A Module System for Scheme

Pavel Curtis
Xerox Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304
Pavel@Xerox.Com

James Rauen
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139
Rauen@LCS.MIT.Edu

Abstract

This paper presents a module system designed for large-scale programming in Scheme. The module system separates specifications of objects from their implementations, permitting the separate development, compilation, and testing of modules. The module system also includes a robust macro facility.

We discuss our design goals, the design of the module system, implementation issues, and our future plans.

1 Introduction

The simplicity, elegance, and expressive power of the Scheme language make it attractive for developing software, but it lacks facilities for structuring large programs. In particular, there is no convenient way to decompose programs into subsystems that can be developed and reused independently. We need mechanisms for controlling the sharing of names, for specifying the interfaces between subsystems, and for mechanically checking that those interfaces are respected.

Additionally, we would like to share syntactic extensions (macros) among subsystems without compromising modularity. Syntactic extension is a valuable abstraction mechanism that has long been a part of most Lisp-like languages, but has not yet been given a comprehensive semantics. In particular, most languages leave unspecified the environment in which transformation functions are evaluated, the persistence of their state across separate compilations, and the guarantees about how frequently and in what order they are invoked. These points must be addressed if macros are to be used in robust programming.

We discuss previous work on module systems, both for Scheme and for other languages, in the next section. Then, we list our goals for the design. This is followed by an informal description of the module system itself. Finally, we discuss issues that arise in any implementation of the system.

2 Previous work

Many languages contain facilities intended to support large-scale programming. The usual approach taken in Scheme implementations, such as T [12] and MIT Scheme [8], uses first-class environments. Although this approach is elegant, there is little hope for statically checking references between subsystems. In addition, the lack of explicit interfaces hampers the independent development of interacting subsystems.

In Common Lisp [14], a parse-time mechanism partitions identifiers into runtime structures called “packages”. Imperative procedure calls control the sharing of names among the packages; it is difficult or impossible to perform static checks. Further, no names are truly hidden, so abstractions cannot be enforced.

Felleisen and Friedman [3] describe a simple module system, but it lacks explicit interfaces and support for syntactic extensions. Rees [9] presents a solution to the modularity problems for syntactic extensions. We have adopted much of this approach to syntax, but the module system Rees describes also lacks explicit interfaces.

The module systems for other functional languages like ML [5] and FX [13] are more highly structured, including explicit interface-like “signatures” and static checking. However, this checking is tightly coupled with the type systems for these languages, so this approach is not easily adapted for use in a latently-typed language like Scheme.

Several languages in the Algol family, such as Ada [15], CLU [7], Mesa [4], and Modula-3 [2] were designed to support large-scale programming. They feature a static module structure with explicit, checked interfaces. Most do not provide more than one level of module scoping, with Ada and Mesa being notable exceptions. Mesa was the primary inspiration for our own design.

3 Design goals

When we began work on the module system, we had several goals in mind, some of which were inspired by our experience with the Mesa module system [4].

Separation: The module system should allow a strong separation between independent pieces of code. All dependencies between modules should be explicitly declared in the program text, making reference to textually disjoint specifications of the interfaces between subsystems.

Simplicity: The module system should consist of a small number of additional language forms that compose well. Scheme is frequently praised for its simplicity, small size, and elegant semantics; it behooves us to make our extensions to it in the same spirit.

P ∈ Program	::=	{E | D | I | M | MM}*
E ∈ Expression	::=	*usual Scheme expressions*
	|	VN
	|	(SN Q*)
D ∈ Definition	::=	(define Id E)
	|	(define-syntax Id E)
I ∈ Interface	::=	(interface IN S*)
	|	(interface IN IN)
S ∈ Specification	::=	(value Id A*)
	|	(syntax Id A*)
A ∈ Attribute	::=	(AN Q*)
M ∈ Module	::=	(module (X*) P)
X ∈ External	::=	(exports ES*)
	|	(imports IS*)
	|	(open OS*)
	|	without-scheme
ES ∈ ExportSpec	::=	IN | (IN { Id | (Id Id) }*)
IS ∈ ImportSpec	::=	IN | (IN Id*)
OS ∈ OpenSpec	::=	IN | (IN { Id | (Id Id) }*)
MM ∈ MetaModule	::=	(meta-module (X*) P)
IN ∈ InterfaceName	::=	Id
AN ∈ AttributeName	::=	Id
VN ∈ VariableName	::=	Id | (access IN Id) | IN#Id
SN ∈ SyntaxName	::=	Id | (access IN Id) | IN#Id
Id ∈ Identifier		
Q ∈ Datum		

Figure 1: Module system grammar.

Syntactic Extension: The module system should support the modular use of macros; they are a valuable abstraction mechanism. Syntactic extension, however, must have a complete, comprehensible, and simple semantics. In particular, the semantics must answer questions concerning the environment, store, and evaluation time of transformation functions. In addition, since those transformation functions may themselves be complex programs, the module system must be available at this "meta" level as well.

Static Checking: Unbound variables, unimplemented or multiply-implemented interfaces, and incorrect intermodule references should be reported *before* running the program. This static checking should be independent of any type analysis, since Scheme is not statically typed, but should not rule out future work we might do in that area.

Static Semantics: The semantics of the module system should not require concepts of state or mutation since modules concern the control of lexical naming, a fundamentally static notion. Scheme already contains well-understood constructs for lexical scoping, so it should be possible to explain the semantics of modules by a rewrite into module-free Scheme code. At the same time, we must recognize that large programs are developed and maintained incrementally. The semantics should not preclude separate compilation, read-eval-print loops, or dynamic loading. However, neither should the semantics make reference to files of code or other environmental commitments.

4 Overview of the system

The module system attacks the complexity of large software systems by providing tools to decompose programs into *modules*. Each module is a black box whose behavior is described by *interface specifications*.

```
(interface Promise
  (syntax delay)
    ; (delay expression) returns a promise to evaluate
    ; expression.
  (value force)
    ; (force promise) returns the value of promise's
    ; expression.
  )
```

Figure 2: An interface called Promise, containing two specifications.

Modules interact by sharing variable bindings and syntax descriptions, collectively referred to as *shared items* or just *items*. A module can *export* any item defined inside the module by associating the definition with some interface specification. The item is then available to other modules. A module can *import* an item by referring to the item's specification, thereby making the imported item available inside the module.

The code inside a module communicates with that outside it only by exporting and importing items via interface specifications. This restriction yields several software engineering benefits: the dependencies among the different modules in a program are all static and explicit, and (given the interface specifications) the modules can be developed, compiled, tested, and maintained independently.

Inside each module is an entire program, which may contain module structure of its own. The program implements the items that the module exports, either by providing top-level definitions of the items or by containing sub-modules that export the items.

5 Description of the module system

Figure 1 displays the grammar of module system programs.

A *program* is a sequence of expressions, definitions, modules, interfaces, and meta-modules. Module system programs are an upward-compatible extension of Scheme programs as defined by the Revised[3] Report [10].

Expressions include the usual kinds of Scheme expressions. Top-level expressions are usually evaluated for their side-effects, not for the values they compute.

Definitions introduce items (variable bindings and syntax descriptions) that can be used throughout the program where they are defined. If this program appears inside a module, then the module can export these items.

Interfaces contain specifications of items that modules can share.

Modules encapsulate subprograms. A module is an isolated scope. It can only interact with the rest of the program by sharing items specified in interfaces. The module's header explicitly identifies the items that the module provides (*exports*) and requires (*imports*).

Meta-modules encapsulate the implementation of complicated syntactic extensions.

5.1 Interfaces

An *interface* is a set of named specifications of items (variable bindings and syntax descriptions) that modules can

```
(module ((exports (Promise delay force)))
  (define-syntax delay
    (lambda (form use-env)
      (let ((expression (close (cadr form)
                               use-env)))
        `(make-promise (lambda ()
                         ,expression)))))
  (define make-promise
    (lambda (thunk)
      (let ((already-forced? #f) (result #f))
        (lambda ()
          (cond (already-forced? result)
                (else (set! result (thunk))
                      (set! already-forced? #t)
                      result))))))
  (define force
    (lambda (promise)
      (promise))))
```

Figure 3: A module implementing the items specified in the `Promise` interface.

share. The `interface` form associates a name with an interface; the scope of the name is the entire program in which the `interface` form appears. Note that we draw a distinction between interfaces and the names that refer to them; $\mathcal{I}$ refers to an interface, while IN refers to an identifier naming an interface. We use the notation $\mathcal{I}\#\mathrm{Id}$ to mean the specification of Id in the interface $\mathcal{I}$.

The specification of a variable binding may include *attributes* describing properties of the values that the variable may assume. The module system does not specify what these attributes are. We envision using attributes to specify such properties as types, effects, associativity, and commutativity.

The specification of a syntax description may also include attributes, describing in this case properties of the code that replaces uses of the new syntax. We anticipate using such attributes to describe both semantic properties, such as variable scoping, and syntactic ones, such as formatting rules for a pretty-printer.

Naturally, interfaces should also contain comments augmenting the specifications with informal descriptions of the items they specify.

Figure 2 shows a simple interface containing two specifications.

It is sometimes useful to be able to give another name to an interface, perhaps for brevity. The second syntax for `interface` forms,

`(interface` IN_1 IN_2`)`

associates IN_1 with the interface named by IN_2.

5.2 Modules

A *module* is an isolated scope that encapsulates a program. The scope is isolated in the sense that items defined outside the module are not visible inside the module, unless the module explicitly *imports* the items; likewise, items defined inside the module are not visible outside unless the module explicitly *exports* the items.

`(exports` IN`)`
  $\Rightarrow$ `(exports (`IN Id_1 ... Id_n`))`
    where Id_1, ..., Id_n are all of the identifiers specified in the interface named IN.
`(exports (`IN i_1 ... i_m`))`
  $\Rightarrow$ `(exports (`IN i_1`)` ... `(`IN i_m`))`
`(exports` ES_1 ... ES_k`)`
  $\Rightarrow$ `(exports` ES_1`)` ... `(exports` ES_k`)`

Figure 4: Notational abbreviations for `exports`, `imports`, and `open` clauses. Only the expansions for `exports` are shown; rewrites for `imports` and `open` follow the same pattern.

5.2.1 Exports

A module can *export* any item (variable binding or syntax description) defined at the top level of the program inside the module. Also, if a module contains a sub-module exporting some item, the module itself can also export the item. A module exports an item by associating the item's definition with a specification from some interface.

A module may export any number of items specified in any number of interfaces. Note that, in constrast to many other module facilities, interfaces need not be exported monolithically. This allows interfaces, the descriptions of abstractions, to be structured differently than modules, the implementations of those abstractions. Mesa programmers have often found this freedom convenient.

To export a variable binding, a module must contain either a definition `(define` Id_{mod} E`)` of the variable or a sub-module that exports the variable binding. The variable must implement some value specification $\mathcal{I}\#\mathrm{Id}_{ifc}$.

Likewise, to export a syntax description, a module must contain either a syntax definition `(define-syntax` Id_{mod} E`)`, or a sub-module that exports the syntax description. The syntax description must implement some syntax specification $\mathcal{I}\#\mathrm{Id}_{ifc}$.

In the case where the module contains a definition implementing an item, the syntax

`(exports (`IN `(`Id_{ifc} Id_{mod}`)))`

associates the implementation with the interface specification. If the identifiers Id_{ifc} and Id_{mod} are the same, then this may be written `(exports (`IN Id_{ifc}`))`. In the case where the module contains a sub-module exporting an item, the syntax `(exports (`IN Id_{ifc}`))` associates the implementation inside the sub-module with the specification.

Figure 3 shows a module that exports the items specified in the `Promise` interface of Figure 2.

Several convenient notational abbreviations are provided for use in `exports` clauses; they are summarized in Figure 4.

5.2.2 Imports

A module can *import* any item exported by another module, as long as either (1) the importing module occurs in the same program as a module exporting the item, or (2) the importing module is nested inside another module that itself imports the item.

To import an item, a module uses the clause

`(imports (`IN Id`))`

```
(module ((imports Promise))
  (define count 0)
  (define p
    (Promise#delay
      (begin (set! count (+ 1 count))
             count)))
  (Promise#force p)          ; Evaluates to 1
  (Promise#force p))         ; Evaluates to 1
```

Figure 5: A client module of the `Promise` interface.

This import clause gives the module access to the implementation of the item whose specification is $\mathcal{I}\#\mathrm{Id}$, where $\mathcal{I}$ is the interface named by IN.

If a module imports a variable binding, the code inside the module gains access to the current value of the binding. This code refers to the binding's current value with the expression `(access` IN Id`)`. Note that it is the binding itself that is imported, not simply some value of that binding; thus, assignments to shared bindings are visible to importers. For software engineering reasons, though, only the original exporter of a particular binding (for which it is a local variable) may assign to it.

If a module imports a syntax description, the forms inside the module may include uses of the imported syntax. The form `((access` IN Id`)` Q*`)` invokes the imported syntax.

We extend the lexical structure of Scheme slightly to provide the shorthand IN#Id for `(access` IN Id`)`. We call uses of this notation *qualified names*.

Figure 5 contains an example of a module that imports the items specified in the `Promise` interface of Figure 2.

5.2.3 Open

To avoid the verbosity of `access` forms and qualified names, a module can *open* any item it imports. By opening the item, the module introduces a new name for the item that is local to the module.

To open an item, a module includes the clause

`(open` (IN (Id_{ifc} Id_{mod})))

This allows the code inside the module to use the identifier Id_{mod} as an abbreviation for the form `(access` IN Id_{ifc}`)`. As with `exports` clauses, if Id_{ifc} and Id_{mod} are identical, the clause may be shortened to `(open` (IN Id_{ifc})`)`.

Users of the Mesa programming language, which includes a feature similar to this, have found that excessive opening of imported items significantly complicates the maintenance of programs. Judicious and sparing use of the feature, however, has proven to be an aid to readability. The precise definitions of "excessive" and "sparing" vary considerably between programmers.

Figure 6 shows a module that imports, opens, and uses the items specified in the `Promise` interface. This module is semantically identical to the module in Figure 5.

5.3 Meta-modules

A *meta-module* is a module that implements only syntax descriptions. We defer to Section 6 a complete discussion of the semantics of meta-modules.

```
(module ((imports Promise)
         (open Promise))
  (define count 0)
  (define p
    (delay
      (begin (set! count (+ 1 count))
             count)))
  (force p)                  ; Evaluates to 1
  (force p))                 ; Evaluates to 1
```

Figure 6: Another client module of the `Promise` interface, equivalent to the module in Figure 5.

5.4 The initial Scheme environment

Programs are interpreted in an environment that maps the names of built-in Scheme procedures and syntax, including the module system primitives, to their usual meanings. This environment is derived from a built-in interface called `Scheme`. The `Scheme` interface contains specifications of the aforementioned items.

Every module and meta-module implicitly imports and opens the `Scheme` interface. In most cases, this is a useful default, but in applications involving the use of an embedded language, for example, it could prove inconvenient. To override this default, therefore, modules and meta-modules may include the clause `without-scheme` in their headers.

6 Syntactic extension

Syntactic extensions (macros) are a powerful feature provided by many Scheme implementations. However, there is currently no accepted standard for macros; each Scheme implementation takes a different approach. We begin our discussion of syntactic extension by describing `define-syntax` and meta-modules, our two facilities for defining new syntax.

Two aspects of macro semantics are particularly important in the context of large-scale programming: avoiding conflicts among the identifiers inside and outside of macro expansions, and the meaning of a transformation procedure with side-effects. We also discuss each of these issues in what follows.

6.1 Defining new syntax

Most macros are defined by a fairly simple rewrite rule that can be implemented in a single Scheme procedure. For these applications, we provide the form `(define-syntax` Id E`)`, in which E evaluates to a procedure mapping uses of the new syntax into their equivalent expansions. Because E is, in effect, an extension to the Scheme parser, it must be possible to evaluate it before running the containing program itself. Thus, the variables of the containing program cannot be available to E. We therefore interpret E in the initial Scheme environment, rather than the lexically apparent one.

Meta-modules provide a more powerful way to define syntactic extensions than `define-syntax`. They enable programmers to use modular program structure in the implementation of transformation procedures. For example, one might use a meta-module to implement a partial evaluator, pattern compiler, or parser generator used in the transformation procedure for some new syntactic form. Also, meta-modules enable transformation procedures that share auxiliary procedures and data.

The `meta-module` form has the same syntax as the `module` form, but somewhat different semantics. Most significantly, the program inside a meta-module (called a *meta-program*) is interpreted in the initial Scheme environment, just as is the expression in a `define-syntax` form. Because of this, the `exports` and other header clauses behave a bit differently in meta-modules.

Within a meta-module, the exported syntax descriptions are treated as if they were variable bindings. In particular, to export a syntax description to the specification $\mathcal{I}\#\mathrm{Id}_{ifc}$, a meta-module must contain either a variable definition (`define` Id_{mod} E), where E evaluates to an appropriate syntax transformation procedure, or a sub-module that exports such a variable.

A meta-module's `imports`, `open`, and `without-scheme` clauses establish an environment in which to interpret the expansions of the macros it implements. This environment-specific interpretation is described below.

6.2 Syntactic environments

Syntax transformation procedures introduce, examine, and in other ways manipulate identifiers in the code on which they operate. Under certain circumstances (described by Bawden and Rees [1]), this can lead to inadvertent name conflicts.

Several techniques for "hygenic" syntax transformation have been proposed [1, 6], including one in the forthcoming Revised[4] Report [11]. We currently use a version of the syntactic closure mechanism [1] in order to control the scopes of identifiers manipulated by syntax transformations.

A *syntactic environment* is a static mapping from identifiers to their meanings. An identifier may be mapped to any one of the following:

- An interface.
- A variable. Variables are names that can be bound to values during program execution.
- A syntax description. The syntax description may be either a special token indicating a built-in syntax primitive (`lambda`, `module`, etc.), or a syntax transformation procedure.

Note that, even though there are three kinds of meanings for identifiers, we maintain Scheme's tradition of a single space of names; any of the three kinds of bindings may be shadowed lexically by any other kind.

A syntax transformation procedure receives two arguments: the instance of the syntax to be transformed, and the syntactic environment where this instance appears. The transformation procedure computes a new form to be interpreted in place of the original instance.

The transformation procedure may *close* any form in a particular syntactic environment, indicating that the form's free identifiers are to be interpreted in that syntactic environment. The `close` procedure performs this operation.

The form returned by a transformation procedure is automatically closed in the syntactic environment where the transformation procedure is defined.

6.3 Semantics of syntax transformation procedures

If syntax transformation procedures are purely functional, then the order in which they are applied does not matter. However, transformation procedures might very well have mutable state; there are some situations where this is desirable. For example, a complicated transformation may require an expensive table construction; the programmer may wish to cache this table so that it does not have to be recomputed every time the syntax is invoked.

Given that program modules may be compiled separately and perhaps even simultaneously, it hardly seems desirable (or even necessarily possible) to specify an order in which transformation procedures are invoked. In general, an interpreter or compiler should be free to apply the syntax transformations any number of times, in any order.

To accomodate both transformation procedure state and order-independence, we use a nondeterministic description of how syntax transformation procedures are applied.

When the interpreter or compiler encounters a form Q invoking syntax defined by (`define-syntax` Id E), something equivalent to the following happens. E is evaluated in the initial Scheme environment and store, producing a transformation procedure p. Then p is applied to an arbitrary number of arbitrary forms and syntactic environments. Next, p is applied to Q and Q's syntactic environment, producing a new form Q'. Finally, Q' is closed in the syntactic environment of the `define-syntax` form, and the result is used in place of Q.

Likewise, when the interpreter or compiler encounters a form Q invoking syntax whose transformation procedure is defined in a meta-module by (`define` Id E), something equivalent to the following happens. The meta-program inside the meta-module is evaluated in the initial Scheme environment and store. The transformation procedures exported by the meta-program are applied in an arbitrary order to an arbitrary number of forms and syntactic environments. Next, the transformation procedure for Id is applied to Q and Q's syntactic environment, producing a new form Q'. Finally, Q' is closed in the syntactic environment established by the meta-module's imports and open clauses, and the result is used in place of Q.

These semantics impose no restrictions on the order in which syntax is transformed. They also permit syntax transformation procedures to maintain state, with some guarantee that the state might be available the "next time" the syntax is invoked. However, it is neither meaningful to say nor possible to guarantee that the state will always be preserved between invocations of the syntax. For instance, it is not possible to write a `gensym`-like macro that expands into a different symbol every time it is invoked.

7 Implementation issues

The semantics of the module system and of Scheme in general give meaning only to whole programs, not fragments thereof. For large programs, though, it is usually undesirable to present the entire text to a compiler all at once. Practical languages must therefore accomodate separate compilation of program fragments.

An important issue in implementing separate compilation for any language is to remain true to the semantics even though the compiler has access to only an isolated fragment of a whole program. We address this issue in two parts:

mechanisms for allowing the compiler to work on a fragment, and mechanisms for putting the compiled fragments together.

7.1 Compiling fragments of programs

Suppose that the following program fragment is to be compiled in isolation:

```
(define (fact n)
   (if (< n 2) 1 (* n (fact (- n 1)))))
```

It could mean anything at all depending upon its syntactic environment. We cannot necessarily determine its meaning even if we are given that this is a fragment of the top-level program. For example, the rest of the program might contain a top-level redefinition of the syntactic keyword `define`.

Separate compilation thus requires that the user provide at least an approximation to the syntactic environment of the given program fragment. Of course, there are useful defaults to make this specification easier, such as assuming that the names of all built-in primitives or syntactic keywords have their usual meanings. The specification becomes more complicated when it involves names without bindings in the initial environment.

To address this problem, we introduce the notion of *exposing* pieces of the rest of the program to the compiler during the compilation of a particular fragment. This exposure is a pledge that the exposed fragments are "visible," in some sense, to the compiled fragment in the full program. The compiler can then make use of what has been exposed and put sufficient information in its output to allow a check at linking time that the pledge has been fulfilled.

We anticipate that, in practice, there will be three useful kinds of exposure:

- Exposing an interface definition; this allows the compiler to check, for every `(access` IN Id`)` form, that the given interface actually contains an item of that name, that names referring to syntax are only used in appropriate contexts (that is, the car of a form), that type declarations are obeyed, etc.
- Exposing the meta-program that implements a new piece of syntax; this allows the compiler to expand uses of the new syntax. Otherwise, the expansion process must be delayed until linking or run time. In practice, a compiler might simply refuse to process any code for which expansion cannot be performed; this seems reasonable given that the charter of the compiler is to make the program run faster.
- Exposing the modules exporting certain imported variables; this allows the compiler to perform several kinds of global optimization, such as procedure- and constant-integration.

Of course, if the programmer exposes more code to the compiler, the compiled fragment depends on more of the full program, and that fragment is more likely to require recompilation when the full program changes. The programmer has complete control over this accretion of dependencies, though, and can even vary the amount exposed for each program fragment. For example, one might expose the implementation of a particular interface during compilation of other code in the same subsystem, but not during compilation of clients of that subsystem.

Many separate compilation facilities work in much this way, even those for languages without modules, like C. But we know of none that use the notion of exposure to explain the process and none that employ that concept so generally.

7.2 Linking compiled fragments

Once the compiler accepts fragments of programs, we need a way to combine those fragments into larger ones and eventually into full programs. We envision a static "binding" tool for combining fragments before execution as well as the usual interactive `load` procedure.

The static binder effectively concatenates a set of program fragments into a larger one for ease of handling. Optionally, it may wrap the result in a given `(module ...)` or `(meta-module ...)` form. In this way, whole subsystems can be packaged together, hiding the various interfaces used for internal communication. We anticipate that most uses of the binder will include the wrapping feature and account for almost all occurrences of nested modules.

During the combining process, the binder checks for static errors that arise due to the combination. For example, final checks can often be made for multiple exports of a single binding, imports of unexported bindings, and, in the wrapping form of combination, undefined variables.

8 Future Plans

We have begun implementation of a compiler, loader, and runtime system for an enhanced version of Scheme including the module system described in this paper. The compiler produces machine-independent C code and the runtime system is built on top of the Portable Common Runtime system [16].

We are building (and will make freely available) a portable version of the module system as an add-on to any Scheme implementation. This will be a standard Scheme program that translates files of module system code into files of standard Scheme along with calls to certain runtime procedures to perform dynamic linking of separately-translated pieces. While we don't expect the performance of such translated code to match that produced by our compiler, it should be enough to allow widespread experimentation with the facility.

In parallel with this implementation work, we have begun a theoretical investigation of the issues involved in adding a useful static type inference system to Scheme. This work, if successful, will eventually interact with the module system at least to the degree of adding type information for the exported values in interfaces. It may also be the case that types will themselves become exportable entities with their own interface specifications, including some information-hiding capability. We do not expect the addition of types to have a significant effect on the existing semantics of the module system.

Another area we hope to investigate quite soon is the use of interfaces in support of multilingual programming. In our own environment, it would be very useful to be able to use Cedar, Modula-3, C, and Common Lisp programs and data from Scheme, and vice-versa. We would like to be able to do this, however, without requiring the client programmer to know that the code being used is written in another, very different language. One approach we've considered is a tool which accepts an interface description in one language and generates an "equivalent" interface in a second language. The new interface comes with an implementation that first performs the appropriate data translations and then invokes procedures in the original interface. Our initial investigations lead us to suspect that, in the majority of cases, we can generate the glue code and new interface entirely automatically.

9 Conclusions

We have presented the design of a simple but powerful module system for the programming language Scheme. It supports a high degree of separation between program subsystems with all communication via explicit interfaces. It also includes a modular, comprehensible, and practical mechanism for sharing syntactic extensions in the same manner as variable bindings. We believe that this module facility finally makes it possible for groups of programmers to write large, robust programs in Scheme.

Acknowledgements

We would like to thank Alan Demers, Gregor Kiczales, and John Lamping for their consultation on the early design of the system.

References

[1] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, ACM, July 1988.

[2] Luca Cardelli et al. *Modula-3 Report (revised)*. Technical Report 52, Digital Equipment Corporation Systems Research Center, November 1989.

[3] Matthias Felleisen and Daniel P. Friedman. A closer look at export and import statements. *Computer Languages*, 11(1):29–37, 1986.

[4] Charles M. Geschke, James H. Morris, Jr., and Edwin H. Satterthwaite. Early experience with Mesa. *ACM SIGPLAN Notices*, 12(3):138–152, March 1977.

[5] Robert Harper, Robin Milner, and Mads Tofte. *The Definition of Standard ML, Version 3*. Technical Report ECS-LFCS-89-81, University of Edinburgh, 1989.

[6] Eugene E. Kohlbecker, Jr. *Syntactic Extensions in the Programming Language Lisp*. Technical Report 199, Indiana University Computer Science Department, August 1986.

[7] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[8] Massachusetts Institute of Technology. *MIT Scheme Reference*. Scheme Release 7 edition, June 1989. (Draft).

[9] Jonathan Rees. *Modular Macros*. Master's thesis, Massachusetts Institute of Technology, May 1989.

[10] Jonathan Rees, William Clinger, et al. Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12), December 1986.

[11] Jonathan Rees, William Clinger, et al. Revised^4 report on the algorithmic language Scheme. (Draft).

[12] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual*. Yale University, fourth edition, January 1984.

[13] Mark A. Sheldon. *Static Dependent Types for First-Class Modules*. Master's thesis, Massachusetts Institute of Technology, June 1989.

[14] Guy L. Steele Jr. *Common LISP: The Language*. Digital Equipment Corporation (Digital Press), 1984.

[15] United States Department of Defense. *Ada Reference Manual*. July 1980.

[16] Mark Weiser, Alan Demers, and Carl Hauser. The Portable Common Runtime approach to interoperability. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 114–122, December 1989.