Another Module System for Scheme

Jonathan Rees
3 January 1993 (updated 15 January 1994)

This memo describes a module system for the Scheme programming language. The module system is unique in the extent to which it supports both static linking and rapid turnaround during program development. The design was influenced by Standard ML modules[4] and by the module system for Scheme Xerox[3]. 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.

Except where noted, everything described here is implemented in Scheme 48, and exercised by the Scheme 48 implementation and a few application programs.

Unlike the Common Lisp package system, the module system described here controls the mapping of names to denotations, not the mapping of strings to symbols.

Introduction

The module system supports the structured division of a corpus of Scheme software into a set of modules. Each module has its own isolated namespace, with visibility of bindings controlled by module descriptions written in a special *configuration language.*

A module may be instantiated multiple times, producing several *packages*, just as a lambda-expression can be instantiated multiple times to produce several different procedures. Since single instantiation is the normal case, I will defer discussion of multiple instantiation until a later section. For now you can think of a package as simply a module's internal environment mapping names to denotations.

A module exports bindings by providing views onto the underlying package. Such a view is called a *structure* (terminology from Standard ML). One module may provide several different views. A structure is just a subset of the package's bindings. The particular set of names whose bindings are exported is the structure's *interface.*

A module imports bindings from other modules by either *opening* or *accessing* some structures that are built on other packages. When a structure is opened, all of its exported bindings are visible in the client package. 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))))

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

This configuration defines two structures, `foo` and `bar`. `foo` is a view on a package 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 package, `foo` only exports those three. Similarly, structure `bar` consists of the binding of `d` from a package 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.

A module's body, the part following `begin` in the above example, is evaluated in an isolated lexical scope completely specified by the package definition's `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.

The configuration language

The configuration language consists of top-level defining forms for modules 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 underlying package 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 package can be seen via that structure in other packages.

An `open` clause specifies which structures will be opened up for use inside the new package. At least one package must be specified or else it will be

⟨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⟩*`)`
  | `(optimize` ⟨optimize-spec⟩*`)`
  | `(for-syntax` ⟨clause⟩*`)`
⟨interface⟩ ⟶ `(export` ⟨item⟩*`)`
  | ⟨name⟩
  | `(compound-interface` ⟨interface⟩*`)`
⟨item⟩ ⟶ ⟨name⟩ | `(`⟨name⟩ ⟨type⟩`)` | `((`⟨name⟩*`)` ⟨type⟩`)`

Figure 1: The configuration language.

impossible to write any useful programs inside the package, since `define`, `lambda`, `cons`, `structure-ref`, etc. will be unavailable. Typical packages 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` package that exports the operators of the configuration 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 package 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 a package accesses any structures, it should probably open the `structure-refs` structure so that the `structure-ref` operator itself will be available.

The package's 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 package definition, 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 package. For example, one can do `(define car 'chevy)` without affecting the binding of the name `car` in the `scheme` package.

Assignments (using `set!`) to imported and undefined variables are not allowed. In order to `set!` a top-level variable, the package 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 of a package's opened structures to export two different bindings for the same name. However, the current implementation does not check for this situation; a name's binding is always taken from the structure that is listed first within the `open` clause. This may be fixed in the future.

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. You can't at present put an absolute path name in the `files` list.

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 macros 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 package definitions 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 module's specification (interface) from its implementation (package).

3. My experience is that configurations that are separated into interface definitions and package definitions are easier to read; the long lists of exported bindings just get in the way most of the time.

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.

Macros

Hygienic macros, as described in [1, 2], 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 is defined by the rewrite rule

```
(delay exp)  ==>  (make-promise (lambda () exp)).
```

The variable `make-promise` is defined in the `scheme` structure's underlying package, 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 package.

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.

```
⟨definition⟩ ⟶ (define-module (⟨name⟩ (⟨name⟩ ⟨interface⟩)*)
                  ⟨definition⟩*
                  ⟨name⟩)
              | (define ⟨name⟩ (⟨name⟩ ⟨name⟩*))
```

Compiling and linking

Scheme 48 has a static linker that produces stand-alone heap images from 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 (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 package, not in the package being loaded or linked.

[I need to describe structure reification.]

Semantics of configuration mutation

During program development it is often desirable to make changes to packages 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 configuration, 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-structure 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 package. 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 package 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 package'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.

Command processor support

While it is possible to use the Scheme 48 static linker for program development, it is far 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.

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 package that is distinct from the current package for evaluating Scheme forms given to the command processor. The command processor interprets Scheme forms in a particular current package, and there are commands that move the command processor between different packages.

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 package followed by a greater-than (`>`).

`,config`

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

`,config` *command*

This form of the `,config` command executes another command in the current configuration package. For example,

```
,config ,load foo.scm
```

interprets configuration language forms from the file `foo.scm` in the current configuration package.

`,in` *struct-name*

The `,in` command moves the command processor to a specified structure's underlying package. 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 a package called `user`, but the `,config` command moves it into the configuration package, 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 package.

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

`,in` *struct-name command*

This form of the `,in` command executes a single command in the specified package without moving the command processor into that package. Example:

```
,in mumble (cons 1 2)
,in mumble ,trace foo
```

`,user` [*command*]

This is similar to the `,config` and `,in` commands. It moves to or executes a command in the user package (which is the default package when the Scheme 48 command processor starts).

`,for-syntax` [*command*]

This is similar to the `,config` and `,in` commands. It moves to or executes a command in the current package's "package for syntax," which is the package in which the forms *f* in `(define-syntax` *name f*`)` are evaluated.

`,load-package` *struct-name*

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

`,reload-package` *struct-name*

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

`,load` *filespec* ...

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

`,structure` *name interface*

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

`,open` *struct-name**

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

Configuration packages

It is possible to set up multiple configuration packages. The default configuration package opens the following structures:

- `module-system`, which exports `define-structure` and the other configuration language keywords, as well as standard types and type constructors (`:syntax`, `:value`, `proc`, etc.).
- `built-in-structures`, which exports structures that are built into the initial Scheme 48 image; these include `scheme`, `tables`, and `records`.
- `more-structures`, which exports additional structures that are available in the development environment; these include `sort`, `random`, and `threads`.

Note that it does not open `scheme`.

You can define other configuration packages by simply making a package that opens `module-system` and, optionally, `built-in-structures`, `more-structures`, or other structures that export structures and interfaces.

For example:

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

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

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

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[5].
- 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[6]) 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 packages, 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, 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.

References

[1] William Clinger and Jonathan Rees. Macros that work. *Principles of Programming Languages*, January 1991.

[2] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers* IV(3):1–55, July-September 1991.

[3] Pavel Curtis and James Rauen. A module system for Scheme. *ACM Conference on Lisp and Functional Programming,* pages 13–19, 1990.

[4] David MacQueen. Modules for Standard ML. *ACM Conference on Lisp and Functional Programming,* 1984.

[5] Jonathan Rees and Bruce Donald. Program mobile robots in Scheme. *International Conference on Robotics and Automation,* IEEE, 1992.

[6] 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.