being the superior lexical environment. The address of a lexical variable is given as two indices: the depth of the containing environment, with the current environment being depth zero, and the index of the variable within the containing environment. A common alternative is to use a single vector for each environment, copying any necessary values from other environments whenever a new environment is made. This eliminates the need to chain back through environments when fetching the values of variables. We used nested environments to keep from having to add free variable analysis to the byte-code compiler.

A.3.3 Storage Management

The storage management system provides for the creation and reclamation of stored objects, including continuations and environments. In addition, a stack interface includes operations for maintaining the operand stack. Most objects are allocated from a heap. Heap storage is reclaimed using a simple two-space copying garbage collector. The design of the virtual machine puts few restrictions on the garbage collector and an early version of the VM has been used for extensive experiments in this area [69].

The contents of the heap may be written to and read from files. Heap image files contain a pointer to a distinguished entry procedure in that heap. Heap images are machine independent; they contain the information needed to relocate the heap in memory and to correct byte order if necessary. When the virtual machine is started it reads a heap image from a file and begins execution by calling the entry procedure. Execution ceases when that procedure returns. Heap image files can be created either by using the `write-image` instruction or by using the static linker. A heap image file can be quite small, since it only needs to contain the information required to execute a call to the entry procedure.

Continuations and environments are treated specially, since they can often be reclaimed more efficiently than most objects. They are ordinarily not needed by the program after the return from the call for which they were created. The only exception to this is when `call-with-current-continuation` is used.

There are four operations involving continuations: 1) create a continuation that contains the current argument stack and the machine's internal state; 2) invoke a continuation, restoring the argument stack and the machine's state; 3) preserve the current continuation for later use; and 4) make a preserved continuation become the current one. (These are used for: 1) making a non-tail-recursive call; 2) procedure return; 3) `call-with-current-continuation` and 4) applying the result of a call to `call-with-current-continuation`.) The simplest implementation of this interface is to create all continuations in the heap. Creating and invoking continuations then requires copying the operand stack to and from the heap. Preserving and selecting continuations are no-ops, since continuations are always in the heap and all heap objects have indefinite extent.

An alternative implementation is to create continuations on the operand stack by pushing the machine's state on top of the current operands. This avoids the need to copy the operand stack back and forth. It also improves data locality by efficiently reusing stack space. Preserving a continuation becomes more expensive because it requires copying the continuation into the heap. The continuation is copied back to the stack when it is invoked. Continuations may be freely copied back and forth, because they are never modified, only created and invoked.

Environments can also be allocated either in the heap or on the stack. An environment is created by making a vector that contains a pointer to the current environment and the contents of the operand stack. This can be done by allocating a vector in the heap and moving the values into it, or by adding a vector header to the operand stack, at which point the stack pointer points to the new environment. As with continuations, preserving an environment that is on the stack requires moving it to the heap. Environments can be modified through the use of `set!`, so there must be only one copy of each environment. Once an environment has migrated to the heap it remains there.

Stack storage is reclaimed in two ways. When a continuation is invoked the stack pointer is set to point to the top of the restored operand stack, freeing up any stack space that was allocated since the continuation's creation. If the stack overflows, the current continuation and environment are copied to the heap, allowing the entire stack to be reused. All environments and continuations pointed to by these values are also copied if they are on the stack. This makes stack allocation compatible with Scheme's requirement for proper tail recursion, and also allows for recursions that are deeper than would fit in the stack. Tail-recursive calls cause the calling procedure's environment to become inaccessible, although it remains on the stack until the program returns past that point or the live portions of the stack are moved to the heap. For a detailed discussion of the interaction of tail-recursion and continuation stacks see [23].

The storage allocation interface has been implemented both with and without stack allocation of environments and continuations. Using stack allocation is faster than using the heap, with some cost in increased complexity of the `stack` and `environment` modules. For more information see [30].

Another method of implementing proper tail recursion is to ensure that at every procedure call the arguments to the call are on the stack directly above the continuation for the call. For tail-recursive calls this requires moving the arguments just before jumping to the code for the called procedure. We added this form of tail call to the VM by adding a special call instruction that did the argument copying, and found that it was only slightly slower than using the stack garbage collector. However, the stack copying logic is still required for implementing `call-with-current-`

Table A.3: The virtual machine's registers

`Value`	the most recent instruction result
`PC`	byte-code program counter
`Template`	instruction vector and a vector of literal values
`Cont`	continuation
`Env`	environment
`Nargs`	number of arguments
`Dynamic`	dynamic state
`Enabled-Interrupts`	which interrupts are currently enabled
`Interrupt-Handlers`	vector of procedures for processing interrupts
`Exception-Handler`	procedure for processing exceptions

`continuation`, so the argument copying instruction is not used. For more details see [30].

A.3.4 Interpreter

This section describes the more important virtual machine instructions. These instructions make use of the interpreter's registers, which are listed in Table A.3. To start with a simple example, here are the instructions for the procedure `(lambda (x) (+ 10 x))`.

```
(check-nargs= 1)
(make-env 1)
(literal '10)
(push)
(local 0 1)
(+)
(return)
```

When control is transferred to the code for the procedure, `Nargs` contains the number of arguments being passed and `Env` holds the lexical environment in which the procedure was created. The procedure first checks that it was called with exactly one argument, and creates a new environment containing the argument and the current value of `Env`. An exception is raised by `check-nargs=` if the procedure was passed the wrong number of arguments.

The body of the procedure begins by loading the literal value `10` from `Template` and pushing it onto the stack. The `local` instruction is used to obtain the value of `x`, using two operands that give the depth of `x`'s environment in the chain of lexical environments and the offset of `x` in that environment. `+` then pops `10` off of the stack, adds it to the value of `x` (which is in `Value`), and leaves the result in `Value`. Finally, the procedure returns. The `return` instruction restores the operand stack and sets the `PC`, `Template`, `Env`, and `Cont` registers using values from the current contents of `Cont`.

For a procedure that takes an arbitrary number of arguments, such as `(lambda (x y . more-args) ...)`, the first few instructions would be

```
(check-nargs>= 2)
(make-rest-list 2)
(push)
(make-env 3)
```

The instruction `(check-nargs>= 2)` verifies that there are least two arguments present. `(make-rest-list 2)` puts all but the first two arguments into a list, which is left in `Value`. This list is then pushed onto the stack to become part of the environment.

The value of a `lambda` expression is a closure, containing a template and an environment. A closure is made using the `closure` instruction. The new closure contains the current environment and a template obtained from the literal values in the current template.

A non-tail-recursive procedure call is performed by creating a new continuation, pushing the arguments onto the stack and using the `call` instruction. Arguments are pushed from left to right. The code for `(g 'a)` is:

```
    (make-cont L1 0)
    (literal 'a)
    (push)
    (global g)
    (call 1)
L1: ...
```

The label operand to `make-cont` specifies the program counter to be used when the continuation is resumed. The second operand to `make-cont` is the number of values that are currently on the operand stack. These values must be saved in the continuation. The value of a top-level variable is obtained from a location stored in the template. The number of arguments in the call is itself an operand supplied to the `call` instruction.

Tail-recursive calls are treated the same way, except that the `make-cont` instruction is omitted.

A variable's value is set using `set-local!` or `set-global!`, which are identical to `local` and `global` except that they set the variable's value to be the contents of `Value` instead of vice-versa.

The architecture contains both jump and conditional jump instructions for executing conditionals. `(if a 1 2)` compiles to:

```
    (global a)
    (jump-if-false L1)
    (literal '1)
    (jump L2)
L1: (literal '2)
L2: ...
```

(If the expression were in a tail-recursive position both arms would end with calls or returns and the `(jump L2)` would be omitted.) For both jump instructions the offset to the label must be positive; backwards jumps are not allowed. This ensures that every loop contains a procedure call, which in turn makes handling interrupts more uniform. There could be a backward jump instruction that included a check for interrupts, but making use of it would require a byte-code compiler somewhat more sophisticated than the current one.

The following is a description of some of the less frequently used instructions that manipulate the interpreter's internal state. In addition to the instructions described, there are a number of others that deal with numeric operations, allocating, accessing, and modifying storage, various input/output operations, and so on. Arguments to these instructions are pushed on the stack from left to right, with the last argument left in `Value`. Instruction results are also placed in `Value`.

There are a set of generic instructions for dealing with stored objects such as pairs, symbols, vectors and strings. These instructions take the type of the object as an additional operand from the instruction stream. Adding new types of stored objects does not require changing the instruction set.

`current-cont`
`with-continuation`
The first instruction sets `Value` to be the contents of `Cont`. The second takes two operands, a continuation obtained using `current-cont` and a procedure of no arguments. It sets `Cont` to be the continuation and then calls the procedure. Together these are used to implement Scheme's `call-with-current-continuation`.

`get-dynamic-state`
`set-dynamic-state!`
These move the contents of `Dynamic-State` to `Value` and vice versa. The machine makes no other use of `Dynamic-State`. An alternative design would be to keep the dynamic state in a top-level variable instead of in a VM register. Using a machine register allows the VM to be used in a multiprocessor environment, where each processor has its own registers but all share a single heap.

`set-exception-handler!`
This sets `Exception-Handler` to be the procedure in `Value`. Whenever an exceptional situation is detected the VM calls this procedure, passing to it the current instruction and its operands. If the exception handler returns, execution proceeds with the instruction following the one that caused the exception. Most causes of exceptions are type errors. Others include arithmetic overflows, unbound and uninitialized variables, passing the wrong number of arguments to procedures, and errors when calling operating system routines. Appropriate exception handlers can be used to extend and augment the virtual machine.

`set-interrupt-handlers!`
`return-from-interrupt`
`set-enabled-interrupts!`
The VM checks for interrupts whenever a `call` instruction is executed. If an enabled interrupt has occurred since the previous call, the state of the machine is saved in a continuation and the procedure appropriate for that interrupt is obtained from

```
 0  (check-nargs= 2)
 2  (make-env 2)
 4  (local 0 1)                value of j
 7  (push)                     push j as the first argument to +
 8  (make-cont (=> 20) 1)      20 is the code, stack has one value
12  (local 0 2)
15  (push)                     push i as the first argument to h
16  (global h)
18  (call 1)                   call h with one argument
20  (+)                        code resumes here after the call to h
21  (push)                     push second argument to g
22  (global g)
24  (call 1)
```

Figure A-1: Byte codes for `(lambda (i j) (g (+ j (h i))))`

the vector `Interrupt-Handlers` and called. The `return-from-interrupt` instruction can then be used to restore the state of the machine and continue execution. `set-enabled-interrupts!` sets `Enabled-Interrupts` and places its old value in `Value`.

`write-image`
This takes two operands, a procedure and the name of a file, and writes the procedure into the file as a restartable heap image. The writing process is similar to a garbage collection in that it only writes objects reachable from the procedure into the file. See section A.3.3 for more information.

There are several instructions used to allow calls to return multiple values. They are more complex than strictly necessary, due to our desire not to require any additional overhead when returning exactly one value, and are not described here for reasons of space.

A.4 Pre-Scheme Implementations

The virtual machine is written in Pre-Scheme, a subset of Scheme chosen to allow compilation into fairly natural C code without losing too much of Scheme's expressive power. The virtual machine and the rest of the system are written in overlapping subsets of the same language, allowing some modules (such as the architecture description code) to be used in both. Because Pre-Scheme is a subset of Scheme, the virtual machine can be run and debugged using any Scheme implementation. This fact has greatly assisted development, in particular because it permitted us to work on the Scheme 48 virtual machine prior to the existence of a direct Pre-Scheme compiler or working Scheme 48 system. The VM is quite slow when run as a Scheme program. At one point a hand translation of the VM into C was done by Bob Brown, an MIT graduate student.

The virtual machine is compiled from Pre-Scheme into C using a compiler based on the transformational compiler described in [31]. The Pre-Scheme compiler evaluates top-level forms, performs static type checking and a great deal of partial evaluation, and translates the resulting program into C. The result is a C version of the VM which is as efficient and as portable as a hand-coded C version would be.

Pre-Scheme differs from Scheme in the following ways:

- Every top-level form is evaluated at compile time and may make unrestricted use of Scheme.
- The Pre-Scheme compiler determines all types at compile time. Type reconstruction is done after the top-level forms have been evaluated. There is no run-time type discrimination.
- There is no automatic storage reclamation. Unused storage must be freed explicitly.
- Proper tail recursion is guaranteed only for calls to `lambda` forms bound by `let` and `letrec` or when explicitly declared for a particular call.

The main restrictions imposed by Pre-Scheme are that the VM must be accepted by Pre-Scheme's type-checking algorithm and that there is no automatic storage reclamation. The latter restriction is reasonable, as Pre-Scheme is intended for writing such things as garbage collectors. The Pre-Scheme compiler uses a Hindley-Milner polymorphic type reconstruction algorithm modified to allow overloaded arithmetic operators and to take into account the compiler's use of partial evaluation. If a procedure is to be in-lined for all uses, then it can be fully polymorphic, as every use will have a distinct implementation.

Using Pre-Scheme has been very successful. The code for the VM makes use of higher-order procedures, macros that operate on parse trees, and other features of Scheme that would not have been available had we used C. Procedures are also used to implement data abstractions within the VM. This results in a large number of small procedures. The virtual machine's 570 top-level forms, consisting of 2314 lines of Scheme code (excluding commments), are compiled to 13 C procedures containing 8467 lines of code. Once compiled into C it runs as quickly as similar hand-coded C programs.

Figure A-2 illustrates the coding style used in the VM. The example consists of the code implementing the addition instruction. The first two forms are from the `fixnum-arithmetic` module, which exports `add-carefully`, and the last two are from the `interpreter` module. The procedure `carefully` takes an arithmetic operator and returns a procedure that performs that operation on two arguments, either passing the tagged result to a success continuation, or passing the original arguments to a failure continuation if the operation overflows. `extract-fixnum` and `enter-fixnum` remove and add type tags to small integers. The function `overflows?` checks that its argument has enough unused bits for a type tag. `goto` indicates that

```
(define (carefully op)
  (lambda (x y succ fail)
    (let ((z (op (extract-fixnum x) (extract-fixnum y))))
      (if (overflows? z)
          (goto fail x y)
          (goto succ (enter-fixnum z))))))
(define add-carefully (carefully +))
(define (arith op)
  (lambda (x y)
    (op x y return arithmetic-overflow)))
(define-primitive op/+ (number-> number->)
  (arith add-carefully))
```

Figure A-2: Implementation of the addition instruction

a tail-recursive call should be compiled using proper tail recursion. `carefully` can then be used to define `add-carefully` which performs addition on integers.

`define-primitive` is a macro that expands into a call to the procedure `define-opcode` which actually defines the instruction. The three arguments to the macro are the instruction to define, input argument specifications, and the body of the instruction. The expanded code retrieves arguments from the stack, performs type checks and coercions, and executes the body of the instruction. This is a simple Scheme macro that would be painful to write using C's limited macro facility.

Figure A-3 shows the C code produced for the addition instruction. This is part of a large `switch` statement which performs instruction dispatch.

(A note on identifiers: many Scheme identifiers are not legal C identifiers, while on the other hand C is case-sensitive and Scheme is not. The compiler uses upper-case letters for the characters that are legal in identifiers in Scheme but not in C; for example `*val*` becomes `SvalS`. The compiler also introduces local variables to shadow global variables to improve register usage (similar to [64]). These introduced variables begin with `R`; thus `RSvalS` is a local variable shadowing the global variable `SvalS`.)

This code is not what we would have written if we had used C in the first place, but it is at least as efficient. The use of Pre-Scheme makes the program relatively more comprehensible and easier to modify without incurring any run-time cost.

A.5 Byte Code Compiler

The byte-code compiler compiles Scheme expressions into appropriate sequences of VM instructions. The compiler is as simple as we could make it and still get acceptable performance. It does very few optimizations. We use a simple compiler because it was easier to write, is easier to read, is more likely to be correct, and improves the

```
case 46 : {
  long arg2_267X;
  /* pop an operand from the stack */
  RSstackS = (4 + RSstackS);
  arg2_267X = *((long*)((unsigned char*)RSstackS));
  /* check operand tags */
  if ((0 == (3 & (arg2_267X | RSvalS)))) {
    long x_268X;
    long z_269X;
    x_268X = RSvalS;
    /* remove tags and add */
    z_269X = (arg2_267X >> 2) + (x_268X >> 2);
    /* overflow check */
    if ((536870911 < z_269X)) {
      goto L20950;}
    else {
      /* underflow check */
      if ((z_269X < -536870912)) {
        goto L20950;}
      else {
        /* add tag and continue */
        RSvalS = (z_269X << 2);
        goto START;}}
   L20950: {
    merged_arg1K0 = 0;
    merged_arg0K1 = arg2_267X;
    merged_arg0K2 = x_268X;
    goto raise_exception2;}}
  else {
    merged_arg1K0 = 0;
    merged_arg0K1 = arg2_267X;
    merged_arg0K2 = RSvalS;
    goto raise_exception2;}}
  break;
```

Figure A-3: Compiler output for the addition instruction

Table A.4: Byte-code compiler interfaces

`module-system`	Signatures, modules, and name lookup
`syntactic`	Denotations; hygienic macros
`compilation`	Byte-code compiler
`analysis`	Optional optimization phase
`linker`	Static heap image linker
`reification`	Support for doing `eval` relative to a statically linked image
`assembler`	Optional byte-code assembler

quality of debugging information provided to users. The less processing the compiler does the easier it is to relate the state of the machine to the source program when errors are encountered.

The compiler takes as arguments an expression, an environment, the number of values currently pushed on the stack, and information about the expression's continuation. The stack value count is needed to generate `make-cont` instructions. The continuation argument has two parts, a kind and an optional name. The kind is either `return` (the expression's value is returned by the procedure being compiled), `ignore` (the value will not be used), or `fall-through` (there is some following code that uses the value). The name indicates the variable to which the value of the expression will be bound, if any. If the expression is a `lambda` form, the name will be used to provide debugging support.

For each lexically bound variable the compile-time environment has the location of the variable in terms of the number of run-time environments that must be chained through (back) and the index of the variable within the final environment (over). For each top-level variable the environment has either a location, from which the variable's value can be obtained at run time, a special compilation method, or both. For a variable bound to a primitive operator the compilation method is a procedure for generating code for the operator. If the variable is bound to a macro, the compilation method is a code transformation procedure. The macro facility is based on the system described in [10]. Many of the special forms in Scheme are implemented using macros.

The compiler itself is quite simple. Each procedure is compiled into a separate template. The code for each checks the number of arguments, makes a new environment and then executes the body.

Literal values are put in the template, and loaded at run time using the `literal` instruction. Each variable is looked up in the environment. The instruction generated for accessing or setting the variable's value depends on the denotation returned. `if` is compiled using `jump-if-false` and then `jump` to reach the following piece of code. A `lambda` form is compiled into a template, which is placed in the current template as an operand to `closure`. `begin` concatenates the code generated for each expression.

From the point of view of the compiler there are three kinds of procedure calls: calls to primitives, calls to `lambda` forms, and all other calls. Calls to primitives are compiled using the procedure obtained from the environment. The code for other calls starts with `make-cont` if the call is not tail-recursive (that is, if the `cont` argument to the compiler is not `return`.) Then the arguments are compiled in order followed by `push` instructions. If the call is to a `lambda` form the code for the `lambda` is generated in-line to save the cost of creating a closure. For all other calls code to evaluate the procedure is compiled followed by a `call` instruction.

The compiler also contains code to compile procedural versions of the primitives. For many primitives, such as `+`, it is necessary also to have a procedure of the same name that invokes the primitive.

The output of the compiler is passed to an internal assembler which calculates jump offsets, builds the vector of values for the template, and makes the template itself. This assembler also produces a structure containing any debugging information emitted by the compiler. Some of this information is indexed by the sections of the assembled code to which it pertains. Debugging information includes the names of bound variables and their locations in the environment, the source code for procedure calls and their continuations, and names of procedures. There is a separate assembler, not used by the compiler, that can assemble hand-written byte-code programs.

A.5.1 Optimizations

We have added a few optimizations to the compiler and the instruction set since the original implementation. There are quite a number of other optimizations that could have been done but were not, usually because the increase in execution speed was not felt to be worth the added complexity. Among the optimizations that are in the system are special instructions for accessing local variables in the current environment or its immediate ancestors, not creating closures for `lambda` forms found in call position, not creating empty environments, and a few others.

The instruction set has not been optimized for execution speed or code size to any great extent. A few instructions, such as the `string=?` instruction, could be replaced by significantly slower Scheme code in the run-time system. There are a number of common sequences of instructions that could be merged into single instructions. Experiments have determined that the resulting speed up is not large; the VM does not appear to be spending a great deal of its time in fetching instructions.

Other possibilities for increased optimization abound. Some examples: calls to known procedures, such as those for simple loops, could be compiled as jumps; a `jump-if-null` instruction would speed up list processing code; boolean expressions could be compiled for effect; leaf procedures (those that do not create other procedures) don't need to create environment vectors.

The VM instructions needed for many such optimizations already exist. However, the compiler would have to be made considerably more complex in order to make use of them. Because the compiler is itself usually run using Scheme 48, additional compiler complexity carries with it the penalty of significantly slower compilation speed. A system was built that did not create leaf environments. In our judgement, the increase in speed did not make up for the additional complexity and increased compilation time, so this optimization was removed.

One change that might or might not improve the simplicity or speed of the system would be to eliminate the value register and the `push` instruction. The value register is

Table A.5: Scheme language interfaces

`primitive-scheme`	Special forms and procedures that are necessarily implemented directly by a compiler or interpreter: `lambda`, `if`, `car`, `read-char`, etc.
`usual-macros`	Scheme's usual derived expression types: `let`, `cond`, `do`, etc.
`scheme-level-1`	Procedures that are easily implemented in terms of `primitive-scheme`: `not`, `memq`, `equal?`, etc.
`numeric-tower`	Number types other than small integers (bignum, ratnum, flonum, recnum)
`winding`	`call-with-current-continuation` and `dynamic-wind`
`number-i/o`	`number->string` and `string->number`
`reading`	The `read` procedure
`writing`	The `write` procedure
`evaluation`	`eval` and `load`
`high-level-macros`	The `syntax-rules` macro
`scheme`	Union of the above interfaces

equivalent to a cache for the top of the stack. Removing it would make the architecture description a little shorter. The size of the code for the system would be essentially unchanged. The number of VM instructions executed would decrease, as there would not be any `push` instructions, but many of the instructions would have to do a little more work.

In any case, speed is not a particularly high priority for Scheme 48. When speed is vital users will go to a system that compiles Scheme to native machine code. Scheme 48's byte-code interpreter cannot match the speed provided by an optimizing native-code compiler.

A.6 Run-time library

The byte-code compiler and virtual machine implement a core Scheme dialect that we call Primitive Scheme. Scheme 48's run-time library builds additional functionality on top of Primitive Scheme.

A.6.1 Scheme in terms of Primitive Scheme

The `scheme` interface specifies the entire Scheme language as defined by the Revised^4 Report [11]. `scheme` is organized as the union of a number of smaller interfaces for two purposes: (1) configuring small systems that use only particular subsets of the language, and (2) structuring implementations of the full language.

The modules that provide implementations of `scheme`'s sub-interfaces (Table A.5) are straightforward, for the most part. The `primitive-scheme` interface is implemented directly by the byte-code compiler and virtual machine, while all the other interfaces have ordinary implementations as Scheme programs.

The module for `numeric-tower` is more closely tied to the VM architecture than most of the other library modules. It consists of a set of exception handlers for the VM's arithmetic instructions. If the VM interpreter encounters such an instruction for which the arguments are numbers that are not implemented directly by the VM, then it raises an exception. The exception is handled by a procedure that dispatches to an appropriate specialist function: for example, if the two arguments are large integers, then the large integer addition routine is invoked.

Following the philosophy of minimizing internal dependencies, the numeric tower is carefully constructed so that the numeric types do not depend on one another. It is possible to make use of any combination of number types; for example, rational numbers can be used in the absence of large integers.

The `evaluation` interface has two implementations. The first is a simple meta-circular interpreter that does not rely on the virtual machine architecture. The second implements `eval` as a three-step process:

1. Invoke the byte-code compiler on the form, producing a template.
2. Make a closure from the given template and a null lexical environment. (Global variables are accessed via the template, not via the environment component of the closure.)
3. Invoke the closure as an ordinary procedure.

Step 3 is the "reflective" step that starts the virtual machine running on the newly compiled form.

Many of these interfaces include entry points that do not become part of Scheme. For example, the `numeric-tower` interface includes the procedures necessary to add additional types of numbers.

A.6.2 Big Scheme in terms of Scheme

"Big Scheme" consists of a diverse collection of utilities and language extensions of general interest. A few of these modules are used in the implementation of the basic Scheme library. For example, the definitions of `current-input-port` and `current-output-port` are in terms of dynamic variables, which come from the `fluids` interface. Also, the byte-code compiler makes heavy internal use of records, tables, and enumerated types. Other modules, such as conditions and interrupts, are used in the development environment (see next section). Still other modules, such as multitasking and external calls, are useful for general applications programming.

Many of the interfaces in this group have two implementations: one written in portable Scheme, and another that exploits special features of the Scheme 48 virtual machine architecture. For example, the record package has a portable implementation

Table A.6: Utilities and extensions interfaces

`bitwise`	Bitwise logical operations on integers
`records`	Record package
`fluids`	Dynamically bound variables
`enumerated`	Enumerated types
`tables`	Hash tables
`byte-vectors`	Blocks of memory accessible by byte, word, or half-word
`conditions`	Condition system
`exceptions`	Particular conditions for errors at the level of `primitive-scheme`
`interrupts`	Handling asynchronous interrupts; critical sections
`queues`	FIFO queues
`random`	Random number generator
`sort`	Sorting lists
`pp`	Pretty-printer
`format`	Formatted output
`extended-ports`	I/O from/to strings, etc.
`external-calls`	External function call
`threads`	Multitasking
`weak-pointers`	Weak pointers
`big-scheme`	Union of the above interfaces

in which records are represented using vectors, and a Scheme 48-specific implementation in which records are a distinct primitive type. The portable version can be used when running the linker or other applications on a substrate other than Scheme 48, while the Scheme 48-specific version provides increased functionality, performance, or debuggability.

A.6.3 Scheme development environment

A complete development environment, comprising byte-code compiler, run-time library, a command processor, and debugging utilities, can be packaged as a single heap image for execution by the VM. This image can be generated by the byte code compiler and linker using any implementation of standard Scheme. Scheme 48 is "bootstrapped" only in the straightforward sense that it includes an implementation of standard Scheme, which is a sufficient basis for executing the code necessary for it to build itself.

The command processor is part of the Scheme development environment. It is similar to an ordinary Lisp read-evaluate-print loop in that it repeatedly reads and executes forms (expressions and definitions), but differs in that it also accepts requests for meta-level operations, and these requests (called commands) are syntactically distinguished from forms to be executed. Meta-level operations include obtaining

Table A.7: Development environment interfaces

`command-processor`	Command-oriented user interface
`package-commands`	Commands for manipulating modules
`application-builder`	Application image builder
`disclosers`	Extensions to `write` and `display-condition` to assist in debugging
`debugging`	Trace, backtrace, time, and similar commands
`inspector`	Data structure, stack, and environment inspector
`disassembler`	Byte-code disassembler

backtraces or trace output, specifying compiler directives, measuring execution time, and exiting the development environment. The purpose for this distinction is to leave the program namespace uncluttered with bindings of operators that are only meaningful during program development. For example, in Common Lisp, one would say `(trace foo)` to request tracing output for calls to `foo`, but in the Scheme 48 command processor one says `,trace foo`. When debugging programs that make use of the module system, the entire command set remains available regardless of the current expression evaluation context.

A.7 Discussion

Our claim is that by writing Scheme 48 as a collection of largely independent modules we have produced a sophisticated and extended Scheme implementation that is reliable, tractable, and easy to modify. Although we have not stressed implementation efficiency in this paper, Scheme 48 is a reasonably fast system, and is in use as a turnkey Scheme programming system. It runs at approximately the speed of the fastest available C-based interpreters. (The widely differing implementation methodologies make it difficult to obtain meaningful numbers.)

The Scheme 48 system has been used for research in memory management, embedded systems, multiprocessing, language design, and computer system verification. Scheme 48 was chosen as the platform for these projects because of its internal tractability and flexibility.

Paul Wilson has done extensive research on memory management techniques using an early version of Scheme 48 [68, 69]. Because of its virtual machine architecture, Scheme 48's memory usage patterns are similar to those of more complex systems, and it is easier to modify and experiment with.

Scheme 48 is currently being used to program mobile robots [48]. Multi-threaded user programs are executed by the virtual machine on an embedded processor on board the robot, while the byte-code compiler and development environment run on a work station using a Scheme implementation built on top of Common Lisp. Programs are compiled on the work station and then downloaded to the robot via a detachable tether. The on-board component of the system uses the standard virtual machine (with some additional hardware operations) and a stripped down version of the run-time library. There is no need for a read-eval-print loop on board, for example, and the system fits easily within the limited memory available on the robot (0.5 megabytes of RAM and 0.25 megabytes of EPROM). The virtual machine and the initial heap image take up 21 and 80 kbytes, respectively, of the EPROM.

Olin Shivers used Scheme 48 as a substrate for `scsh` [54], a Unix shell programming language based on Scheme that gives the user full access to existing Unix programs. Scheme 48 was chosen because it provides a good programming environment, a general exception handling mechanism, and the ability to make small stand-alone programs, all of which are desirable for a shell programming language.

The VLISP project at MITRE Corporation and Northeastern University [21] used Scheme 48 as the basis for a fully verified Scheme implementation. The existence of the virtual machine interface, the simplicity of the byte-code compiler, and the fact that the entire system is written in Scheme combined to greatly simplify the verification process. The fact that the VLISP project was able to adopt much of Scheme 48's design unchanged corroborates our claim that Scheme 48 is tractable and reliable.

The current version of Scheme 48, 0.36, is available at the following locations:

```
ftp://cs.indiana.edu/pub/scheme-repository/imp/scheme48-0.36.tar.Z
ftp://ftp-swiss.ai.mit.edu:/pub/s48/scheme48-0.36.tar.gz
```

Appendix B

Program Mobile Robots in Scheme

This paper is co-authored with Bruce Donald. Most of the work and text is mine. An earlier version of this paper appeared in the Proceedings of the 1992 IEEE International Conference on Robotics and Automation, *Nice, France, pages 2681–2688. Reproduced with permission of the IEEE.*

We have implemented a software environment that permits a small mobile robot to be programmed using the Scheme programming language[11]. The environment supports incremental modifications to running programs and interactive debugging using a distributed read-evaluate-print loop. To ensure that the programming environment consumes a minimum of the robot's scarce on-board resources, it separates the essential on-board run-time system from the development environment, which runs on a separate workstation. The development environment takes advantage of the workstation's large address space and user environment. It is fully detachable, so that the robot can operate autonomously if desired, and can be reattached for retrospective analysis of the robot's behavior.

To make concurrent applications easier to write, the run-time library provides multitasking and synchronization primitives. Tasks are light-weight and all tasks run in the same address space. Although the programming environment was designed with one particular mobile robot architecture in mind, it is in principle applicable to other embedded systems.

B.1 Introduction

The Lisp family of programming languages has a long history as a basis for rapid prototyping of complex systems and experimentation in new programming paradigms. Polymorphism, higher-order procedures, automatic memory management, and the ability to use a functional programming style all aid the development of concise and reliable programs[1]. In this paper we promote the application of the Scheme dialect of Lisp to programming a mobile robot system whose limited resources might contraindicate the use of high-level language features and an integrated programming environment.

The following example gives the flavor of the Scheme system. The procedure `sonar-accumulate` takes as its arguments an initial state and a procedure that combines a previous state with a sonar reading to obtain a new state. The combination procedure is called once for each of the twelve sonar transducers, and is supplied with the transducer number and the sensed distance. The returned value is the final state. `sonar-accumulate` is polymorphic in that the nature of states is not known by the implementation of `sonar-accumulate`, and may be different in each use of this operator.

```
(define (sonar-accumulate combine init)
  (let loop ((val init)
             (i 0))
    (if (>= i 6)
        val
        (let* ((j (+ i 6))
               (rs (read-sonars i j)))
          (loop (combine j (cdr rs)
                         (combine i (car rs)
                                  val))
                (+ i 1))))))
```

The implementation of this control abstraction hides the fact that the sonar hardware allows only particular transducer pairs to be read simultaneously. `sonar-accumulate` might be used to determine which transducer is giving the smallest reading:

```
(define (nearest-sonar)
  (sonar-accumulate
        (lambda (i dist previous)
          (if (< dist (cdr previous))
              (cons i dist)
              previous))
        (cons -1 infinity)))
```

The paper is structured as follows: Sections B.2 and B.3 describe the hardware and software architecture of the robot and its Scheme implementation. Section B.4 briefly presents facilities available to the programmer beyond what Scheme ordinarily provides. Section B.5 discusses the relative success of the design and ways in which it could have been different. Section B.6 describes what we would like to do next.

B.2 Hardware

The particular hardware targeted by this project is a Cornell mobile robot built on the Real World Interface B12 wheel base (figures 1 and 2). The robot is roughly cylindrical, about 50 cm tall with a 30 cm diameter. An enclosure contains a rack in which is mounted several processor and I/O boards. The robot architecture is distributed and modular, so that different sensors and effectors are easily added and removed. Interprocessor communications is via 19.2 Kbaud serial lines. Low-level I/O is handled by a Cornell Generic Controller (CGC), a general-purpose control board

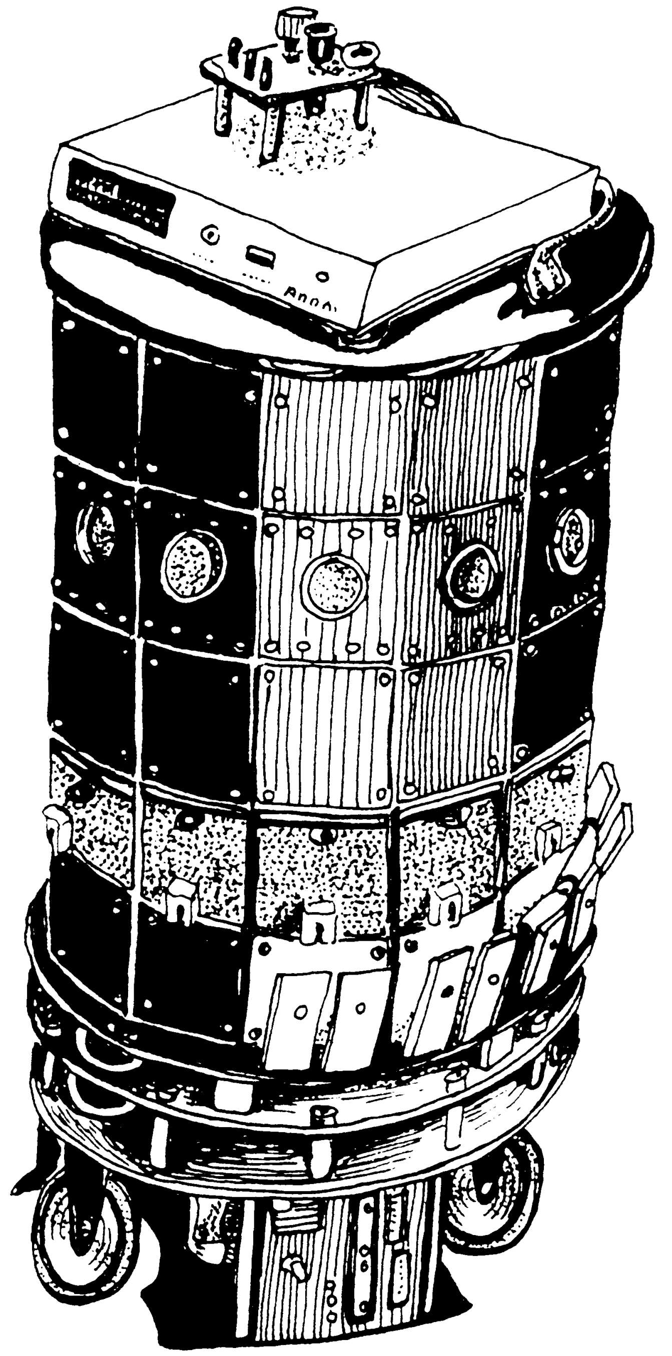

Figure B-1: Tommy, the mobile robot.

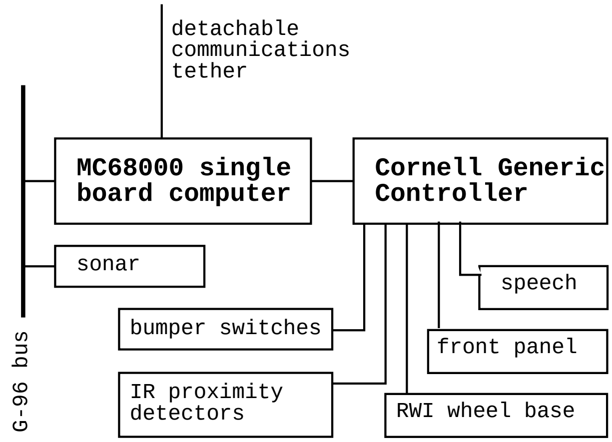

Figure B-2: Hardware architecture.

based on an Intel 80C196 processor[28]. High-level task control and planning are performed by Scheme programs running on an off-the-shelf Motorola 68000 processor board (Gespak MPL 4080). The 68000 board has .5 Mbyte RAM and .25 Mbyte EPROM, and Scheme currently uses no off-board memory.

The entire robot is powered by rechargeable batteries in the wheel base. The robot draws about 1.2 amps when idle, more when moving.

Much of Scheme's communication with sensors and effectors is accomplished by messages transmitted to and received from a Cornell Generic Controller over a serial line. The CGC relays messages to and from various other devices. In principle Scheme could communicate directly with many devices as it does with the sonar, but we prefer to off-load device control to the CGC, which has richer I/O capabilities.

The features of this architecture relevant to Scheme are:

- Small physical size (one 10 by 17 by 1 cm circuit board) — this means small memory size compared to a workstation.
- Low power consumption — this means a slow processor (16 MHz 68000 board, 130 mA).
- Low bandwidth for communications with the workstation.

Also part of the hardware for the overall development system is a workstation capable of running a full-sized Scheme or Common Lisp implementation, text editor, and so on; currently this is a Sun Sparcstation, but any similar workstation would work as well. The workstation communicates with Scheme on the robot over a 19.2 Kbaud serial line tether. Some kind of wireless communication would be nice, but we are concerned about robustness and dropout, and would like to maintain the option of running autonomously.

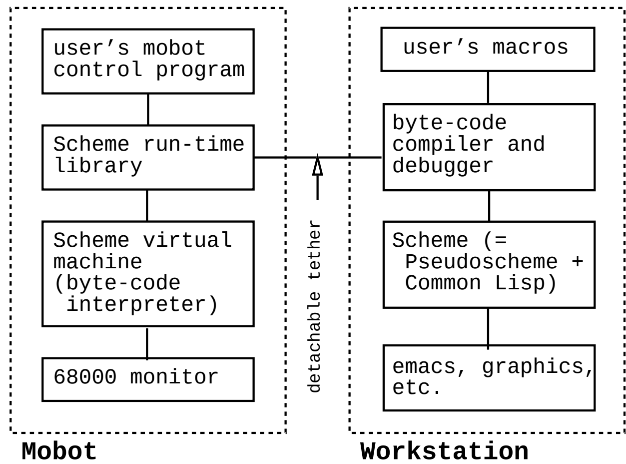

Figure B-3: Software architecture.

B.3 Scheme System Architecture

The Scheme system consists of a run-time environment, which resides on the robot, and a development environment, which resides on a separate workstation. The two communicate with each other over a serial line. Figure 3 summarizes the major software components of the Scheme system.

B.3.1 Run-time Environment

The run-time environment builds on the Scheme 48 virtual machine architecture [32]. The virtual machine is byte-code based and stack-oriented, closely resembling the target machine of the Scheme 311 compiler [9], and similar in spirit to [20]. The VM handles memory management: allocation instructions (such as `cons`) can cause garbage collections. The virtual machine is implemented by an interpreter and garbage collector written in C. Compiled for the 68000, the virtual machine consumes about 24 Kbytes of EPROM.

The VM has I/O instructions corresponding to the Scheme procedures `read-char` and `write-char`. These instructions are executed using traps to a simple supervisor-mode monitor. Access to the appropriate sonar control registers also requires a small amount of virtual machine support. Other than Scheme and the monitor, no other operating system runs on the 68000.

The virtual machine executes both user code and software for communication with the development system. The communications software and a standard run-time library (see section B.4) reside in EPROM, while byte codes for user programs are downloaded from the workstation over the tether.

B.3.2 Development Environment

The programmer using the Scheme system interacts with the development environment, which runs under a Scheme implementation on a workstation. For Scheme on the workstation, we are currently using a Scheme to Common Lisp translator under Lucid Common Lisp, but MIT Scheme or any of various other Scheme implementations would work as well. We could run Scheme 48 itself on the workstation, but Lucid and MIT Scheme are preferable for their speed (they both sport optimizing native-code compilers) and for their integration with other software on the workstation, including existing packages for graphics, planning, computational geometry, and spatial reasoning.

The development environment is an 11 Mbyte executable image, and usually runs under the control of GNU Emacs. It includes a command loop that accepts Scheme expressions to evaluate and other commands that control the environment in various ways: load a source file, reset the run-time system, etc. The development system translates Scheme source code into a byte-code instruction stream, which is transmitted over the serial line to the Scheme run-time system on the 68000.

The development system performs as much preprocessing as possible on user code before sending the code to the run-time system. The compilation process includes symbol table lookups, so no variable names or tables need to reside in the robot itself. Error messages and backtraces that come back from the run-time system are interpreted relative to the same symbol tables resident in the development system. As a result of this policy, on-board overhead is kept to a minimum. The run-time library, which includes communications software, standard Scheme library procedures such as `append` and `string->number`, and the extensions described below, is only about 70 Kbytes of byte codes and data.

When the tether is connected, it is possible for a user program on the robot to call procedures on the workstation, and vice versa. For example, a program running on the robot can initiate graphics routines that display output on the workstation's monitor.

B.4 Run-time Library

Besides standard Scheme procedures such as `+` and `vector-ref`, our Scheme comes equipped with procedures that support sensor/effector control, multitasking, and remote procedure call.

B.4.1 Controlling Sensors and Effectors

The run-time library contains a set of procedures for obtaining information from sensors and for issuing commands to effectors. For example, `(read-sonar 7)` reads sonar unit number 7, and `(translate-relative 250)` instructs the wheel base to initiate a 250 mm forward motion. At a low level, the various devices use different units and coordinate systems, but the library converts to consistent units for use by Scheme programs.

Most sensor and effector control is mediated by a CGC. The run-time library contains routines written in Scheme that communicate with the CGC over a serial line. Most operations consist of a single message exchange. Access to the serial lines and sonar hardware is synchronized to prevent conflicts when several threads perform different operations concurrently.

B.4.2 Lightweight Threads

Multitasking is useful in writing programs that simultaneously manage several different input and output devices, or sensors and effectors in the case of a robot. Our Scheme environment supports light-weight threads with a library routine `spawn`. The argument to `spawn` is a procedure of no arguments. The call to `spawn` returns immediately, and a new thread is started concurrently to execute a call to the procedure. All threads run in the same address space, so threads may communicate with one another by writing and reading shared variables or data structures.

For example, it may be convenient to have a dead-reckoning integrator running continuously in its own thread:

```
    (define (reckon-loop)
      (let ((o1 *current-odometry*)
            (o2 (read-odometers)))
        (set! *current-configuration*
              (integrate-configuration
                     *current-configuration*
                     o1
                     o2))
        (set! *current-odometry* o2))
      (sleep *reckon-interval*)
      (reckon-loop))

(spawn reckon-loop)
```

Other threads may consult the integrator's estimation of the current configuration (x, y, θ) by simply consulting the value of `*current-configuration*`. Because variable references and assignments (and in fact, all virtual machine instructions) are atomic, any particular reference to this variable will yield a consistent configuration triple.

The implementation of threads is written entirely in Scheme, that is, above the level of the virtual machine. This is made possible, even easy, by the existence in Scheme of the `call-with-current-continuation` primitive, which the virtual machine implements efficiently. See [66] for an elegant presentation of this technique for building an operating system kernel. (Actually, thread switching uses the same low-level continuation mechanism that `call-with-current-continuation` does, but does not interact with `dynamic-wind`, described in the appendix.)

Because threads are ordinary Scheme objects with no special status in the virtual machine, they are garbage collected when no longer accessible. (All currently runnable threads are of course accessible.)

Synchronization is provided by lock operations similar to those in Zetalisp or Lucid Common Lisp: `(make-lock)` creates a new lock, and `(with-lock` *lock thunk*`)` calls *thunk* after obtaining control of the lock, blocking if some other thread currently holds the lock. The lock is released on any normal or exceptional return from the call to *thunk*. A second form of synchronization is condition variables as in [49], which are single-assignment storage locations. A condition variable is created with `make-condvar`, assigned with `condvar-ref`, and set (at most once) with `condvar-set!`. A `condvar-ref` that precedes a condition variable's `condvar-set!` blocks until some other thread performs such a `condvar-set!`.

B.4.3 Remote Procedure Call

Our Scheme system also supports a simple remote procedure call capability. Whenever the tether is attached, a procedure on the mobot may call a procedure on the workstation, or vice versa. Because address spaces are not shared, the mechanism is not transparent, but it is supported by the procedures `host-procedure` and `mobot-procedure`. E.g.

```
(define plan-path
  (host-procedure 'plan-path))


... (plan-path destination) ...
```

This defines a procedure `plan-path` on the robot that, when called, initiates an RPC to the procedure of the same name residing on the host. This might be desirable if, say, `plan-path` were too slow or too large to run on the robot itself. Displaying graphical output is another use for RPC's from the robot to the workstation.

Similarly, a Scheme program on the workstation may call procedures on the robot using `mobot-procedure`. For example, robot sensor and effector routines are easily accessed from the workstation:

```
(define read-all-infrareds
  (mobot-procedure 'read-all-infrareds))

... (read-all-infrareds) ...
```

Multiple threads on host and robot may perform remote procedure calls concurrently.

B.5 Discussion

Our thesis is that the combination of a high level language with rapid turnaround for changes allows for more experiments with the robot per unit time. As with many claims about software engineering, this is difficult to test in any objective way. But we think this will turn out to be true, as it has apparently been true when Lisp and Scheme have been applied to other domains.

There is nothing new about programming robots in Lisp. To our knowledge, however, our implementation is unique in providing an advanced on-board Lisp environment for a small robot.

Why implement Scheme? There exist good cross-compilers and cross-debuggers for C, and these are in many ways well suited for developing embedded systems. However:

- We prefer Scheme to C because its high-level features (polymorphism, automatic memory management, and higher-order procedures) allow for more concise, reliable programs.

- Because of its neutral syntax and powerful macro and code-manipulation facilities, Scheme and Lisp have historically been a good base for experimentation with special-purpose languages.

- The conventional architecture for a C programming environment requires compiling entire modules and linking the entire program every time a change is made to the program. With compiled programs being sent over a 19.2 Kbaud serial line, this makes the turnaround time for changes unacceptably slow; and the alternative of putting the C compiler and linker on the robot itself would make the robot too large.

Another possible choice for a general-purpose robot programming language would have been ML (or Concurrent ML[49]). Scheme made more sense for us given the educational background of the students and researchers that use the robot. Also, the Scheme 48 system already existed when this project started, and was already well suited for cross-development; adapting an existing ML implementation would have been much more work for us.

What about real-time constraints? Languages with garbage collection have traditionally suffered from delays of up to several seconds while memory is being reclaimed. We take the approach that one can live with short delays. Essential tasks that require that there be no garbage collection delays can run on separate processors that do not run garbage collectors (e.g. they can be programmed in C). However, the fact that the 68000's memory is nonvirtual and fixed size means that we can put an upper bound on the amount of time taken by a garbage collection, and limiting the amount of live data will limit the frequency of garbage collection; thus we can get absolute time bounds for specific tasks, even when they run as Scheme programs that allocate memory. With our current garbage collector, a garbage collection of a 50% full heap requires over half a second. This is slow, but we believe that this time can be improved upon by tuning the code or by switching to a generational collector[40].

Our choice of synchronization primitives is merely conventional, not necessarily final. The set is not really sufficient in that it does not include an easy way to wait on multiple events. We experimented with Concurrent ML's primitives[49], but found that programs using them were difficult to debug. The `future` construct[22] would be easy to implement in the Scheme virtual machine, but is probably inappropriate in this context, since its purpose is exploiting parallelism, not programming embedded systems.

While there has been some work on special-purpose robot control languages (see e.g. [29] and review in [19]), we considered it safer to hedge our bets by putting our effort into building an uncommitted general-purpose infrastructure.

B.6 Future Work

Members of the Cornell Robotics and Vision Laboratory are using this Scheme system for prototyping a variety of navigation, planning, and manipulation systems. In particular, we intend to use the mobile robot for testing a mathematical theory of task-directed sensing and planning[15]. As the Scheme system continues to be exercised, opportunities to improve the programming infrastructure will continue to arise. The communications software needs to be made more robust, and further debugging aids, including a trace package and inspector, need to be implemented.

The imminent arrival of additional robots running Scheme will raise interesting issues in developing control programs for collaboration. It will be possible to use a single host environment for coordinated debugging of multiple robot systems.

We would like to experiment with and compare various programming language constructs and paradigms for describing robot control systems. Scheme should be an ideal medium for this. Of particular interest to us are subsumption architecture[7], ALFA[19], and Amala[14].

Performance may be a problem in the future. Any interpreter for a virtual instruction set is likely to be 20 to 30 times slower than equivalent code compiled for the target hardware. If the interpreter turns out to be a bottleneck, we'll consider using a Scheme compiler, either an existing one (Scheme-to-C, MIT Scheme, etc.) or a new one. The advantages of doing so must be weighed against the effect of lower density of native code relative to the virtual instruction set. This might be an important consideration given current memory limitations.

Appendix C

Scheme 48 Module System

This memo describes a module system for the Scheme programming language. The module system is designed so that it can support both static linking and rapid turnaround during program development. The design was influenced by Standard ML modules[41] and by the module system for Scheme Xerox[13]. It has also been shaped by the needs of Scheme 48, a virtual-machine-based Scheme implementation designed to run both on workstations and on relatively small (less than 1 Mbyte) embedded controllers.

C.1 Introduction

The module system supports the structured division of a corpus of Scheme software into a set of module descriptions. A module description is a syntactic construct that describes one or more *structures* in a manner similar to the description of procedures by `lambda`-expressions. A structure is a set of bindings of names to denotations; a denotation is either a location or a syntactic keyword.

The structures specified by a module description are formed by extracting bindings from an environment consisting of a set of explicitly imported bindings augmented by bindings specified by a sequence of top level Scheme forms processed in that environment. Bindings are not automatically inherited from the environment in which the module description is processed.

A structure is created by taking a subset of the bindings in the new environment. The particular set of names whose bindings form the structure is the structure's *interface*.

An environment imports bindings by either *opening* or *accessing* structures. When a structure is opened, all of its exported bindings become visible in the environment. On the other hand, bindings from an accessed structure require explicitly qualified references written with the `structure-ref` operator.

For example:

```
(define-structure foo (export a c cons)
  (open scheme)
  (begin (define a 1)
         (define (b x) (+ a x))
         (define (c y) (* (b a) y))))
```

```
⟨configuration⟩ ⟶ ⟨definition⟩*
⟨definition⟩ ⟶  (define-structure ⟨name⟩ ⟨interface⟩ ⟨clause⟩*)
              | (define-structures ((⟨name⟩ ⟨interface⟩)*) ⟨clause⟩*)
              | (define-interface ⟨name⟩ ⟨interface⟩)
              | (define-syntax ⟨name⟩ ⟨transformer-spec⟩)
⟨clause⟩ ⟶  (open ⟨name⟩*)
          | (access ⟨name⟩*)
          | (begin ⟨program⟩)
          | (files ⟨filespec⟩*)
⟨interface⟩ ⟶  (export ⟨item⟩*)
             | ⟨name⟩
             | (compound-interface ⟨interface⟩*)
⟨item⟩ ⟶  ⟨name⟩ | (⟨name⟩ ⟨type⟩)
```

Figure C-1: The module language.

```
(define-structure bar (export d)
  (open scheme foo)
  (begin (define (d w) (+ a (c w)))))
```

These module descriptions define two structures, `foo` and `bar`. `foo` is a view on an environment in which the `scheme` structure's bindings (including `define` and `+`) are visible, together with bindings for `a`, `b`, and `c`. `foo`'s interface is `(export a c cons)`, so of the bindings in its underlying environment, `foo` only exports those three. Similarly, structure `bar` consists of the binding of `d` from an environment in which both `scheme`'s and `foo`'s bindings are visible. `foo`'s binding of `cons` is imported from the Scheme structure and then re-exported.

The module body, the part of the description following `begin` in the above example, is evaluated in an isolated lexical scope completely specified by the `open` and `access` clauses. In particular, the binding of the syntactic operator `define-structure` is not visible unless it comes from some opened structure. Similarly, bindings from the `scheme` structure aren't visible unless they become so by `scheme` (or an equivalent structure) being opened.

C.2 The Module Language

The module language consists of top-level defining forms for structures and interfaces. Its syntax is given in figure 1.

A `define-structure` form introduces a binding of a name to a structure. A structure is a view on an environment which is created according to the clauses of the `define-structure` form. Each structure has an interface that specifies which bindings in the structure's underlying environment can be seen when that structure is opened in other environments.

An `open` clause specifies which structures will be opened up for use inside the new environment. At least one structure must be specified or else it will be impossible to write any useful programs inside the environment, since `define`, `lambda`, `cons`, `structure-ref`, etc. will be unavailable. Typical structures to list in the `open` clause are `scheme`, which exports all bindings appropriate to Revised[5] Scheme, and `structure-refs`, which exports the `structure-ref` operator (see below). For building structures that export structures, there is a `defpackage` structure that exports the operators of the module language. Many other structures, such as record and hash table facilities, are also available in the Scheme 48 implementation.

An `access` clause specifies which bindings of names to structures will be visible inside the module body for use in `structure-ref` forms. `structure-ref` has the following syntax:

⟨expression⟩ ⟶ `(structure-ref` ⟨struct-name⟩ ⟨name⟩`)`

The ⟨struct-name⟩ must be the name of an `access`ed structure, and ⟨name⟩ must be something that the structure exports. Only structures listed in an `access` clause are valid in a `structure-ref`. If an environment accesses any structures, it should probably open the `structure-refs` structure so that the `structure-ref` operator itself will be available.

The module body is specified by `begin` and/or `files` clauses. `begin` and `files` have the same semantics, except that for `begin` the text is given directly in the module description, while for `files` the text is stored somewhere in the file system. The body consists of a Scheme program, that is, a sequence of definitions and expressions to be evaluated in order. In practice, I always use `files` in preference to `begin`; `begin` exists mainly for expository purposes.

A name's imported binding may be lexically overridden or *shadowed* by simply defining the name using a defining form such as `define` or `define-syntax`. This will create a new binding without having any effect on the binding in the opened structure. For example, one can do `(define car 'chevy)` without affecting the binding of the name `car` in the `scheme` structure.

Assignments (using `set!`) to imported and undefined variables are not allowed. In order to `set!` a top-level variable, the module body must contain a `define` form defining that variable. Applied to bindings from the `scheme` structure, this restriction is compatible with the requirements of the Revised[5] Scheme report.

It is an error for two opened structures to both export the same name. However, the current implementation does not check for this situation; the binding is taken from the structure that is comes first within the `open` clause.

File names in a `files` clause can be symbols, strings, or lists (Maclisp-style "namelists"). A "`.scm`" file type suffix is assumed. Symbols are converted to file names by converting to upper or lower case as appropriate for the host operating system. A namelist is an operating-system-indepedent way to specify a file obtained from a subdirectory. For example, the namelist `(rts record)` specifies the file `record.scm` in the `rts` subdirectory.

If the `define-structure` form was itself obtained from a file, then file names in `files` clauses are interpreted relative to the directory in which the file containing the `define-structure` form was found.

C.3 Interfaces

An interface can be thought of as the type of a structure. In its basic form it is just a list of variable names, written `(export` *name* ...`)`. However, in place of a name one may write `(`*name type*`)`, indicating the type of *name*'s binding. Currently the type field is ignored, except that exported syntactic keywords must be indicated with type `:syntax`.

Interfaces may be either anonymous, as in the example in the introduction, or they may be given names by a `define-interface` form, for example

```
(define-interface foo-interface (export a c cons))
(define-structure foo foo-interface ...)
```

In principle, interfaces needn't ever be named. If an interface had to be given at the point of a structure's use as well as at the point of its definition, it would be important to name interfaces in order to avoid having to write them out twice, with risk of mismatch should the interface ever change. But they don't.

Still, there are several reasons to use `define-interface`:

1. It is important to separate the interface definition from the module descriptions when there are multiple distinct structures that have the same interface — that is, multiple implementations of the same abstraction.

2. It is conceptually cleaner, and useful for documentation purposes, to separate a specification (interface) from its implementation (module description).

3. My experience is that segregating interface definitions from module descriptions leads to text that is easier to read than if the two kinds of definition are mixed together or interfaces go unnamed. The interface definitions, usually consisting of rather long lists of exported bindings, are usually not of interest when examining module descriptions.

The `compound-interface` operator forms an interface that is the union of two or more component interfaces. For example,

```
(define-interface bar-interface
  (compound-interface foo-interface (export mumble)))
```

defines `bar-interface` to be `foo-interface` with the name `mumble` added.

C.4 Macros

Lexically scoped macros, as described in [10, 11], are implemented. Structures may export macros; auxiliary names introduced into the expansion are resolved in the environment of the macro's definition.

For example, the `scheme` structure's `delay` macro might be defined by the rewrite rule

$$\texttt{(delay } \mathit{exp}\texttt{)} \quad \Longrightarrow \quad \texttt{(make-promise (lambda () } \mathit{exp}\texttt{))}.$$

The variable `make-promise` is defined in the `scheme` structure's underlying environment, but is not exported. A use of the `delay` macro, however, always accesses the correct definition of `make-promise`. Similarly, the `case` macro expands into uses of `cond`, `eqv?`, and so on. These names are exported by `scheme`, but their correct bindings will be found even if they are shadowed by definitions in the client environment.

ednote

C.5 Higher-order Modules

There are `define-module` and `define` forms for defining modules that are intended to be instantiated multiple times. But these are pretty kludgey — for example, compiled code isn't shared between the instantiations — so I won't describe them yet. If you must know, figure it out from the following grammar.

$$\begin{array}{lll} \langle\text{definition}\rangle & \longrightarrow & \texttt{(define-module (}\langle\text{name}\rangle\ \texttt{(}\langle\text{name}\rangle\ \langle\text{interface}\rangle\texttt{)}^*\texttt{)} \\ & & \quad \langle\text{definition}\rangle^* \\ & & \quad \langle\text{name}\rangle\texttt{)} \\ & | & \texttt{(define } \langle\text{name}\rangle\ \texttt{(}\langle\text{name}\rangle\ \langle\text{name}\rangle^*\texttt{))} \end{array}$$

C.6 Compiling and Linking

Scheme 48 has a static linker that produces stand-alone heap images from a set of module descriptions. One specifies a particular procedure in a particular structure to be the image's startup procedure (entry point), and the linker traces dependency links as given by `open` and `access` clauses to determine the composition of the heap image.

There is not currently any provision for separate compilation; the only input to the static linker is source code. However, it will not be difficult to implement separate compilation. The unit of compilation is one module description (not one file). Any opened or accessed structures from which macros are obtained must be processed to the extent of extracting its macro definitions. The compiler knows from the interface of an opened or accessed structure which of its exports are macros. Except for macros, a module may be compiled without any knowledge of the implementation of its opened and accessed structures. However, inter-module optimization will be available as an option.

The main difficulty with separate compilation is resolution of auxiliary bindings introduced into macro expansions. The module compiler must transmit to the loader or linker the search path by which such bindings are to be resolved. In the case of the `delay` macro's auxiliary `make-promise` (see example above), the loader or linker needs to know that the desired binding of `make-promise` is the one apparent in `delay`'s defining environment, not in the environment being loaded or linked.

C.7 Semantics of Mutation

During program development it is often desirable to make changes to environments, structures, and interfaces. In static languages it may be necessary to recompile and re-link a program in order for such changes to be reflected in a running system. Even in interactive Common Lisp implementations, a change to a package's exports often requires reloading clients that have already mentioned names whose bindings change. Once `read` resolves a use of a name to a symbol, that resolution is fixed, so a change in the way that a name resolves to a symbol can only be reflected by re-`read`ing all such references.

The Scheme 48 development environment supports rapid turnaround in modular program development by allowing mutations to a program's module, and giving a clear semantics to such mutations. The rule is that variable bindings in a running program are always resolved according to current structure and interface bindings, even when these bindings change as a result of edits to the configuration. For example, consider the following:

```
(define-interface foo-interface (export a c))
(define-structure foo foo-interface
  (open scheme)
  (begin (define a 1)
         (define (b x) (+ a x))
         (define (c y) (* (b a) y))))
(define-structures bar (export d)
  (open scheme foo)
  (begin (define (d w) (+ (b w) a))))
```

This program has a bug. The variable `b`, which is free in the definition of `d`, has no binding in `bar`'s underlying environment. Suppose that `b` was supposed to be exported by `foo`, but was omitted from `foo-interface` by mistake. It is not necessary to re-process `bar` or any of `foo`'s other clients at this point. One need only change `foo-interface` and inform the development system of that one change (using, say, an appropriate Emacs command), and `foo`'s binding of `b` will be found when procedure `d` is called.

Similarly, it is also possible to replace a structure; clients of the old structure will be modified so that they see bindings from the new one. Shadowing is also supported in the same way. Suppose that a client environment C opens a structure `foo` that exports a name `x`, and `foo`'s implementation obtains the binding of `x` as an import from some other structure `bar`. Then C will see the binding from `bar`. If one then alters `foo` so that it shadows `bar`'s binding of `x` with a definition of its own, then procedures in C that reference `x` will automatically see `foo`'s definition instead of the one from `bar` that they saw earlier.

This semantics might appear to require a large amount of computation on every variable reference: The specified behavior requires scanning the environment's list of opened structures, examining their interfaces, on every variable reference, not just at compile time. However, the development environment uses caching with cache invalidation to make variable references fast.

C.8 Discussion

This module system was not designed as the be-all and end-all of Scheme module systems; it was only intended to help Richard Kelsey and me to organize the Scheme 48 system. Not only does the module system help avoid name clashes by keeping different subsystems in different namespaces, it has also helped us to tighten up and generalize Scheme 48's internal interfaces. Scheme 48 is unusual among Lisp implementations in admitting many different possible modes of operation. Examples of such multiple modes include the following:

- Linking can be either static or dynamic.
- The development environment (compiler, debugger, and command processor) can run either in the same address space as the program being developed or in a different address space. The environment and user program may even run on different processors under different operating systems[48].
- The virtual machine can be supported by either of two implementations of its implementation language, Prescheme.

The module system has been helpful in organizing these multiple modes. By forcing us to write down interfaces and module dependencies, the module system helps us to keep the system clean, or at least to keep us honest about how clean or not it is.

The need to make structures and interfaces second-class instead of first-class results from the requirements of static program analysis: it must be possible for the compiler and linker to expand macros and resolve variable bindings before the program is executed. Structures could be made first-class (as in FX[53]) if a type system were added to Scheme and the definitions of exported macros were defined in interfaces instead of in module bodies, but even in that case types and interfaces would remain second-class.

The prohibition on assignment to imported bindings makes substitution a valid optimization when a module is compiled as a block. The block compiler first scans the entire module body, noting which variables are assigned. Those that aren't assigned (only `define`d) may be assumed never assigned, even if they are exported. The optimizer can then perform a very simple-minded analysis to determine automatically that some procedures can and should have their calls compiled in line.

The programming style encouraged by the module system is consistent with the unextended Scheme language. Because module system features do not generally show up within module bodies, an individual module may be understood by someone who is not familiar with the module system. This is a great aid to code presentation and portability. If a few simple conditions are met (no name conflicts between module environments, no use of `structure-ref`, and use of `files` in preference to `begin`), then a multi-module program can be loaded into a Scheme implementation that does not support the module system. The Scheme 48 static linker satisfies these conditions, and can therefore run in other Scheme implementations. Scheme 48's bootstrap process, which is based on the static linker, is therefore nonincestuous. This contrasts with most other integrated programming environments, such as Smalltalk-80 [20], where the system can only be built using an existing version of the system itself.

Like ML modules, but unlike Scheme Xerox modules, this module system is compositional. That is, structures are constructed by single syntactic units that compose existing structures with a body of code. In Scheme Xerox, the set of modules that can contribute to an interface is open-ended — any module can contribute bindings to any interface whose name is in scope. The module system implementation is a cross-bar that channels definitions from modules to interfaces. The module system described here has simpler semantics and makes dependencies easier to trace. It also allows for higher-order modules, which Scheme Xerox considers unimportant.

Appendix D

Macros That Work

This paper reports on joint work with William Clinger, who is a co-author. Previously published in Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, *pages 155–162, January 1991. Reproduced with permission of the ACM.*

This paper describes a modified form of Kohlbecker's algorithm for reliably hygienic (capture-free) macro expansion in block-structured languages, where macros are source-to-source transformations specified using a high-level pattern language. Unlike previous algorithms, the modified algorithm runs in linear instead of quadratic time, copies few constants, does not assume that syntactic keywords (e.g. `if`) are reserved words, and allows local (scoped) macros to refer to lexical variables in a referentially transparent manner.

Syntactic closures have been advanced as an alternative to hygienic macro expansion. The problem with syntactic closures is that they are inherently low-level and therefore difficult to use correctly, especially when syntactic keywords are not reserved. It is impossible to construct a pattern-based, automatically hygienic macro system on top of syntactic closures because the pattern interpreter must be able to determine the syntactic role of an identifier (in order to close it in the correct syntactic environment) before macro expansion has made that role apparent.

Kohlbecker's algorithm may be viewed as a book-keeping technique for deferring such decisions until macro expansion is locally complete. Building on that insight, this paper unifies and extends the competing paradigms of hygienic macro expansion and syntactic closures to obtain an algorithm that combines the benefits of both.

Several prototypes of a complete macro system for Scheme have been based on the algorithm presented here.

D.1 Introduction

A macro is a source-to-source program transformation specified by programmers using a high-level pattern language. Macros add a useful degree of syntactic extensibility to a programming language, and provide a convenient way to abbreviate common programming idioms.

Macros are a prominent component of several popular programming languages, including C [25] and Common Lisp [57]. The macro systems provided by these languages suffer from various problems that make it difficult and in many cases impossible to write macros that work correctly regardless of the context in which the macros are used. It is widely believed that these problems result from the very nature of macro processing, so that macros are an inherently unreliable feature of programming languages [8].

That is not so. A macro system based upon the algorithm described in this paper allows reliable macros to be written, and is simple to use. The algorithm is nearly as fast as algorithms in current use.

This introduction illustrates several problems common to existing macro systems, using examples written in C and Scheme [11]. With one exception, all of these problems are solved automatically and reliably by the algorithm presented in this paper.

The original preprocessor for C allowed the body of a macro definition to be an arbitrary sequence of characters. It was therefore possible for a macro to expand into part of a token. For example,

```
#define foo "salad
printf(foo bar");
```

would expand into `printf("salad bar");`. We may say that this macro processor *failed to respect the integrity of lexical tokens.* This behavior caused so many problems that it has been banished from ANSI C except when special operators are used for the specific purpose of violating the integrity of lexical tokens.

Similar problems arise in macro systems that allow the body of a macro to expand into an arbitrary sequence of tokens. For example,

```
#define discriminant(a,b,c) b*b-4*a*c
  2*discriminant(x-y,x+y,x-y)*5
```

expands into `2*x+y*x+y-4*x-y*x-y*5`, which is then parsed as

```
(2*x)+(y*x)+y-(4*x)-(y*x)-(y*5)
```

instead of the more plausible

```
2*((x+y)*(x+y)-4*(x-y)*(x-y))*5.
```

We may say that systems such as the ANSI C preprocessor *fail to respect the structure of expressions.* Experienced C programmers know they must defend against this problem by placing parentheses around the macro body and around all uses of macro parameters, as in

```
#define discriminant(a,b,c) ((b)*(b)-4*(a)*(c))
```

This convention is not enforced by the C language, however, so the author of a macro who neglects this precaution is likely to create subtle bugs in programs that use the macro.

Another problem with the `discriminant` macro occurs when the actual parameter for `b` is an expression that has a side effect, as in `discriminant(3,x--,2)`. Of the problems listed in this introduction, the inappropriate duplication of side effects is the only one that is not solved automatically by our algorithm.

One can argue that actual parameter expressions should not have side effects, but this argument runs counter to the C culture. In a fully block-structured language such as Scheme, the author of a macro can defend against actual parameters with side effects by introducing a local variable that is initialized to the value obtained by referencing the macro parameter. In the syntax we adopt for this paper, which differs slightly from syntax that has been proposed for Scheme [11], a correct version of the `discriminant` macro can be defined by

```
(define-syntax discriminant
  (syntax-rules
    ((discriminant ?a ?b ?c)
     => (let ((temp ?b))
          (- (* temp temp) (* 4 ?a ?c))))))
```

Unfortunately there is no equivalent to this macro in C, since blocks are not permitted as expressions.

From the above example we see that macros must be able to introduce local variables. Local variables, however, make macros vulnerable to accidental conflicts of names, as in

```
#define swap(v,w) {int temp=(v); \
     (v) = (w); (w) = temp;}
  int temp = thermometer();
  if (temp < lo_temp) swap(temp, lo_temp);
```

Here the `if` statement *never* exchanges the values of `temp` and `lo_temp`, because of an accidental name clash. The macro writer's usual defense is to use an obscure name and *hope* that it will not be used by a client of the macro:

```
#define swap(v,w) {int __temp_obscure_name=(v); \
     (v) = (w); (w) = __temp_obscure_name;}
```

Even if the macro writer is careful to choose an obscure name, this defense is less than reliable. For example, the macro may be used within some other use of a macro whose author happened to choose the same obscure name. This problem can be especially perplexing to authors of recursive macros.

The most troublesome name clashes arise not from variables local to a macro, but from the very common case of a macro that needs to refer to a global variable or procedure:

```
#define complain(msg) print_error(msg);
  if (printer_broken()) {
      char *print_error = "The printer is broken";
      complain(print_error);
      }
```

Here the name given to the error message conflicts with the name of the procedure that prints the message. In this example the compiler will report a type error, but programmers are not always so lucky.

In most macro systems there is no good way to work around this problem. Programs would become less readable if obscure names were chosen for global variables and procedures, and the author of a macro probably does not have the right to choose those names anyway. Requiring the client of each macro to be aware of the names that occur free in the macro would defeat the purpose of macros as syntactic abstractions.

The examples above show that at least two classes of inadvertent name conflicts can be created by macro processors that *fail to respect the correlation between bindings and uses of names*. One class involves macros that introduce bound (local) variables. Another class involves macros that contain free references to variables or procedures.

Hygienic macro expansion [34] is a technology that maintains the correlation between bindings and uses of names, while respecting the structure of tokens and expressions. The main problem with hygienic macro expansion has been that the algorithms have required $O(n^2)$ time, where n is the time required by naive macro expansion as in C or Common Lisp. Although this is the worst case, it sometimes occurs in practice.

The purpose of this paper is to build on Kohlbecker's algorithm for hygienic macro expansion by incorporating ideas developed within the framework of syntactic closures [5]. Our result is an efficient and more complete solution to the problems of macro processing. Our algorithm runs in $O(n)$ time and solves several additional problems associated with local (scoped) macros that had not been addressed by hygienic macro expansion. In particular, our algorithm supports referentially transparent local macros as described below.

If macros can be declared within the scope of lexical variables, as in Common Lisp, then a seemingly new set of problems arises. We would like for free variables that occur on the right hand side of a rewriting rule for a macro to be resolved in the lexical environment of the macro definition instead of being resolved in the lexical environment of the use of the macro. Using `let-syntax` to declare local macros, for example, it would be nice if

```
(let-syntax ((first
              (syntax-rules
                ((first ?x) => (car ?x))))
             (second
              (syntax-rules
                ((second ?x) => (car (cdr ?x))))))
  (let ((car "duesenberg"))
    (let-syntax ((classic
                  (syntax-rules
                    ((classic) => car))))
      (let ((car "yugo"))
        (let-syntax ((affordable
                      (syntax-rules
                        ((affordable) => car))))
          (let ((cars (list (classic)
                            (affordable))))
            (list (second cars)
                  (first cars))))))))
```

could be made to evaluate to `("yugo" "duesenberg")`. This would be the case if the references to `car` that are introduced by the `first`, `second`, `classic`, and `affordable` macros referred in each case to the `car` variable within whose scope the macro was defined. In other words, this example would evaluate to `("yugo" "duesenberg")` if local macros were *referentially transparent*.

Our algorithm supports referentially transparent local macros. We should note that these macros may not be referentially transparent in the traditional sense because of side effects and other problems caused by the programming language whose syntax they extend, but we blame the language for this and say that the macros themselves are referentially transparent.

Now consider the following Scheme code:

```
(define-syntax push
  (syntax-rules
    ((push ?v ?x) => (set! ?x (cons ?v ?x)))))
(let ((pros (list "cheap" "fast"))
      (cons (list)))
  (push "unreliable" cons))
```

It is easy to see that the `cons` procedure referred to within the macro body is different from the `cons` variable within whose scope the macro is used. Through its reliance on the meaning of the syntactic keyword `set!`, the `push` macro also illustrates yet a third class of inadvertent name conflicts. If syntactic keywords are not reserved, then the `push` macro might be used within the scope of a local variable or macro named `set!`.

We can now identify four classes of *capturing* problems. The first class involves inadvertent capturing by bound variables introduced by a macro. The second class involves the inadvertent capture of free variable references introduced by a macro. The third is like the first, but involves inadvertent capturing by bindings of syntactic keywords (i.e. local macros) introduced by a macro. The fourth is like the second, but involves the inadvertent capture of references to syntactic keywords introduced by a macro.

Syntactic closures [5] have been advanced as an alternative to hygienic macro expansion. Syntactic closures can be used to eliminate all four classes of capturing problems, and can support referentially transparent local macros as well. Why then have we bothered to develop a new algorithm for hygienic macro expansion, when syntactic closures also run in linear time and solve all the problems that are solved by our algorithm?

The problem with syntactic closures is that they do not permit macros to be defined using the kind of high-level pattern language that is used in C and is proposed for Scheme [24]. The difficulty is that a pattern interpreter that uses syntactic closures must be able to determine the syntactic role of an identifier (to know whether to close over it, and if so to know the correct syntactic environment) before macro expansion has made that role apparent. Consider a simplified form of the `let` macro, which introduces local variables:

```
(define-syntax let
  (syntax-rules
    ((let ((?name ?val)) ?body)
     => ((lambda (?name) ?body) ?val))))
```

When this macro is used, the `?val` expression must be closed in the syntactic environment of the use, but the `?body` cannot simply be closed in the syntactic environment of the use because its references to `?name` must be left free. The pattern interpreter cannot make this distinction unaided until the lambda expression is expanded, and even then it must somehow remember that the bound variable of the lambda expression came from the original source code and must therefore be permitted to capture the references that occur in `?body`. Recall from the `swap` example that a reliable macro system must *not* permit bound variables introduced by macro expansion to capture such references.

Kohlbecker's algorithm can be viewed as a book-keeping technique for deferring all such decisions about the syntactic roles of identifiers until macro expansion is locally complete and has made those roles evident. Our algorithm is therefore based on Kohlbecker's but incorporates the idea of syntactic environments, taken from the work on syntactic closures, in order to achieve referentially transparent local macros.

The question arises: Don't the capturing problems occur simply because we use names instead of pointers, and trees instead of dags? If we first parsed completely before doing macro expansion, could we not replace all occurrences of names by pointers to symbol table entries and thereby eliminate the capturing problems?

The answer is no. One of the most important uses of macros is to provide syntactically convenient binding abstractions. The distinction between binding occurrences and other uses of identifiers should be determined by a particular syntactic abstraction, not predetermined by the parser. Therefore, even if we were to parse actual

$$
\begin{aligned}
\text{env} &= \text{identifier} \to \text{denotation} \\
\text{denotation} &= \text{special} + \text{macro} + \text{identifier} \\
\text{special} &= \{\textit{lambda}, \textit{let-syntax}\} \\
\text{macro} &= (\text{pattern} \times \text{rewrite})^{+} \times \text{env}
\end{aligned}
$$

$$
\begin{aligned}
\textit{ident} &\in \text{env} \\
\textit{lookup} &\in \text{env} \times \text{identifier} \to \text{denotation} \\
\textit{bind} &\in \text{env} \times \text{identifier} \times \text{denotation} \to \text{env} \\
\textit{divert} &\in \text{env} \times \text{env} \to \text{env}
\end{aligned}
$$

$$
\begin{aligned}
\textit{lookup}\,(\textit{ident}, x) &= x \\
\textit{lookup}\,(\textit{bind}\,(e, x, v), x) &= v \\
\textit{lookup}\,(\textit{bind}\,(e, x, v), y) &= \textit{lookup}\,(e, y) \qquad \text{if } x \neq y \\
\textit{divert}\,(e, \textit{ident}) &= e \\
\textit{divert}\,(e, \textit{bind}\,(e', x, v)) &= \textit{bind}\,(\textit{divert}\,(e, e'), x, v)
\end{aligned}
$$

Figure D-1: Syntactic environments.

parameters into expression trees before macro expansion, the macro processor still could not reliably distinguish bindings from uses and correlate them until macro expansion is performed.

Consider an analogy from lambda calculus. In reducing an expression to normal form by textual substitution, it is sometimes necessary to rename variables as part of a beta reduction. It doesn't work to perform all the (naive) beta reductions first, without renaming, and then to perform all the necessary alpha conversions; by then it is too late. Nor does it work to do all the alpha conversions first, because beta reductions introduce new opportunities for name clashes. The renamings must be *interleaved* with the (naive) beta reductions, which is the reason why the notion of substitution required by the non-naive beta rule is so complicated.

The same situation holds for macro expansions. It does not work to simply expand all macro calls and then rename variables, nor can the renamings be performed before expansion. The two processes must be interleaved in an appropriate manner. A correct and efficient realization of this interleaving is our primary contribution.

D.2 The Algorithm

Our algorithm avoids inadvertent captures by guaranteeing the following *hygiene condition*:

$$\frac{lookup\,(e, \mathbf{x}) \in \text{identifier}}{e \vdash \mathbf{x} \rightarrow lookup\,(e, \mathbf{x})} \qquad [\textit{variable references}]$$

$$\frac{\begin{array}{c} lookup\,(e, \mathbf{k}_0) = lambda \\ bind\,(e, \mathbf{x}, \mathbf{x}') \vdash \mathbf{E} \rightarrow \mathbf{E}' \text{ (where } \mathbf{x}' \text{ is a fresh identifier)} \end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \texttt{(}\mathbf{x}\texttt{)}\ \mathbf{E}\texttt{)} \rightarrow \texttt{(lambda (}\mathbf{x}'\texttt{)}\ \mathbf{E}'\texttt{)}} \qquad [\textit{procedure abstractions}]$$

$$\frac{e \vdash \mathbf{E}_0 \rightarrow \mathbf{E}_0', \quad e \vdash \mathbf{E}_1 \rightarrow \mathbf{E}_1'}{e \vdash \texttt{(}\mathbf{E}_0\ \ \mathbf{E}_1\texttt{)} \rightarrow \texttt{(}\mathbf{E}_0'\ \ \mathbf{E}_1'\texttt{)}} \qquad [\textit{procedure calls}]$$

$$\frac{\begin{array}{c} lookup\,(e, \mathbf{k}_0) = let\text{-}syntax \\ bind\,(e, \mathbf{k}, \langle \tau, e \rangle) \vdash \mathbf{E} \rightarrow \mathbf{E}' \end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \texttt{((}\mathbf{k}\ \tau\texttt{))}\ \mathbf{E}\texttt{)} \rightarrow \mathbf{E}'} \qquad [\textit{macro abstractions}]$$

$$\frac{\begin{array}{c} lookup\,(e, \mathbf{k}) = \langle \tau, e' \rangle \\ transcribe\,(\texttt{(}\mathbf{k}\ \ldots\texttt{)}, \tau, e, e') = \langle \mathbf{E}, e'' \rangle \\ e'' \vdash \mathbf{E} \rightarrow \mathbf{E}' \end{array}}{e \vdash \texttt{(}\mathbf{k}\ \ldots\texttt{)} \rightarrow \mathbf{E}'} \qquad [\textit{macro calls}]$$

Figure D-2: The modified Kohlbecker algorithm.

$$\frac{\begin{array}{c} match\,(\mathbf{E}, \pi, e_{use}, e_{def}) = nomatch \\ transcribe\,(\mathbf{E}, \tau', e_{use}, e_{def}) = \langle \mathbf{E}', e' \rangle \end{array}}{transcribe\,(\mathbf{E}, \langle \langle \pi, \rho \rangle, \tau' \rangle, e_{use}, e_{def}) = \langle \mathbf{E}', e' \rangle}$$

$$\frac{\begin{array}{c} match\,(\mathbf{E}, \pi, e_{use}, e_{def}) = \sigma \\ rewrite\,(\rho, \sigma, e_{def}) = \langle \mathbf{E}', e_{new} \rangle \end{array}}{transcribe\,(\mathbf{E}, \langle \langle \pi, \rho \rangle, \tau' \rangle, e_{use}, e_{def}) = \langle \mathbf{E}', divert\,(e_{use}, e_{new}) \rangle}$$

$transcribe\,(\mathbf{E}, \langle\ \rangle, e_{use}, e_{def})$ is an error.

Figure D-3: Definition of $transcribe\,(\mathbf{E}, \tau, e_{use}, e_{def})$.

1. It is impossible to write a high-level macro that inserts a binding that can capture references other than those inserted by the macro.

2. It is impossible to write a high-level macro that inserts a reference that can be captured by bindings other than those inserted by the macro.

These two properties can be taken as the defining properties of hygienic macro expansion. The hygiene condition is quite strong, and it is sometimes necessary to write non-hygienic macros; for example, the `while` construct in C implicitly binds `break`, so if `while` were implemented as a macro then it would have to be allowed to capture references to `break` that it did not insert. We here ignore the occasional need to escape from hygiene; we have implemented a compatible low-level macro system in which non-hygienic macros can be written.

The reason that previous algorithms for hygienic macro expansion are quadratic in time is that they expand each use of a macro by performing a naive expansion followed by an extra scan of the expanded code to find and paint (i.e. rename, or time-stamp) the newly introduced identifiers. If macros expand into uses of still other macros with more or less the same actual parameters, which often happens, then large fragments of code may be scanned anew each time a macro is expanded. Naive macro expansion would scan each fragment of code only once, when the fragment is itself expanded.

Our algorithm runs in linear time because it finds the newly introduced identifiers by scanning the rewrite rules, and paints these identifiers as they are introduced during macro expansion. The algorithm therefore scans the expanded code but once, for the purpose of completing the recursive expansion of the code tree, just as in the naive macro expansion algorithm. The newly introduced identifiers can in fact be determined by scanning the rewrite rules at macro definition time, but this does not affect the asymptotic complexity of the algorithm.

The more fundamental difference between our algorithm and previous algorithms lies in the book-keeping needed to support referentially transparent local macros. The syntactic environments manipulated by this book-keeping are shown in Figure D-1. These are essentially the same as the environments used by syntactic closures, but the book-keeping performed by our algorithm allows it to defer decisions that involve the syntactic roles of identifiers.

The overall structure of the algorithm is shown formally in Figure D-2 for the lambda calculus subset of Scheme, extended by macro calls and a restricted form of `let-syntax`. The notation "$e \vdash \mathbf{E} \rightarrow \mathbf{E}'$" indicates that $\mathbf{E}'$ is the completely macro-expanded expression that results from expanding $\mathbf{E}$ in the syntactic environment e. For this simple language the initial syntactic environment e_{init} is

$$bind(bind(ident, \texttt{lambda}, lambda), \texttt{let-syntax}, let\text{-}syntax).$$

The variables bound by a procedure abstraction are renamed to avoid shadowing outer variables. Identifiers that denote variable references must therefore be renamed as well. Procedure calls are straightforward (but would be less so if expressions were allowed to expand into syntactic keywords; we have implemented this both ways). The rule for macro abstractions is also straightforward: the body is macro-expanded in a syntactic environment in which the syntactic keyword bound by the abstraction denotes the macro obtained from the transformation function τ (a set of rewrite rules) by closing τ in the syntactic environment of the macro definition.

The rule for macro calls is subtle. The macro call is transcribed using the transformation function τ obtained from the macro being called, the syntactic environment of the call, and the syntactic environment in which the macro was defined. This transcription yields not only a new expression $\mathbf{E}$ but a new syntactic environment e'' in which to complete the macro expansion. The key fact about algorithms for hygienic macro expansion is that any identifiers introduced by the macro are renamed,

$$match(\mathbf{E}, \mathbf{?v}, e_{use}, e_{def}) = \{\mathbf{?v} \mapsto \mathbf{E}\}$$

$$\frac{lookup(e_{def}, \mathbf{x}') = lookup(e_{use}, \mathbf{x})}{match(\mathbf{x}, \mathbf{x}', e_{use}, e_{def}) = \{\}}$$

$$\frac{\begin{array}{c} match(\mathbf{E}_i, \pi_i, e_{use}, e_{def}) = \sigma_i,\ i = 1, \ldots, n \\ \sigma_i \neq nomatch,\ i = 1, \ldots, n \end{array}}{match((\mathbf{E}_1 \ldots \mathbf{E}_n), (\pi_1 \ldots \pi_n), e_{use}, e_{def}) = \sigma_1 \cup \ldots \cup \sigma_n}$$

$$match(\mathbf{E}, \pi, e_{use}, e_{def}) = nomatch,$$
$$\text{if the other rules are not applicable}$$

Figure D-4: Definition of $match(\mathbf{E}, \pi, e_{use}, e_{def})$.

so the macro can only introduce fresh identifiers. The new syntactic environment binds these fresh identifiers to the denotations of the corresponding original identifiers in the syntactic environment of the macro definition. These will be the ultimate denotations of the fresh identifiers unless subsequent macro expansion exposes an intervening procedure abstraction that binds them. Since the identifiers were fresh, any such binding must have been introduced by the same macro call that introduced the references. Hence the hygiene condition will be satisfied.

Figure D-3 defines the *transcribe* function in terms of *match* and *rewrite*. The *transcribe* function simply loops through each rewrite rule π `=>` ρ until it finds one whose pattern π matches the macro call. It then rewrites the macro call according to ρ and adds new bindings for any identifiers introduced by the rewrite to the syntactic environment of the use.

Actually, rather than matching the entire macro call to the pattern, it is sufficient to match only the tail (or "actual parameters") of the call against the tail ("formal parameters") of the pattern. This complication is not reflected in Figure D-3.

The *match* function (Figure D-4) delivers a substitution σ mapping pattern variables to components of the input expression. It is quite conventional except for one important wrinkle: Matching an identifier x in the macro use against a literal identifier y in the pattern succeeds if and only if $lookup(e_{use}, x) = lookup(e_{def}, y)$. This implies that identifiers in patterns are lexically scoped: bindings that intervene between the definition and use of a macro may cause match failure.

For simplicity, the definition of *match* in Figure D-4 assumes that no pattern variable is duplicated.

The *rewrite* function (Figure D-5) rewrites the macro call using the substitution σ generated by the matcher. It is also responsible for choosing the fresh identifiers that replace those that appear in the output pattern and establishing their proper denotations. *transcribe* uses *divert* to add these new bindings to the environment in which the macro output is expanded.

Given constant-time operations on environments, this algorithm takes $O(n)$ time, where n is the time required by naive macro expansion.

$$rewrite(\rho, \sigma, e_{def}) = \langle rewrite'(\rho, \sigma'), e_{new} \rangle,$$

where

$\sigma' = \sigma \cup \{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$,
$x_1, \ldots, x_n$ are all the identifiers that occur in ρ,
$x'_1, \ldots x'_n$ are fresh identifiers,
$e_{new} = bind(\ldots bind(ident, x'_1, d_1) \ldots, x'_n, d_n)$,
and $d_i = lookup(e_{def}, x_i)$ for $i = 1, \ldots, n$.

$$rewrite'(\mathbf{?v}, \sigma) = \sigma(\mathbf{?v})$$

$$rewrite'(\mathbf{x}, \sigma) = \sigma(\mathbf{x})$$

$$rewrite'(\mathbf{(}\pi_1 \ldots \pi_n\mathbf{)}, \sigma) = \mathbf{(E}'_1 \ldots \mathbf{E}'_n\mathbf{)},$$

where $\mathbf{E}'_i = rewrite'(\pi_i, \sigma)$, $i = 1, \ldots, n$.

Figure D-5: Definition of $rewrite(\rho, \sigma, e_{def})$.

D.3 Examples

To make the examples easier to follow, we'll assume that `let` is understood primitively by the expansion algorithm, in a manner similar to `lambda`. The initial environment e_{init} thus has a nontrivial binding for the identifier `let` in addition to `lambda` and `let-syntax`.

Example 1. Consider the problem of expanding the following expression in the initial syntactic environment:

```
(let-syntax ((push (syntax-rules
                     ((push ?v ?x)
                      => (set! ?x (cons ?v ?x))))))
  (let ((pros (list "cheap" "fast"))
        (cons (list)))
    (push "unreliable" cons)))
```

The rule for macro abstractions applies, because

$$lookup(e_{init}, \texttt{let-syntax}) = let\text{-}syntax.$$

The next step is to expand the body of the `let-syntax` expression in an augmented syntactic environment

$$e_1 = bind(e_{init}, \texttt{push}, \langle \tau, e_{init} \rangle)$$

where

$$\tau = \langle\langle \texttt{(?v ?x)}, \texttt{(set! ?x (cons ?v ?x))}\rangle\rangle$$

The pattern is `( ?v ?x)`, rather than `(push ?v ?x)`, because the rule will only be examined when the fact that the head of the form denotes the `push` macro is already apparent. Thus there is never a need to match the head of the pattern with the head of the expression to be expanded.

The initializers in the `let` expression expand to themselves, because $e_1 \vdash \texttt{list} \rightarrow \texttt{list}$. (Recall that e_{init}, and therefore e_1 as well, is based on the identity environment *ident*.)

Next we expand the body of the `let` expression in the augmented environment

$$e_2 = \mathit{bind}(\mathit{bind}(e_1, \texttt{cons}, \texttt{cons.1}), \texttt{pros}, \texttt{pros.1})$$

where `pros.1` and `cons.1` are fresh identifiers. In e_2, the identifier `push` denotes the macro $\langle \tau, e_{init}\rangle$, so the rule for macro calls applies to the `push` expression. We now compute

$$\mathit{transcribe}(\texttt{(push "unreliable" cons)}, \tau, e_2, e_{init})$$

τ consists of only one rule, which matches the input:

$$\begin{aligned}&\mathit{match}(\texttt{("unreliable" cons)}, \texttt{(?v ?x)}, e_2, e_{init})\\ &\quad = \{\texttt{?v} \mapsto \texttt{"unreliable"},\ \texttt{?x} \mapsto \texttt{cons}\}\end{aligned}$$

Note that *match* made no use of its two environment arguments, because the pattern `( ?v ?x)` didn't contain any identifiers. The `cond` example below gives a situation in which these environments are used.

Now we compute the replacement expression:

$$\begin{aligned}&\mathit{rewrite}(\texttt{(set! ?x (cons ?v ?x))},\\ &\qquad\quad \{\texttt{?v} \mapsto \texttt{"unreliable"},\ \texttt{?x} \mapsto \texttt{cons}\},\\ &\qquad\quad e_{init})\\ &\quad = \langle \texttt{(set!.2 cons (cons.2 "unreliable" cons))}, e_3\rangle\end{aligned}$$

where

$$e_3 = \mathit{bind}(\mathit{bind}(\mathit{ident}, \texttt{set!.2}, \texttt{set!}), \texttt{cons.2}, \texttt{cons})$$

and `set!.2` and `cons.2` are fresh identifiers. *transcribe* now delivers the rewritten expression together with an environment

$$\begin{aligned}e_4 &= \mathit{divert}(e_2, e_3)\\ &= \mathit{bind}(\mathit{bind}(e_2, \texttt{set!.2}, \texttt{set!}), \texttt{cons.2}, \texttt{cons})\end{aligned}$$

that gives bindings to use in expanding the replacement expression. The environment e_4 takes `set!.2` to `set!`, `cons.2` to `cons`, and `cons` to `cons.1`, so

$$\begin{aligned}e_4 \vdash\ &\texttt{(set!.2 cons (cons.2 "unreliable" cons))}\\ &\rightarrow \texttt{(set! cons.1 (cons "unreliable" cons.1))}\end{aligned}$$

The final result is

```
(let ((pros.1 (list "cheap" "fast"))
      (cons.1 (list)))
  (set! cons.1 (cons "unreliable" cons.1)))
```

Example 2. This example illustrates reliable reference to local variables that are in scope where the macro is defined.

```
(let ((x "outer"))
  (let-syntax ((m (syntax-rules
                   ((m) => x))))
    (let ((x "inner"))
      (m))))
```

To expand this expression in e_{init}, a fresh identifier `x.1` is chosen to replace the outer `x`, and the `let-syntax` expression is expanded in the syntactic environment

$$e_1 = \mathit{bind}(e_{init}, \mathtt{x}, \mathtt{x.1})$$

This leads to expanding the inner `let` expression in the syntactic environment

$$e_2 = \mathit{bind}(e_1, \mathtt{m}, \langle\langle\langle\mathtt{(\ \ )}, \mathtt{x}\rangle\rangle, e_1\rangle)$$

Finally a fresh identifer `x.2` is chosen to replace the inner `x`, and `(m)` is expanded in the syntactic environment

$$e_3 = \mathit{bind}(e_2, \mathtt{x}, \mathtt{x.2})$$

Now

$$\mathit{transcribe}(\mathtt{(m)}, \langle\langle\langle\mathtt{(\ \ )}\rangle, \mathtt{x}\rangle\rangle, e_3, e_1) = \langle\mathtt{x.3}, \mathit{bind}(e_3, \mathtt{x.3}, \mathtt{x.1})\rangle$$

where `x.3` is a fresh identifer introduced for the right-hand side of the rewrite rule for the macro `m`. The denotation of `x.3` is the denotation of `x` in the environment of `m`'s definition, which is `x.1`. The final expansion is

```
(let ((x.1 "outer"))
  (let ((x.2 "inner"))
    x.1))
```

Example 3. This example illustrates lexical scoping of constant identifiers that occur in the left-hand side of rewrite rules. Following [35], we adopt the use of an ellipsis token `...` as part of the syntax of patterns, not as a meta-notation indicating that something has been elided from this example.

```
(define-syntax cond
  (syntax-rules
    ((cond) => #f)
    ((cond (else ?result ...) ?clause ...)
     => (begin ?result ...))
    ((cond (?test) ?clause ...)
     => (or ?test (cond ?clause ...)))
    ((cond (?test ?result ...) ?clause ...)
     => (if ?test
            (begin ?result ...)
            (cond ?clause ...)))))
```

The second pattern for this macro contains a fixed identifier `else`. This will only match a given identifer x in a use of `cond` if the denotation of x in the environment of use matches the denotation of `else` in the environment of `cond`'s definition. Typically x is `else` and is self-denoting in both environments. However, consider the following:

```
(let ((else #f))
  (cond (#f 3)
        (else 4)
        (#t 5)))
```

Expanding this expression requires calculating $\mathit{match}(\texttt{else}, \texttt{else}, e_{use}, e_{def})$ where

$$\begin{aligned} \mathit{lookup}(e_{use}, \texttt{else}) &= \texttt{else.1}, \\ \mathit{lookup}(e_{def}, \texttt{else}) &= \texttt{else}. \end{aligned}$$

Thus

$$\mathit{match}(\texttt{else}, \texttt{else}, e_{use}, e_{def}) = \mathit{nomatch}$$

and the final expansion is

```
(let ((else.1 #f))
  (if #f (begin 3)
      (if else.1 (begin 4)
          (if #t (begin 5) #f))))
```

D.4 Integration Issues

We emphasize that our algorithm is suitable for use with any block-structured language, and does not depend on the representation of programs as lists in Scheme. Scheme's representation is especially convenient, however. This section explains how the algorithm can be made to work with less convenient representations.

The simplicity of the *match* function defined in Figure 4 results from the regularity of a Cambridge Polish syntax. For Algol-like syntaxes the matcher could be much more complicated. To avoid such complications, a macro system may eschew complex patterns and may specify a fixed syntax for macro calls.

Our algorithm must understand the structure of the source program, so Algol-like syntaxes require that the algorithm be integrated with a parser. If the macro language is sufficiently restricted, then it may be possible to parse the input program completely before macro expansion is performed. If the actual parameters of a macro call need to be transmitted to the macro in unparsed form, however, then parsing will have to be interleaved with the macro expansion algorithm.

For the algorithm to work at all, the parser must be able to locate the end of any macro definition or macro use. If parentheses are used to surround the actual parameters of a macro call, for example, then mismatched parentheses within an actual parameter cannot be tolerated unless they are somehow marked as such, perhaps by an escape character. This is a fundamental limitation on the generality of the syntactic transformations that can be described using a macro system based on our algorithm.

This is not a particularly burdensome limitation. For example, it is possible to design a hygienic macro system for C that allows the `for` construct to be described as a macro.

Experience with macros in Lisp and related languages has shown that macros are most useful when local variables can be introduced in any statement or expression context. Most Algol-like languages are not fully block-structured in this sense. C, for example, does not admit blocks as expressions, while Pascal and Modula-2 do not even admit blocks as statements. Fortunately this particular shortcoming can usually be overcome by the macro processor. Macros can be written as though the language admits blocks in all sensible contexts, and the macro processor itself can be responsible for replacing these blocks by their bodies while lifting all local declarations to the head of the procedure body within which the declarations appear.

In our algorithm, the matcher compares an identifier in the pattern against an identifier in the input by comparing their denotations. This makes it difficult to use such a sophisticated matcher in languages such as Common Lisp, where an identifier may denote many different things at once, and where the overloading is resolved by syntactic context. The problem is that the matcher cannot reliably determine the syntactic context in which the identifier will ultimately appear. One solution is to ban identifiers from patterns. Another is to resolve the overloading by relying on declarations provided by the author of the macro.

D.5 Previous Work, Current Status, Future Work

The problems with naive macro expansion have been recognized for many years, as have the traditional work-arounds [8]. The capturing problems that afflict macro systems for block-structured languages were first solved by Kohlbecker's work on hygienic macro expansion. Bawden and Rees then proposed syntactic closures [5] as a more general and efficient but lower-level solution. Our algorithm unifies and extends this recent research, most of which has been directed toward the goal of developing a reliable macro system for Scheme.

At the 1988 meeting of the Scheme Report authors at Snowbird, Utah, a macro committee was charged with developing a hygienic macro facility akin to `extend-syntax` [16] but based on syntactic closures. Chris Hanson implemented a prototype and discovered that an implementation based on syntactic closures must determine the syntactic roles of some identifiers before macro expansion based on textual pattern matching can make those roles apparent [24]. Clinger observed that Kohlbecker's algorithm amounts to a technique for delaying this determination, and proposed a linear-time version of Kohlbecker's algorithm. Rees married syntactic closures to the modified Kohlbecker's algorithm and implemented it all, twice. Bob Hieb found some first implementation and proposed fixes.

A high-level macro system similar to that described here is currently implemented on top of a compatible low-level system that is not described in this paper. Bob Hieb and Kent Dybvig have redesigned this low-level system to make it more abstract and easier to use, and have constructed yet another implementation. It is expected that both the high-level and low-level macro facilities will be described in a future report on Scheme [11].

Some problems remain. The algorithm we have described treats identifiers uniformly, but identifiers in Scheme are lexically indistinguishable from symbols that appear in constants. Consequently any symbols introduced by a macro will be renamed just as if they were identifiers, and must therefore be reverted after macro expansion has revealed that they are part of a constant. This means that constants introduced by a macro may have to be copied. In this respect our algorithm improves upon Kohlbecker's, which copied *all* constants, but is still not ideal.

More significantly, the pattern variables used to define macros might be lexically indistinguishable from identifiers and symbols. In order for macros that define other macros to remain referentially transparent, pattern variables must not be reverted to their original names even though they are represented as symbols in our existing implementation. We are not completely certain that this refinement eliminates all problems with macros that define other macros.

One project we intend to pursue is to integrate our algorithm with a module system for Scheme such as that described in [13]. For example, it should be possible for a module to export a macro without also having to export bindings needed by the macro's expansions.

An obvious application for this research is to develop better macro facilities for other block-structured languages such as Modula-2.

Acknowledgements

The authors thank an anonymous member of the program committee who helped us to write the introduction. Jim O'Toole and Mark Sheldon provided helpful comments on drafts of the paper.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of The Department of Defense under the Office of Naval Research contract N00014-89-J-3202.

Appendix E

Scheme 48 User's Guide

Scheme 48 is an implementation of the Scheme programming language as described in the Revised[4] Report on the Algorithmic Language Scheme. It is based on a compiler and interpreter for a virtual Scheme machine. The name derives from our desire to have an implementation that is simple and lucid enough that it looks as if it were written in just 48 hours. We don't claim to have reached that stage yet; much more simplification is necessary.

Scheme 48 tries to be faithful to the upcoming Revised[5] Scheme Report, providing neither more nor less in the initial user environment. (This is not to say that more isn't available in other environments; see below.) Support for numbers is weak: bignums are slow and floating point is almost nonexistent (see description of floatnums, below). `define-syntax`, `let-syntax`, `letrec-syntax`, and `syntax-rules` are supported, but not the rest of the Revised[4] Scheme Report's appendix's macro proposal.

E.1 Invoking the Virtual Machine

A few command line arguments are processed by the virtual machine as it starts up.

`scheme48 [-i` *image*`] [-h` *heapsize*`] [-o` *filename*`] [-s` *stacksize*`]`
`[-a` *argument* `...]`

`-i` *image*

specifies a heap image file to resume. *image* defaults to a heap image that runs a Scheme command processor. Heap images are created by the `,dump` and `,build` commands, for which see below.

`-h` *heapsize*

specifies how much space should be reserved for allocation. *Heapsize* is in words (where one word = 4 bytes), and covers both semispaces, only one of which is in use at any given time (except during garbage collection). Cons cells are currently 3 words, so if you want to make sure you can allocate a million cons cells, you should specify `-h 6000000` (actually somewhat more than this, to account for the initial heap image and breathing room).

`-s` *stacksize*

specifies how much space should be reserved for the continuation and environment stack. If this space is exhausted, continuations and environments are copied to the heap. *Stacksize* is in words and defaults to 2500.

`-o` *filename*

This switch specifies an executable file in which foreign identifiers can be looked up for the foreign function interface. *Filename* should be the file that contains the scheme48vm executable image.

`-a` *argument* ...

is only useful with images built using `,build`. The *arguments* are passed as a list to the procedure specified in the `,build` command. E.g.

```
> (define (go a)
    (for-each display a) (newline) 0)
> ,build go foo.image
> ,exit
% scheme48vm -i foo.image -a mumble "foo x"
mumblefoo x
%
```

The usual definition of the `s48` or `scheme48` command is actually a shell script that starts up the virtual machine with a `-i` argument specifying the development environment heap image, and a `-o` argument specifying the location of the virtual machine.

E.2 Command Processor

While it is possible to use the Scheme 48 static linker for program development, it is more convenient to use the development environment, which supports rapid turnaround for program changes. The programmer interacts with the development environment through a *command processor*. The command processor is like the usual Lisp read-eval-print loop in that it accepts Scheme forms to evaluate. However, all meta-level operations, such as exiting the Scheme system or requests for trace output, are handled by *commands,* which are lexically distinguished from Scheme forms. This arrangement is borrowed from the Symbolics Lisp Machine system, and is reminiscent of non-Lisp debuggers. Commands are a little easier to type than Scheme forms (no parentheses, so you don't have to shift), but more importantly, making them distinct from Scheme forms ensures that programs' namespaces aren't clutterred with inappropriate bindings. Equivalently, the command set is available for use regardless of what bindings happen to be visible in the current program. This is especially important in conjunction with the module system, which puts strict controls on visibility of bindings.

E.2.1 Logistical Commands

`,load` *filename* ...

Load Scheme source file(s). This command is similar to `(load "`*filename*`")`. Unlike that expression, however, it works properly even in environments in which the variable `load` is defined properly.

`,exit` [*exp*]

Exit back out to shell (or executive or whatever invoked Scheme 48 in the first place). *Exp* should evaluate to an integer. The integer is returned to the calling program. (On Unix, 0 is generally interpreted as success, nonzero as failure.)

E.2.2 Module System Commands

The Scheme 48 command processor supports the module system with a variety of special commands. For commands that require structure names, these names are resolved in a designated configuration environment that is distinct from the environment used to evaluate Scheme forms given to the command processor. The command processor interprets Scheme forms in a particular current environment, and there are commands that move the command processor between different environments.

Commands are introduced by a comma (`,`) and end at the end of line. The command processor's prompt consists of the name of the current environment followed by a greater-than (`>`).

`,config`

The `,config` command sets the command processor's current environment to be the current configuration environment. Forms entered at this point are interpreted as being configuration language forms, not Scheme forms.

`,in` *struct-name*

The `,in` command moves the command processor to a specified structure's underlying environment. For example:

```
user> ,config
config> (define-structure foo (export a)
          (open scheme))
config> ,in foo
foo> (define a 13)
foo> a
13
```

In this example the command processor starts in an environment called `user`. The `,config` command moves it into the configuration environment, which has the name `config`. The `define-structure` form binds, in `config`, the name `foo` to a structure that exports `a`. Finally, the command `,in foo` moves the command processor into structure `foo`'s underlying environment.

An module's body isn't executed (evaluated) until the environment is *loaded*, which is accomplished by the `,load-package` command.

`,load-package` *struct-name*

The `,load-package` command ensures that the specified structure's underlying environment's program has been loaded. This consists of (1) recursively ensuring that the environments of any opened or accessed structures are loaded, followed by (2) executing the environment's body as specified by its definition's `begin` and `files` forms.

`,reload-package` *struct-name*

This command re-executes the structure's environment's program. It is most useful if the program comes from a file or files, when it will update the environment's bindings after mutations to its source file.

`,load` *filespec* ...

The `,load` command executes forms from the specified file or files in the current environment. `,load` *filespec* is similar to `(load "`*filespec*`")` except that the name `load` needn't be bound in the current environment to Scheme's `load` procedure.

`,structure` *name interface*

The `,structure` command defines *name* in the configuration environment to be a structure with interface *interface* based on the current environment.

`,open` *struct-name*

The `,open` command opens a new structure in the current environment, as if the environment definition's `open` clause had listed *struct-name*.

A number of other commands are not strictly necessary, but they capture common idioms.

`,config ,load` *filespec* ...

The `,config` command can be used to execute a `,load` command that interprets configuration language forms from the specified file or files in the current configuration environment.

`,in` *struct-name form*

This form of the `,in` command evaluates a single form in the specified environment without moving the command processor into that environment.

`,load-into` *struct-name* *file* ...

The `,load-into` command loads one or more files into a specified environment. It is similar to

```
,in struct-name
,load file ...
```

except that the command processor's current environment is unchanged afterwards.

`,new-package` *name* *open* ...

The `,new-package` command creates a new environment. It abbreviates the sequence

```
> ,in config
config> (define-structure name (export)
          (open open ... scheme))
config> ,in name
name>
```

E.2.3 Configuration Environments

It is possible to set up multiple configuration environments. Configuration environments open the `module-system` structure instead of the `scheme` structure, and in addition use structures that export structures and interfaces. The initial configuration environment opens the two structures `built-in-structures` and `more-structures`. The `built-in-structures` structure exports most of the basic structures, including `scheme`, `table`, `record`, and `compiler`. `more-structures` exports structures involved in the programming environment and optional features such as `sort`, `random`, and `threads`.

For example:

```
> ,new-package foo
foo> ,open module-system built-in-structures
foo> (define-structures ((x (export a b)))
       (open scheme)
       (files x))
foo>
```

`,config-package-is` *struct-name*

The `,config-package-is` command designates a new configuration environment for use by the `,config` command and resolution of *struct-name*s for other commands.

Appendix F

Object-Oriented Programming in Scheme

This material is excerpted from "Object-oriented programming in Scheme" by Norman Adams and Jonathan Rees, published in Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, *pages 277–288. Reproduced with permission of ACM.*

Scheme was originally inspired by Actors, Hewitt's message-passing model of computation [62, 2]. Steele has described the relationship between Scheme and Actors at length [58]. We take advantage of this relationship—and we try not to duplicate functionality that Scheme already provides—to add full support for object-oriented programming. Our extensions are in keeping with the spirit of Scheme: "It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions." [11]

We develop examples of how one might program in Scheme using object-oriented style. Inherited behavior is handled in a straightforward manner. We show that a new disjoint data type for instances must be added to Scheme in order to permit application of generic operations to all Scheme objects, and that making generic operations anonymous supports modularity.

We use the terms *object* and *instance* idiosyncratically: an object is any Scheme value; an instance is a value returned by a constructor in the object system. An *operation*, sometimes called a "generic function," can be thought of as a procedure whose definition is distributed among the various objects that it operates on. The distributed pieces of the definition are called "methods."

F.1 Objects Implemented as Procedures

F.1.1 Simple Objects

Our first approximation to object-oriented programming in Scheme is straightforward: an instance is represented as a procedure that maps operations to methods. A method is represented as a procedure that takes the arguments to the operation and performs the operation on the instance.

As an example, consider the following definition of a constructor for cells:

```
(define (make-simple-cell value)
  (lambda (selector)
    (cond ((eq? selector 'fetch)
           (lambda () value))
          ((eq? selector 'store!)
           (lambda (new-value)
             (set! value new-value)))
          ((eq? selector 'cell?)
           (lambda () #t))
          (else not-handled))))

(define a-cell (make-simple-cell 13))
((a-cell 'fetch))        →  13
((a-cell 'store!) 21)    →  unspecified
((a-cell 'cell?))        →  true
((a-cell 'foo))          →  error
```

Each call to `make-simple-cell` returns a new cell. Abstract operations are represented by symbols, here `fetch`, `store`, and `cell?`. An instance is represented by the procedure returned by the constructor. The lexical variables referenced by the methods serve the purpose of instance variables. To perform an operation on an instance, the instance is called, passing the operation as the argument; the resulting method is then applied to the arguments of the operation. There is no explicit notion of class, but the code portion of the closure constructed for the expression

```
(lambda (selector) ...)
```

serves a similar purpose: it is static information shared by all instances returned by the `make-simple-cell` constructor.

An instance returns the object `not-handled` instead of a method to indicate it defines no behavior for the operation. The particular value of `not-handled` is immaterial, as long as it is identifiable as being something other than a method. We will make use of this property later.

```
(define not-handled (list 'not-handled))
```

To improve the readability of operation invocation, we introduce the procedure `operate`:

```
(define (operate selector the-instance . args)
  (apply (the-instance selector) args))
```

This lets us replace `((a-cell 'store!) 34)` with the somewhat more perspicuous

```
(operate 'store! a-cell 34).
```

`Operate` is analogous to `send` in old Flavors [67] and `=>` in CommonObjects [56].

F.1.2 Inheritance

The following defines a "named" cell that inherits behavior from a simple cell.

```
(define make-named-cell
  (lambda (value the-name)
    (let ((s-cell (make-simple-cell value)))
      (lambda (selector)
        (cond ((eq? selector 'name)
               (lambda () the-name))
              (else (s-cell selector)))))))
```

We say that objects returned by `make-named-cell` have two *components*: the expression `(make-simple-cell ...)` yields the first component, and the expression `(lambda (selector) ...)` yields the second. The programmer controls what state is shared by choosing the arguments passed to the constructors, and by choosing the expressions used to create the components. In this style of inheritance, only behavior is inherited. An instance can name its components, but can assume nothing about how they are implemented.

We have single inheritance if the instance consults one component (not including itself) for behavior. We have multiple inheritance if the instance consults multiple components (not including itself) for behavior. For the above example, that might be done by expanding the `else` clause as follows:

```
    ⋮
(else (let ((method (s-cell selector)))
        (cond ((eq? method not-handled)
               (another-component selector))
              (else method))))
```

This is similar to the style of inheritance in CommonObjects [56] in that instance variables are considered private to the component, and cannot be inherited. However, CommonObjects forces the components to be new, where our formulation has no such requirement; thus distinct instances may share components and, in turn, the components' state. Because classes are not explicit, and it is possible to share state and behavior, one may say this approach provides delegation rather than inheritance [39, 60]. When more than one component defines behavior for an operation, the behavior from the more specific component (the one consulted first) shadows the less specific.

F.1.3 Operations on Self

One frequently wants to write a method that performs further operations on the instance. For example, if we were implementing an open output file as an instance that supported `write-char`, we could implement a `write-line` operation as a loop performing `write-char` operations on the instance itself. In simple cases, a method can refer to the instance by using `letrec` in the instance's implementation:

```
(letrec ((self
          (lambda (selector)
            (cond ((eq? selector 'write-line)
                   (lambda (self string)
                     ...
                     (operate 'write-char self char)))
                  ...))))
  self)
```

When inheritance is involved this will not work. The variable `self` will not be in scope in the code for inherited methods. A component that uses `letrec` as above will be able to name its "component self", but not the composite. A small change to the protocol for invoking methods remedies this: the composite is passed as an argument to every method.

```
(define (operate selector the-instance . args)
  (apply (the-instance selector) the-instance args))
```

F.1.4 Operations on Components

Because components may be given names, a composite's method for a given operation can be defined in terms of the behavior of one of its components. For example, we can define a kind of cell that ignores stores of values that do not satisfy a given predicate:

```
(define make-filtered-cell
  (lambda (value filter)
    (let ((s-cell (make-simple-cell value)))
      (lambda (selector)
        (cond ((eq? selector 'store!)
               (lambda (self new-value)
                 (if (filter new-value)
                     (operate 'store! s-cell new-value))))
              (else (s-cell selector)))))))
```

This practice is analogous to "sending to `super`" in Smalltalk and `call-next-method` in the Common Lisp Object System (CLOS) [18].

There is a problem here. When the composite performs an operation on a component, the "self" argument to the component's method will be the component, not the composite. While this is sometimes useful, we also need some way to perform an operation on a component, passing the composite as the "self" argument. (This is the customary operation of send to super and its analogues.) We can do this using a variant on `operate` that takes as arguments both the composite and the component from which a method is to be obtained:

```
(define (operate-as component selector composite . args)
  (apply (component selector) composite args))

(define (operate selector instance . args)
  (apply operate-as instance selector instance args))
```

F.2 Integration With the Rest of the Language

F.2.1 Operations on Non-instances

One often wants to include non-instances in the domain of an abstract operation. For example, the operation `print`, when performed on a non-instance, should invoke a printer appropriate to that object's type. Similarly, the abstract operation `cell?` should apply to any object in the Scheme system, returning false if the object is not a cell. But given our present definition of `operate`, we cannot meaningfully say `(operate` *op* x`)` unless x is an instance.

We can make `print` work on non-instances if we can associate methods with every non-instance on which we expect to perform operations. This is accomplished with a simple change to `operate-as` (remember that `operate` is defined in terms of `operate-as`):

```
(define (operate-as component selector the-object . args)
  (apply (get-method component selector)
         the-object args))

(define (get-method component selector)
  (if (instance? component)
      (component selector)
      (get-non-instance-method component selector)))
```

We will consider the definition of `instance?` below. As for `get-non-instance-method`, we would like it to behave in a manner consistent with the dispatch routines we have set up for instances, which are specific to the particular kind of object being operated on.

```
(define (get-non-instance-method object selector)
  ((cond ((pair? object)
          get-method-for-pair)
         ((symbol? object)
          get-method-for-symbol)
         ... other types ...)
   object selector))

(define (get-method-for-pair pair selector)
  (cond ((eq? selector print)
         (lambda (self port) ...))
        ... other methods ...))
```

```
(define (get-method-for-symbol symbol selector)
  ...)
```

Unfortunately, any time we want a new operation to work on some kind of non-instance, we have to modify the appropriate non-instance dispatch routine to handle the new operation. This presents problems because these dispatch routines are global resources for which there may be contention. What if two different modules chose to define incompatible behavior for a shared operation? But even worse, in the case of the `cell?` operation, we would have to modify *all* dispatch code, even that for non-instances. That would hardly be modular.

One way to define a default behavior for `cell?` would be to have a "vanilla object," analogous to "class `object`" in many object systems, that acted as a component of every object, instance or non-instance. We could change the dispatch for that component every time we wanted to add a new operation to the system. This approach would also suffer the contention problem mentioned above.

Instead of using such a "vanilla object," we take the approach of associating default behavior with each operation. The default behavior is used when there is no method for the operation in the object being operated on. We change the definition of `operate-as` to implement this:

```
(define (operate-as component selector composite . args)
  (let ((method (get-method component selector)))
    (if (eq? method not-handled)
        (apply (default-behavior selector) composite args)
        (apply method composite args))))
```

The procedures `default-behavior` and `set-default-behavior!` maintain a table mapping selectors to procedures that implement the default behavior.

```
(set-default-behavior! 'cell? (lambda () #f))
(define a-cell (make-simple-cell 5))
(operate 'cell? a-cell)  →  true
(operate 'cell? 1)       →  false
```

Later we will see how to associate default behavior with operations without using side effects.

Returning to the predicate `instance?`: Our new version of `operate-as` indirectly uses `instance?` to determine whether the object is an instance. Defining `instance?` to be the same as `procedure?` almost works, but it fails to account for the distinction between those procedures that obey the protocols of the object system and those that don't: `(operate 'cell? list)` would generate an error instead of returning false.

To implement `instance?`, we need to be able to mark some procedures as being instances. But in Scheme, the only operations on procedures are application and identity comparison (`eq?`). So we must add a new primitive type to Scheme, disjoint from the types already provided, that satisfies the following rules:

```
(instance-ref (make-instance x)) → x
(instance? (make-instance x)) → true
(instance? y) → false, if y was not a value
                 returned by make-instance
```

Instance constructors such as `make-cell` must now mark instances using `make-instance`, and `get-method` must coerce instances to procedures using `instance-ref`.

```
(define (get-method component selector)
  (if (instance? component)
      ((instance-ref component) selector)
      (get-non-instance-method component selector)))

(define (make-simple-cell value)
  (make-instance (lambda (selector)
                   ...)))
```

F.2.2 Non-operations Applied to Instances

In purely object-oriented languages, all operations are generic. In our framework this would mean that a procedure call such as `(car x)` would be interpreted the same way as, say, `(operate 'car x)`. Thus the domain of `car` and other primitives would be extended to include all instances that choose to handle the corresponding generic operation.

We consider this kind of extension to be undesirable for a number of reasons, all of which stem from the fact that programs become harder to reason about when all operations are generic. For example, if a call to `car` can invoke an arbitrary method, then a compiler cannot assume that the call will not have side effects. If arithmetic can be extended to work on arbitrary objects, then algebraic properties such as associativity no longer hold. Not only is efficiency impaired, but programs can become harder for humans to understand as well.

In the case of some system procedures, however, extension to instances can be very useful for the sake of modularity. For example, procedures on I/O ports can probably be extended without significant impact on performance, and they may already be generic internally to the system. If such procedures are extended, the contract of the corresponding abstract operation must be made clear to those programmers who define methods for them.

F.2.3 Instances As procedures; Procedures as Instances

One particular Scheme primitive can be usefully extended to instances without sacrificing efficiency or static properties: procedure call. This is because procedure call has no algebraic or other properties that are in danger of being destroyed; invoking a procedure can potentially do anything at all. So we can arrange for the Scheme implementation to treat a combination

```
( proc  arg ...)
```

the same as

```
(operate 'call  proc  arg ...)
```

whenever the object `proc` turns out to be an instance. The `call` operation should also be invoked whenever an instance is called indirectly via `apply` or any other system procedure that takes a procedure argument.

If in addition we define the `call` operation so that when applied to a procedure, it performs an ordinary procedure call, then any lambda-expression

```
(lambda ( var ...)  body)
```

is indistinguishable from the instance construction (assuming alpha-conversion to avoid name conflicts)

```
(make-instance
 (lambda (selector)
   (cond ((eq? selector 'call)
          (lambda (self  var ...)  body))
         (else not-handled)))).
```

This means that if we take instance construction to be primitive in our language, then `lambda` need not be.

Allowing the creation of instances that handle procedure call is equivalent to allowing the creation of procedures that handle generic operations. Procedures that handle operations can be very useful in a language like Scheme in which procedures have high currency. In T [46], for example, the operation `setter`, when applied to a procedure that accesses a location, returns by convention a procedure that will store into that location:

```
((operate 'setter car) pair new-value)
```

is equivalent to

```
(set-car! pair new-value).
```

`Car` and other built-in access procedures are set up as if defined by

```
(define car
  (make-instance
   (lambda (selector)
     (cond ((eq? selector 'call)
            (lambda (self pair)
              (primitive-car pair)))
           ((eq? selector 'setter)
            (lambda (self) set-car!))
           (else not-handled)))))
```

T has a generalized `set!` special form, analogous to Common Lisp's `setf`, that provides the more familiar syntax

```
(set! (car pair) obj).
```

To define a new procedure that works with `set!`, it is sufficient to create an instance that handles the `call` and `setter` operations appropriately.

The T system itself makes heavy use of operations on procedures. The system printer and the trace facility provide two examples: Often, a procedure has a print method that displays values of closed-over variables so that the procedure is easily identifiable during debugging. The `trace` utility creates a "traced" version of a given procedure; the original procedure is recovered by invoking an operation on the traced procedure.

F.3 Anonymous Operations

In our development so far, as in Smalltalk[20] and CommonObjects[56], symbols are used to represent abstract operations. This practice can lead to name clashes. If two modules use the same symbol to represent two different abstract operations, then one module may not work in the presence of the other. For example, the two modules may require different default behaviors for two different abstract operations for which they have coincidentally chosen the same name. Or, if module A defines objects that include as components objects created by module B (i.e. "subclasses" one of B's classes), then A might inadvertently use the name of an operation intended to be internal to B. When a method in B performs this operation on instances created by A, the method in A will be seen instead of the intended method in B.

The problem can be eliminated if operations are represented by unique tokens instead of symbols. New tokens must be obtained by calling a system procedure that generates them. The creator of the token can use lexical scoping to control what other code in the system may perform the corresponding abstract operation. (In contrast to Common Lisp, Scheme uses lexical scoping to control visibility of names between modules.) We call this feature *anonymous operations* because, although the token representing an operation is usually the value of some variable, the name of the variable is immaterial.

Here is the cell example reformulated using anonymous operations. New operations are created with `make-operation`, which takes as an argument the operation's default behavior.

```
(define fetch (make-operation error))
(define store! (make-operation error))
(define cell?
  (make-operation (lambda () #f)))
```

```
(define make-simple-cell
  (lambda (value)
    (make-instance
     (lambda (selector)
       (cond ((eq? selector fetch)
              (lambda (self) value))
             ((eq? selector store!)
              (lambda (self new-value)
                (set! value new-value)))
             ((eq? selector cell?)
              (lambda (self) #t))
             (else not-handled))))))

(define a-cell (make-simple-cell 8))
(operate fetch a-cell)  →  8
```

The variables `fetch`, `store!`, and `cell?` are evaluated in the course of dispatching, and the selector argument in calls to `operate` is an operation, not a symbol.

How are we to represent operations? The only property we need is that they be distinct from each other, as determined by `eq?`. We could use symbols generated by `gensym`, or pairs, but a better choice is to represent operations as procedures. This way we can arrange that when an operation is called, it performs itself on its first argument.[1]

```
(define (make-operation default-behavior)
  (letrec ((this-op
            (lambda (the-object . args)
              (apply operate this-op the-object args))))
    (set-default-behavior! this-op default-behavior)
    this-op))
```

We can now convert from the "message passing" style interface that we have been using to a generic procedure style interface. That is, rather than writing

```
(operate fetch a-cell)
```

we write

```
(fetch a-cell).
```

The advantages of the generic procedure style are described in [33].

If we are allowed to define `call` methods on instances (as described in section F.2.3), then we can eliminate the global table maintained by `set-default-behavior!` and access an operation's default method via an operation applied to the operation itself.

[1]The definition of `make-operation` given here requires every evaluation of the inner `lambda`-expression to result in a distinct closure. However, some Scheme compilers may try to "optimize" the storage allocation performed by `make-operation` by reusing the same procedure object every time. This is not a serious difficulty, however, since it is always possible to circumvent such optimizations.

```
(define (make-operation a-default-behavior)
  (letrec ((this-op
            (make-instance
             (lambda (selector)
               (cond ((eq? selector call)
                      (lambda (self the-object . args)
                        (apply operate this-op the-object args)))
                     ((eq? selector default-behavior)
                      a-default-behavior))))))
    this-op))

(define default-behavior (make-operation error))
```

F.4 An Example

The example program given here illustrates multiple inheritance, sending to self, and sending to a component.

To improve readability and hide the details of the object system implementation, some syntactic sugar is introduced for expressions that create instances and dispatchers. An expression of the form

```
(dispatcher (disp1 disp2 ...)
    op1 op2 ...)
```

yields a dispatcher for instances that handle the operations op_i directly and have components with dispatchers $disp_1$ $disp_2$ .... An expression of the form

```
(instance disp
    (component1 component2 ...)
  ((op1 formal11 formal12 ...) body1)
  ((op2 formal21 formal22 ...) body2)
  ...)
```

yields an instance with components $component_1$, $component_2$, ..., and methods

```
(lambda (formal1i ...) bodyi)
```

for the corresponding operations. The operations, and their order, in the instance and its components must match those of the given dispatcher.

The example defines a kind of cell that maintains a history of all the values that have been stored into it. These cells are implemented as objects that have two components: a "filtered cell" (section F.1.4), and a sequence. The cell's history may be accessed using sequence operations.

```scheme
; ------------------ Sequences

(define seq-ref (make-operation #f))
(define seq-set! (make-operation #f))
(define seq-length (make-operation #f))

(define sequence-dispatcher
  (dispatcher () seq-ref seq-set! seq-length print))

(define (make-simple-sequence size)
  (let ((v (make-vector size)))
    (instance sequence-dispatcher
        ()
      ((seq-ref self n)
       (vector-ref v n))
      ((seq-set! self n val)
       (vector-set! v n val))
      ((seq-length self)
       (vector-length v))
      ((print self port)
       (format port "#<Sequence ~s>" (seq-length self))))))
```

A sequence is an object that behaves just like a vector, but is manipulated using generic operations.

```scheme
; ------------------ Filtered cells

(define fetch (make-operation #f))
(define store! (make-operation #f))

(define cell-dispatcher
  (dispatcher () fetch store! print))

(define (make-filtered-cell value filter)
  (instance cell-dispatcher
      ()
    ((fetch self) value)
    ((store! self new-value)
     (if (filter new-value)
         (set! value new-value)
         (discard self new-value)))
    ((print self port)
     (format port "#<Cell ~s>" value))))
```

```
; ------------------ Cells with history

(define position (make-operation #f))
(define discarded-value (make-operation #f))

(define discard
  (make-operation
   (lambda (cell value)
     (format t "Discarding ~s~%" value))))

(define cell-with-history-dispatcher
  (dispatcher (cell-dispatcher
               sequence-dispatcher)
    position
    store!
    discard
    discarded-value
    print))

; ------------------ Cells with history, continued

(define (make-cell-with-history initial-value filter size)
  (let ((cell (make-filtered-cell initial-value
                                  filter))
        (seq (make-simple-sequence size))
        (most-recent-discard #f)
        (pos 0))
    (let ((self
           (instance cell-with-history-dispatcher
               (cell seq)
             ((position self) pos)
             ((store! self new-value)
              (operate-as cell store! self new-value)
              (seq-set! self pos new-value)
              (set! pos (+ pos 1)))
             ((discard self value)
              (set! most-recent-discard value))
             ((discarded-value self)
              most-recent-discard)
             ((print self port)
              (format port
                      "#<Cell-with-history ~s>"
                      (fetch self))))))
      (store! self initial-value)
      self)))
```

The two methods for the `store!` operation, together with the somewhat frivolous `discard` operation, illustrate sending to self and sending to a component. The `store!`

method for cells with history overrides the `store!` method from the cell component. It invokes the `store!` operation on the component using `operate-as`. The component's method in turn calls a `discard` operation on the composite if the stored value fails to pass the filter. If the first `store!` method had simply invoked the `store!` operation on the component, the `discard` would have been applied to the component, not the composite, and the wrong method would have been invoked.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman.
Lisp: A language for stratified design.
BYTE, February 1988, pages 207–218.

[2] Gul A. Agha.
Actors: A Model of Concurrent Computation in Distributed Systems.
MIT Press, Cambridge, MA, 1986.

[3] David B. Albert.
Issues in MUSE security.
Manuscript (?), 1994.

[4] Alan Bawden.
Linear Graph Reduction: Confronting the Cost of Naming.
PhD thesis, MIT, 1992.

[5] Alan Bawden and Jonathan Rees.
Syntactic closures.
1988 ACM Conference on Lisp and Functional Programming, pages 86–95.

[6] Nathaniel Borenstein and Marshall T. Rose.
MIME extensions for mail-enabled applications: application/Safe-TCL and multipart/enabled-mail.
Working draft, 1993.
Internet: `ftp://ftp.fv.com/pub/code/other/safe-tcl.tar`.

[7] Rodney A. Brooks.
A robust layered control system for a mobile robot.
IEEE Journal of Robotics and Automation RA-2 (1):14–23, 1986.

[8] P. J. Brown.
Macro Processors and Techniques for Portable Software.
Wiley, 1974.

[9] William Clinger.
The Scheme 311 compiler: An exercise in denotational semantics.
Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, pages 356–364.

[10] William Clinger and Jonathan Rees.
Macros that work.

Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, pages 155–162, January 1991.

[11] William Clinger and Jonathan Rees (editors).
Revised^4 report on the algorithmic language Scheme.
LISP Pointers IV(3):1–55, July–September 1991.

[12] Pavel Curtis.
LambdaMOO programmer's manual for LambdaMOO version 1.7.6.
Xerox Corp., `ftp://parcftp.xerox.com/pub/MOO/ProgrammersManual.txt`, 1993.

[13] Pavel Curtis and James Rauen.
A module system for Scheme.
Proceeding of the 1990 ACM Conference on Lisp and Functional Programming, pages 13–19.

[14] Mike Dixon.
Embedded Computation and the Semantics of Programs.
PhD thesis, Stanford University, 1991.

[15] Bruce Donald and Jim Jennings.
Constructive recognizability for task-directed robot programming.
Journal of Robotics and Autonomous Systems 9(1):41–74, Elsevier/North-Holland, 1992.

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

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

[18] Richard P. Gabriel, et al.
Common lisp object system specification.
ANSI X3J13 Document 87-002, 1987.

[19] Erann Gat.
ALFA: A language for programming reactive robotics control systems.
Proceedings of the 1991 IEEE International Conference on Robotics and Automation, pages 1116–1121.

[20] Adele Goldberg and David Robson.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, 1983.

[21] Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand.
VLISP: A verified implementation of Scheme.
Lisp and Symbolic Computation 8(1/2):5–32, 1995.

[22] Robert H. Halstead.
Multilisp: a language for concurrent symbolic computation.
ACM Transactions on Programming Languages and Systems 7(4):501–538, October 1985.

[23] Chris Hanson.
Efficient stack allocation for tail-recursive languages.
Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, pages 106–118, Nice, France, June 1990.

[24] Chris Hanson.
A syntactic closures macro facility.
Lisp Pointers IV(4): 9–16, October–December 1991.

[25] Samuel P. Harbison and Guy L. Steele Jr.
C: A Reference Manual.
Prentice-Hall, second edition 1987.

[26] Christopher P. Haynes and Daniel P. Friedman.
Engines build process abstractions.
Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, pages 18–24.

[27] IEEE Std 1178-1990.
IEEE Standard for the Scheme Programming Language.
Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[28] Jim Jennings.
Modular software and hardware for robot construction.
Unpublished manuscript, Cornell Computer Science Robotics and Vision Laboratory.

[29] Leslie Pack Kaelbling and Stanley J. Rosenstein.
Action and planning in embedded agents.
In *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back,* MIT Press, 1990.

[30] Richard Kelsey.
Tail-Recursive Stack Disciplines for an Interpreter.
Technical Report NU-CCS-93-03, Northeastern University College of Computer Science, Boston, MA, 1992.

[31] Richard Kelsey and Paul Hudak.
Realistic compilation by program transformation.

In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages,* pages 281–292, 1989.

[32] Richard Kelsey and Jonathan Rees.
A tractable Scheme implementation.
Lisp and Symbolic Computation 7(4):315–335, 1995 (to appear).

[33] James Kempf, et al.
Experience with CommonLoops.
SIGPLAN Notices 22(12), 1987.

[34] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba.
Hygienic macro expansion.
1986 ACM Conference on Lisp and Functional Programming, pages 151–159.

[35] Eugene E. Kohlbecker Jr.
Syntactic extensions in the programming language Lisp.
Technical report no. 199, Indiana University Computer Science Department, 1986.

[36] Butler W. Lampson.
A note on the confinement problem.
CACM 16(10):613–615, 1973.

[37] Peter Lee.
Topics in Advanced Language Implementation.
MIT Press, Cambridge, MA (1991).

[38] Henry M. Levy.
Capability-based Computer Systems.
Bedford, MA: Digital Press, 1984.

[39] Henry Lieberman.
Using prototypical objects to implement shared behavior in object-oriented systems.
SIGPLAN Notices 21(11), 1986.

[40] Henry Lieberman and Carl Hewitt.
A real-time garbage collector based on the lifetimes of objects.
Communications of the ACM 26(6): 419–429, 1983.

[41] David MacQueen.
Modules for Standard ML.
ACM Conference on Lisp and Functional Programming, 1984.

[42] Robin Milner, Mads Tofte, and Robert Harper.
The Definition of Standard ML.
MIT Press, 1990.

[43] David Moon.
Genera retrospective.
International Workshop on Object-Oriented Operating Systems, 1991.

[44] James H. Morris Jr.
Protection in Programming Languages.
CACM 16(1):15–21, 1973.

[45] Jonathan A. Rees.
The June 1992 Meeting.
Lisp Pointers V(4):40–45, October–December 1992.

[46] Jonathan A. Rees and Norman I. Adams.
T: A dialect of Lisp or, LAMBDA: the ultimate software tool.
In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming,* pages 114–122.

[47] Jonathan A. Rees and Norman I. Adams.
The T manual, fourth edition.
Yale University Computer Science Department, 1984.

[48] Jonathan Rees and Bruce Donald.
Program mobile robots in Scheme.
Proceedings of the 1992 IEEE International Conference on Robotics and Automation, pages 2681–2688.

[49] John H. Reppy.
CML: a higher-order concurrent language.
In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation,* pages 293–305, 1991.

[50] R. L. Rivest, A. Shamir, and L. Adleman.
A method for obtaining digital signatures and public-key cryptosystems.
Communication of the ACM 21:120–126, Feb. 1978.

[51] Jerome H. Saltzer and M. D. Schroeder.
The protection of information in computer systems.
Proceedings of the IEEE 63(9):1278–1308, 1975.

[52] Michael D. Schroeder.
Cooperation of Mutually Suspicious Subsystems in a Computer Utility.
Ph.D. thesis, MIT Project MAC TR-104, 1972.

[53] Mark A. Sheldon and David K. Gifford.
Static dependent types for first-class modules.
ACM Conference on Lisp and Functional Programming, pages 20–29, 1990.

[54] Olin G. Shivers.
A Scheme shell.
To appear in *Journal of Lisp and Symbolic Computation.*
Also available as Technical Report TR-635, Laboratory for Computer Science, MIT, 1994.

[55] Brian C. Smith.
Reflection and Semantics in LISP.
In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages,* pages 23–35, Salt Lake City, January 1986.

[56] Alan Snyder.
CommonObjects: an overview.
SIGPLAN Notices 21(10), 1986.

[57] Guy L. Steele Jr.
Common Lisp: The Language.
Digital Press, Burlington MA, second edition 1990.

[58] Guy Lewis Steele Jr.
Lambda: the ultimate declarative.
MIT AI Memo 379, 1976.

[59] Guy Lewis Steele Jr. and Gerald Jay Sussman.
Lambda: the ultimate imperative.
MIT AI Memo 353, 1976.

[60] Lynn Andrea Stein.
Delegation is inheritance.
SIGPLAN Notices 22(12), 1987.

[61] Robert Strandh.
Oooz, a multi-user programming environment based on Scheme.
BIGRE Bulletin 65, IRISA, Campus de Beaulieu, F-35042, Rennex Cédex, France, July 1989.

[62] Gerald Jay Sussman and Guy Lewis Steele Jr.
Scheme: an interpreter for extended lambda calculus.
MIT AI Memo 349, 1975.

[63] Swinehart et al.
Cedar.
TOPLAS 4(8), 1986.

[64] D. Tarditi, A. Acharya, and P. Lee.
No assembly required: Compiling standard ML to C.
Technical Report, School of Computer Science, Carnegie Mellon University, 1991.

[65] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham.
Efficient software-based fault isolation.
Computer Science Division, University of California, Berkeley.
Year unknown; probably 1993 or 1994.

[66] Mitchell Wand.
Continuation-based multiprocessing.
In *Conference Record of the 1980 Lisp Conference*, pages 19–28.
The Lisp Conference, P.O. Box 487, Redwood Estates CA, 1980.
Proceedings reprinted by ACM.

[67] Daniel Weinreb and David Moon.
Lisp Machine Manual.
MIT AI Lab, 1981.

[68] Paul R. Wilson and Thomas G. Moher.
Design of the opportunistic garbage collector.
In *Proceedings of the SIGPLAN 1989 Conference on Object-Oriented Programming: Systems, Languages, and Applications.*

[69] Paul R. Wilson, Micheal S. Lam, and Thomas G. Moher.
Effective "static-graph" reorganization to improve locality in garbage collected systems.
In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation.*

[70] William A. Wulf, Roy Levin, and Samuel P. Harbison.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.

Biographical Note

Jonathan Rees was born in Detroit, Michigan, and lived in Ann Arbor until age 17. His childhood interests included amphibians, both fossil and living, and space travel. He attended Thomas Jefferson College, the University of Michigan, and Yale College, graduating from Yale in 1981. Following this he was a staff programmer at Yale. He enrolled at MIT in 1984, earning an S.M. in 1989 and a Ph.D. in 1995. He had summer internships at DEC's Western Systems Laboratory in 1984 and at Xerox PARC in 1990. In 1991-93 he took a sabbatical at Cornell to study biology and to work at the Computer Science Robotics and Vision Laboratory.

He has published eight papers on topics relating to programming language design and implementation. A growing interest in biology, coupled with frustration at the vagaries of engineering research, has led him to a change in research direction. His new quest is to understand how the behavior of "lower" organisms such as arthropods is programmed. His next position will be as Research Fellow at the University of Sussex Centre for Neuroscience, where he will study visual memory and behavior in Hymenoptera.

His leisure activities include playing the piano and looking under decaying logs for interesting animals, especially amphibians and insects.