LISP AND SYMBOLIC COMPUTATION: An International Journal, 9, 343–358, 1996

Reliable Interactive Programming with Modules*

SHO-HUAN SIMON TUNG (*tungsh@mis.yuntech.edu.tw*)

Department of Management and Information Systems
National Yunlin Institute of Technology
Touliu, Taiwan, R.O.C

R. KENT DYBVIG (*dyb@cs.indiana.edu*)

Computer Science Department
Indiana University
Bloomington, IN 47405 USA

(Received: October, 1994)

(Revised: June, 1995)

Keywords: Modules, Linking, Loading, Interactive Programming, Programming Environments.

Abstract. Interactive programming is a convenient programming style that supports fast prototyping and debugging but often results in a loss of modularity and security. This article addresses the problem of supporting reliable and modular interactive programming. A module system designed for interactive use is introduced. This module system supports separate compilation and automatic updating of module interfaces during program development. It also allows the programmer to obtain a fresh program state for reliable testing, eliminating the need to reload an entire program.

1. Introduction

Modular programming facilitates large-scale program development and system maintenance by providing a clean way for programs to be organized and enforcing barriers between program pieces. In module systems supporting a batch-oriented programming style, each module of a program is compiled separately and linked with other modules before execution. Upon discovery of an error, the programmer modifies and recompiles one or more modules of the program, and the integrity of the resulting executable is maintained by recompiling dependent pieces via software utilities such as the Unix *make* program [5].

Interactive programming environments allow programmers to alter existing definitions by entering new definitions directly into the system. Without

*A preliminary version of this article was presented at the 1992 ACM Conference on Lisp and Functional Programming.

automatic maintenance of module interfaces, which traditional interactive programming environments do not provide, interactive modifications to module interfaces are not propagated so that subsequent execution within the interactive system is inherently unreliable, reducing the utility of the interactive programming environment. Module systems designed for interactive programming languages and environments must address this *module consistency* problem.

A different but related problem is the *program text and system state consistency* problem. For interactive and imperative languages, changes to variables and objects caused by the evaluation of one expression in an interactive session may affect subsequent evaluation, even if module interfaces are not affected. As a result, the program text at any given point in an interactive programming session cannot be relied upon to understand the program's behavior [10]. This problem and the module consistency problem can make modular programming in interactive and imperative languages confusing and unreliable.

The approach we have taken to solve the module consistency problem is to design a module system with two levels, the meta level and the base level. The meta level provides a set of language constructs for defining modules and relationships among modules. The base level is used to describe the actual computation. During program development, the connection between the meta level and the base level is achieved by extending the variable lookup semantics of the base level. For fully developed applications, however, the meta level is unnecessary and can be compiled away.

The program text and system state consistency problem is addressed by extending the meta level system with additional syntactic constructs that force the programmer to write programs that can be reinitialized reliably. As a result, the module system achieves the convenience of interactive programming with the reliability of batch-style modular programming.

The remainder of this article is organized as follows. Section 2 reviews related work. Section 3 presents the semantics of the module system using a minimal base language and shows that the module system solves the module consistency problem. Section 4 discusses problems caused by side effects during interactive programming and presents an extended module system that solves the program text and system state consistency problem. Section 5 describes an implementation of the module system. Section 6 presents our conclusions.

2. Related Work

Friedman and Felleisen's module system [6] is one of only a few designed with consideration for interactive programming. Interactive languages require late binding for flexible program development. By restricting exported values to procedures, Friedman and Felleisen support various late binding techniques for defining modules and importing items. The ultimate goal is to allow arbitrary load orders for modules and to permit interactive extension and redefinition of bindings in modules. This goal is not entirely achieved, however, since interactively added bindings cannot access lexical variables in a module. This restriction is too strong to be acceptable.

Some Scheme implementations support first-class environments [1]. A first-class environment captures the current lexical environment at the point when the first-class environment is created. When used with `eval` or `access`, which allow expressions or variables to be evaluated dynamically in a first-class environment, a form of modular programming can be supported. First-class environments, however, are inherently dynamic in nature, so that module interfaces often cannot be fully determined until run-time.

Common Lisp's [11] package system uses symbol tables to represent modules. Symbols defined as external in a package can be exported. Various mechanisms are available to access or to import exported symbols. The package system could be used as the low-level implementation for fully developed (static) modules. The association of symbols to packages is fixed at read time, however, and can be changed only by rereading and recompiling all affected code, which is undesirable in an interactive programming system. Furthermore, information hiding is not enforced by the package system.

Queinnec and Padget have designed a module system for Lisp [9]. The design goals of their system are to support separate compilation and to control the visibility of resources. A module is available for use after it is defined and loaded. In the module definition phase, the module's top-level environment is established with imported bindings for free variables in the module. In the module loading phase, the module body is evaluated. In their system, imported variables are bound early in the module definition phase. This requires the module dependency to be acyclic and defeats possibilities for flexible interaction.

Curtis and Rauen recently proposed a module system designed for large-scale programming in Scheme [3]. In their proposal, a module is an isolated scope which may be nested inside other modules. Modules are anonymous. They communicate with each other by sharing items specified in *interfaces*. Interfaces are named environments that may contain both syntax bindings and value bindings. They did not address, however, the problem of fitting

their system into Scheme's interactive programming style.

Talk is a modern Lisp dialect which contains a module system, a metaobject protocol, a transparent interface to C and C++, and an extensive set of libraries [4]. Talk's module system is designed to support easy and efficient application delivery. An executable Talk application contains several separately compiled modules and libraries. Talk distinguishes between compilation and execution dependencies. Compilation units can be automatically excluded from the delivered executable. The process of program development in Talk, however, adheres closely to the traditional compile-link-execute model.

Standard ML is a statically scoped functional programming language with a secure polymorphic type system [7]. The module extension to Standard ML associates each module with two environments: a *signature* containing the interface of the module, and a *structure* containing the implementation of the module [8]. However, ML's type system limits its flexibility as an interactive language and makes it difficult to support arbitrary module load order.

Modula-2 [15], Modula-3 [2], and Ada [14] are conventional modular languages. Modula-2 separates the definition of a module from its implementation, and it requires a one-to-one correspondence between the two components. Modula-3 differs from Modula-2 by allowing an implementation module to be associated with several interface specifications. This facility allows exporting of different levels of detail of an implementation to different importing modules. Ada supports generic packages. Generic packages, however, can be instantiated only statically through declarations. None of these systems support interactive programming.

3. The Module System

Most modular programming languages define a module in terms of three subcomponents: an import specification, public variables and their values, and private variables and their values. A module hides details about its implementation using private variables, provides services by exporting its public variables, and receives services by binding free variable references within the module to imported variables or variables defined in the module.

3.1. Syntax and Semantics

In addition to supporting import, private, and public definitions, the module system presented here has the following features:

- it provides the flavor and benefits of lexical scoping;
- it supports mutually referential modules and allows arbitrary module

$$
\begin{array}{rcll}
c & \in & \mathit{Con} & = \{\text{undefined}, +, -, \ldots, 0, 1, \ldots\} \quad \text{constants} \\
m,\ i & \in & \mathit{Ide} & \quad \text{identifiers} \\
s & \in & \mathit{Stmt} & \quad \text{statements} \\
e & \in & \mathit{Exp} & \quad \text{expressions}
\end{array}
$$

$$
\begin{array}{rcl}
s & ::= & (\textbf{module}\ m\ \textbf{import}\ m^*) \\
 & | & (\textbf{public}\ m\ i\ e) \\
 & | & (\textbf{private}\ m\ i\ e) \\
 & | & (\textbf{with}\ m\ e) \\
 & | & s\ s \\
e & ::= & c \\
 & | & i \\
 & | & (\textbf{lambda}\ (i)\ e) \\
 & | & (e\ e)
\end{array}
$$

Figure 1: λ_{imp} syntax

load orders; and

- it permits interactive extension and redefinition of module bindings.

Figure 1 presents the syntax of a simple language, λ_{imp}, which extends the call-by-value λ-calculus with modular programming constructs. λ_{imp} is intended to be used as the semantic foundation for module systems for interactive call-by-value programming languages.

The syntax of λ_{imp} is divided into two levels, the meta level and the base level. The meta level consists of **module**, **private**, **public**, and **with** statements. The first three statements are used to support interactive operations that a user might perform to define bindings in modules and relationships among modules. The **with** statement allows a user to test a program by evaluating expressions within environments associated with a module.

The base level consists of constants, variables, abstractions, and applications. These expressions are merely λ-calculus expressions and are used to describe the actual computation performed by a program. For the remainder of this section we assume that the base level contains no assignment operators, since we are concerned primarily with scoping of variables, and assignments do not affect scoping.

Figure 2 presents the semantic domains for λ_{imp}. These domains include environments, module environments, constant values, and expressed

ρ	$\in$	$Env = Ide \rightarrow E$	environments
μ	$\in$	$MEnv = Ide \rightarrow (Env \times Env \times Ide^*)$	module environments
		C	constant values
ϵ	$\in$	$E = C + (MEnv \rightarrow E \rightarrow E)$	expressed values

Figure 2: λ_{imp} semantic domains

values. Environments are functions that map identifiers (variables) to expressed values. Module environments are functions that map identifiers (module names) to modules. Each module has three components: a private environment, a public environment, and imports. The imports of a module consist of zero or more identifiers which are the names of other modules. Expressed values include constant values and function values. Function values take a module environment as an implicit argument.

The concept of free versus bound variables is essential to the semantics of λ_{imp} as it is to the λ-calculus. Within an expression, the value of a bound variable can be determined locally; the value of a free variable, however, must be obtained from the surrounding context. Similarly, within a module, free variables must obtain their values from definitions in the module or imported from other modules. The rules for determining the free variables of a λ_{imp} expression mirror those for the λ-calculus. The rules for the meta-level constructs are straightforward: a λ_{imp} **module** statement contains no free variables, and if a variable occurs free in the expression part of a **private**, **public**, or **with** statement then it occurs free in the statement as a whole.

Figure 3 presents the semantic functions for λ_{imp}. We assume that an initial environment (represented by ρ_0) maps every identifier to the undefined value, and initially a module environment maps every identifier to a triple consisting of an initial environment for private bindings, an initial environment for public bindings, and an empty set indicating no imported modules. The **module** statement allows the import of a module to be defined or modified. A **private** or **public** statement evaluates its expressions and returns a new module environment that contains the new binding in the corresponding module's private or public environment. These three statements all return a sequence consisting of a new module environment and the "undefined" value.

The **with** statement evaluates its subexpression using a module environment, a module name, and an initial lexical environment. It returns a sequence consisting of the unchanged module environment and a value. Since **with** is not an expression, the value is used only for display purposes

Notations:

$\langle \ldots \rangle$	sequence formation
$s \downarrow k$	kth member of the sequence s (1-based)
$(\rho[i \leftarrow x])\ i'$	$(i' = i) \rightarrow x, \rho\ i'$
$x\ in\ D$	injection of x into domain D
$x \mid D$	projection of x to domain D

Semantic functions:

$\mathcal{S}$: $Stmt \rightarrow MEnv \rightarrow (MEnv \times E)$
$\mathcal{K}$: $Con \rightarrow E$
$\mathcal{E}$: $Exp \rightarrow Menv \rightarrow Ide \rightarrow Env \rightarrow E$

$\mathcal{S}\,[\![(\textbf{module}\ m_0\ \textbf{import}\ m^*)]\!]\,\mu =$
$\quad \langle \mu[m_0 \leftarrow \langle \mu m_0 \downarrow 1, \mu m_0 \downarrow 2, m^* \rangle], \text{undefined} \rangle$
$\mathcal{S}\,[\![(\textbf{private}\ m\ i\ e)]\!]\,\mu =$
$\quad \langle \mu[m \leftarrow \langle (\mu m \downarrow 1)[i \leftarrow \mathcal{E}\,[\![e]\!]\mu m \rho_0], \mu m \downarrow 2, \mu m \downarrow 3 \rangle], \text{undefined} \rangle$
$\mathcal{S}\,[\![(\textbf{public}\ m\ i\ e)]\!]\,\mu =$
$\quad \langle \mu[m \leftarrow \langle \mu m \downarrow 1, (\mu m \downarrow 2)[i \leftarrow \mathcal{E}\,[\![e]\!]\mu m \rho_0], \mu m \downarrow 3 \rangle], \text{undefined} \rangle$
$\mathcal{S}\,[\![(\textbf{with}\ m\ e)]\!]\,\mu = \quad \langle \mu, \mathcal{E}\,[\![e]\!]\mu m \rho_0 \rangle$
$\mathcal{S}\,[\![s_0\ s_1]\!]\,\mu = \quad \mathcal{S}\,[\![s_1]\!]\,(\mathcal{S}\,[\![s_0]\!]\,\mu) \downarrow 1$

$\mathcal{E}\,[\![c]\!]\mu m \rho = \mathcal{K}\,[\![c]\!]$
$\mathcal{E}\,[\![i]\!]\mu m \rho = lookup\ \mu\ m\ \rho\ i$
$\mathcal{E}\,[\![(\textbf{lambda}\ (i)\ e)]\!]\mu_0 m \rho =$
$\quad \lambda \mu_1.[strict(\lambda \epsilon.\mathcal{E}\,[\![e]\!]\mu_1 m(\rho[i \leftarrow \epsilon]))]\ in\ E$
$\mathcal{E}\,[\![(e_0\ e_1)]\!]\mu m \rho =$
$\quad (\mathcal{E}\,[\![e_0]\!]\mu m \rho \mid (MEnv \rightarrow E \rightarrow E))\mu(\mathcal{E}\,[\![e_1]\!]\mu m \rho)$

Figure 3: λ_{imp} semantic functions

by the interactive environment. Consecutive meta-level statements are evaluated by passing the module environment returned from one statement to the next.

In addition to modifying a module's private bindings, public bindings, and imports directly, a few features of the semantics of **lambda** expressions, applications, and variables are essential to the semantics of the λ_{imp} language:

- The semantics enforces call-by-value evaluation. This is indicated by the use of the *strict* function in the semantic function for abstractions.
- The lexical environment is closed in the procedure value of an ab-

straction.
- The name of the module in which the abstraction is evaluated is also closed in the procedure value of the abstraction.
- The module environment is *not* closed but is instead provided dynamically as an implicit argument when the procedure is applied.
- The value of a variable is determined from its lexical environment, its module name, and a module environment. The module name may or may not be the current module name.

The search for a variable's value starts from the lexical environment, followed by the private and public bindings of the module corresponding to the module name, followed by public bindings of modules imported by the module. If a variable is referenced but is not imported or defined locally, the search terminates with an "undefined" value.

The essence of the semantics is that bindings of free occurrences of variables in a procedure are determined dynamically using the use-time module environment. The dynamic search is confined, however, to the three use-time components (private environment, public environment, and imports) of the module within which the procedure is created. Since the user can modify the bindings and imports of each module in a program, free occurrences of variables in a module are sensitive to such modifications. In addition, bound occurrences of variables are determined from the lexical environment. The resulting semantics preserves lexical scoping while providing flexible interactive capabilities.

Figure 4 presents the auxiliary functions *lookup* and *lookupM*. Although a direct implementation of these functions would be inefficient, a technique is presented in Section 5 that allows the variable-lookup semantics of the λ_{imp} language to be implemented efficiently.

The semantics of λ_{imp} requires some variables to be looked up dynamically based on a module's import list. This should not be confused with the kind of dynamic lookup required to support dynamic object-oriented programming languages. The distinction is that in λ_{imp} the bindings are static during program evaluation, since only meta-level statements can alter scoping, whereas bindings may change at run time in dynamic object-oriented programming languages as the same method code is applied to different types of objects. This distinction is the basis for the efficient variable-lookup mechanism described in Section 5.

3.2. Examples

To simplify the presentation of the scope control mechanisms of λ_{imp}, we have chosen λ-calculus as its base language. In this section we assume that λ_{imp} has been extended with various Scheme constructs and primitives,

```
lookup: MEnv → Ide → Env → Ide → E
lookupM: MEnv → Ide* → Ide → E

lookup μ m ρ i =
    if ρ i = undefined then
        if (μm ↓ 1) i = undefined then
            if (μm ↓ 2) i = undefined then
                lookupM μ (μm ↓ 3) i
            else (μm ↓ 2) i
        else (μm ↓ 1) i
    else ρ i

lookupM μ [[]] i = undefined
lookupM μ [[first rest*]] i =
    if (μfirst ↓ 2) i = undefined then
        lookupM μ rest* i
    else (μfirst ↓ 2) i
```

Figure 4: Auxiliary functions

such as **if** and +.

Figure 5 presents an example λ_{imp} program. This example involves two modules. The module A defines and exports a *square* procedure that is initially incorrect. The module B imports module A and uses the *square* procedure to define a *distance* procedure that calculates the distance between a point and the origin. The user can test individual procedures using **with** statements. Upon realizing that the definition of *square* is incorrect, the user can redefine the *square* procedure and test it again. This example demonstrates that a procedure may be redefined interactively with its new definition available automatically to other modules that use the procedure.

Figure 6 presents another example λ_{imp} program. This example involves three modules. Both *triangle* and *rectangle* export an *area* procedure. The *compute-area* module can import an *area* procedure from either the *triangle* module or the *rectangle* module using the **module** statement. This example demonstrates that a module's import may be modified interactively during program development. It also illustrates a potential problem regarding name conflicts with imported variables: the *compute-area* module cannot use the two different *area* procedures of the *triangle* module and the *rectangle* module simultaneously. This problem is solved with a more general import specification. The following definition:

```
(module A import)
(public A square
  (lambda (x) (+ x x)))

(module B import A)
(public B distance
  (lambda (x y)
    (sqrt (+ (square x) (square y)))))

(with B (distance 3 4)) ⇒ 3.74
(public A square
  (lambda (x) (* x x)))
(with B (distance 3 4)) ⇒ 5
```

Figure 5: Interactive redefinition

```
(module compute-area import (triangle (rectangle (rec-area area))))
```

declares that all the public bindings of the module *triangle* and the *area* binding of the module *rectangle* are imported by the module *compute-area* with *area* renamed as *rec-area*.

The third example is presented in Figure 7. This example uses modules *A* and *B* from Example 1 and adds a *compare* module that exports the procedure *less-than-4?*, which computes whether the distance of a point from the origin is less than 4. The call to *distance* in *less-than-4?* indirectly calls the *square* procedure defined in module *A* rather than the *square* procedure locally defined in the *compare* module. This example demonstrates the importance of using the definition-time module name to look up free variables. If the use-time module name were used to look up free variables in exported procedures, the call to *less-than-4?* would erroneously return #t.

The last example is presented in Figure 8. This example uses two mutually referential modules that import and export to each other to implement mutually recursive *even?* and *odd?* procedures. Note that the **with** statement can refer to imported variables in addition to variables locally defined in a module. Support for mutually referential modules greatly increases flexibility, especially when constructing large libraries that interact in nontrivial ways.

```
(module triangle import)
(public triangle area
  (lambda (x y)
    (/ (* x y) 2)))

(module rectangle import)
(public rectangle area
  (lambda (x y)
    (* x y)))

(module compute-area import triangle)
(private compute-area compute
  (lambda (x y)
    (area x y)))

(with compute-area (compute 3 4)) ⇒ 6

(module compute-area import rectangle)

(with compute-area (compute 3 4)) ⇒ 12
```

Figure 6: Modifying imports

4. Program Text and System State Consistency

During an interactive programming session, a programmer typically loads a program into the interactive environment then repeatedly tests and modifies the program to enhance its functionality or eliminate errors. During this process, the value of each program run can be affected by earlier changes to the set of public and private variables, changes in the definitions of public or private values, and changes to system state resulting from assignments to **public** or **private** variables or data structure mutations.

The variable-reference semantics of λ_{imp} guarantees that after such changes, all variable references resolve to the proper locations. It does not, however, guarantee that the values in those locations are consistent with the program text, as the sample interactive session in Figure 9 demonstrates. Immediately after the definition of *fun*, the call (**with** *m* (*fun* 3)) returns (4 . 4). After redefining *op*, however, (**with** *m* (*fun* 3)) returns (-1 . 5) rather than (-2 . -2). This incorrectly reflects the current program text, since the values of *op1* and *x* do not have the expected values. The value of *op1* is incorrect because *op1* is bound to the value of *op* before the

```
(module A import)
(public A square
  (lambda (x) (* x x)))

(module B import A)
(public B distance
  (lambda (x y)
    (sqrt (+ (square x) (square y)))))

(module compare import B)
(public compare less-than-4?
  (lambda (x y)
    (< (distance x y) 4)))
(private compare square
  (lambda (x) (+ x x)))

(with compare (less-than-4? 3 4)) ⇒ #f
```

Figure 7: Lexical principle

redefinition of *op*, in effect caching the old value of *op*. The value of x is incorrect because it is altered by the first call to *fun* and not reset for the second call. In general, without proper reinitialization of the program state after each test run, subsequent runs occur in a contaminated state. This problem is easily overlooked by the programmer, who may draw erroneous conclusions from the behavior of the program running in the contaminated state.

Several approaches may be used to solve these problems. One is to reevaluate all variable definitions, including those that do not rely on the altered bindings. This is essentially the traditional batch-oriented programming style. Another approach is to provide undo/redo facilities to the programmer, as in Interlisp [12]. The programmer, however, is required to keep track of dependencies among variable definitions and to undo/redo selected ones manually. This process is burdensome and error prone. Furthermore, some side effects are expensive to undo. A third approach is to have the system keep track of the dependencies among definitions. After a variable is redefined by the user, the system automatically reevaluates those affected definitions only. Unfortunately, doing so is difficult, since a simple dependency analysis may lead to unnecessary recomputation, and a thorough analysis may result in more total overhead than simply reloading

```
(module even import odd)
(public even even?
  (lambda (x)
    (if (= 0 x)
        #t
        (odd? (- x 1)))))

(module odd import even)
(public odd odd?
  (lambda (x)
    (if (= 0 x)
        #f
        (even? (- x 1)))))

(with even (odd? 3)) ⇒ #t
(with odd (even? 2)) ⇒ #t
```

Figure 8: Mutually referential modules

the program [13].

We have chosen instead to rely upon a combination of syntactic restrictions and programming conventions to solve the program text and system state consistency problem. The syntactic restrictions force all variable references to occur at run time, not at **public**/**private** variable definition time as with the reference to *op* in Figure 9. This is accomplished by requiring the initialization expression of each **public** and **private** variable to be a **lambda** expression. Furthermore, bindings established by **public** and **private** definitions are immutable, *i.e.*, assignment operators in the base language are enjoined from assigning **public** or **private** variables.

These restrictions effectively guarantee program text and system state consistency, but also prevent modules from maintaining any local state. Local state is supported via assignable private variables, which are established by a new meta-level **private!** construct. **private!** establishes bindings for assignable variables, while leaving their values uninitialized. Figure 10 presents the revised meta-level syntax of the λ_{imp} language.

By convention, **private!** variables are initialized in explicitly coded initialization procedures, and responsibility for invoking these initialization routines rests with the programmer. The programming environment should detect and report references to uninitialized **private!** variables. To ensure consistency, the programming environment should also reset

```
> (private m op +)
> (private m fun
     (let ([op1 op]
           [x 0])
       (lambda (y)
         (set! x (+ x 1))
         (cons (op x y) (op1 x y)))))
> (with m (fun 3))
(4 . 4)
> (private m op -)
> (with m (fun 3))
(-1 . 5)
```

Figure 9: Consistency problems

all **private!** variables to unititialized whenever any changes are made to **public**, **private**, or **private!** definitions.

These restrictions result in no inherent loss of functionality. Suppose we wish to export the value of the variable x and to allow x to be assignable by other modules. We simply export two procedures: a reference procedure, *e.g.*, (**lambda** () x), and an assignment procedure, *e.g.*, (**lambda** (v) (**set!** x v)). Similarly, suppose that we wish to export a variable y, giving it the value of (**let** ([x (*proc*)]) (**lambda** (*arg*) ...)), where x may be assigned within (**lambda** (*arg*) ...). We instead define x as **private!**, put (**set!** x (*proc*)) in an exported initialization procedure, and export the value of (**lambda** (*arg*) ...) as the value of y.

These restrictions have the side benefit that they encourage programmers to write programs that are more easily analyzed by both compilers and programmers, since any code that can assign a variable is insulated within a single module. A programmer and compiler can simply scan a module to determine which variables are assignable and can more often determine the types of values assigned to those variables. Furthermore, these restrictions naturally lead to the use of assignment procedures that ensure the new value is in the range of acceptable values. For example, if an assignable variable must take on only positive values, the assignment procedure might be written as:

```
(lambda (v)
  (if (positive? v)
      (set! x v)
      (error)))
```

$$
\begin{array}{rcl}
s & ::= & (\textbf{module}\ m\ \textbf{import}\ m^*) \\
& | & (\textbf{public}\ m\ i\ \lambda\text{-}expr) \\
& | & (\textbf{private}\ m\ i\ \lambda\text{-}expr) \\
& | & (\textbf{private!}\ m\ i) \\
& | & (\textbf{with}\ m\ e) \\
& | & s\ s
\end{array}
$$

Figure 10: Revised λ_{imp} syntax

which results in safer, more readable, and more easily analyzed code.

Another benefit is that definition-time computation involving potentially order-dependent variable references disappears. Such computation is contained instead in explicitly defined initialization procedures associated with each module. Modules can be therefore loaded in arbitrary order, although the programmer must still order the invocation of initialization procedures. Incorrect ordering will typically result in reference to uninitialized variables and hence an error reported by the programming environment.

5. Implementation

The preceding two sections present the design of a module system that supports reliable interactive programming. This section discusses some user-interface issues and techniques for implementing the system efficiently.

5.1. Module Structure

The syntax of λ_{imp} does not enforce any module structure. It even allows statements in one module to be interleaved with statements in other modules. This flexibility is needed to support interactive programming, but a more structured approach to writing modules is generally preferable. The **in-module** syntactic form can be used to group definitions in a module:

(**in-module** *stack*
  (**import** (*list*))
  (**private!** *my-stack*)
  (**public** *init-stack* ...)
  ...)
⇒

(**module** *stack* **import** (*list*))
(**private!** *stack my-stack*)
(**public** *stack init-stack* ...)
...

It is possible to break a large module into several **in-module** forms. Since the module system disallows load-time computation, the different pieces of a module can be loaded in arbitrary order. To aid program analysis, it may be preferable, however, to require that each module be wholly contained in one **in-module** form.

5.2. User Interface

In order to provide convenient interactions between the programmer and the module system, the interactive user interface (typically some form of read-eval-print loop, or REPL) must become "module-sensitive." A simple design is to use the REPL prompt to represent the name of a "current" module. Expressions or definitions entered at the prompt for a given module are evaluated in the module environment of the module. The current module can be changed to some other module upon request. The **with** syntactic form would become unnecessary, as would the module specifier in **public**, **private**, and **private!** forms.

A similar but more involved design is to use a window-based user interface that associates each module with a file and an edit window [13].

An interactive user interface should provide some mechanism for eliminating modules and bindings within modules.

5.3. Module Linkage

The semantics of our module system requires free-variable bindings to be determined dynamically. This property provides the necessary flexibility for interactive programming. A naive implementation of the semantics, however, can result in unacceptable performance. The implementation presented here keeps track of the import/export relations among modules and uses double indirections with an implicit incremental link step after each interactive modification to resolve the bindings of free variables. As we describe in Section 5.4, these indirections are not needed in fully developed code.

The linker keeps track of variable bindings and free variables for each module, maintaining for each module environments for public variables, private variables, and free variables. The public and private environments associate variables with the locations that contain their values. The free-variable environments (FVE) associate free variables with locations that

```
(in-module cpu
  (import (memory))
  (private! bus-size)
  (public init-cpu
    (lambda (n)
      (set! bus-size n)
      (init-memory 65535)))
  (public maxmem
    (lambda () (expt 2 bus-size))))

(in-module memory
  (import (cpu))
  (private! size)
  (private! mem)
  (public init-memory
    (lambda (n)
      (set! size n)
      (if (< n (maxmem))
          (set! mem (make-vector n))
          (error)))))
```

Figure 11: *Cpu* and *Memory*

contain pointers to the locations of their values, *i.e.*, pointers to the locations of local or imported variables. A binding in the public or private environment of a module is allocated when a public or private definition is linked into the system. Similarly, a binding in the free-variable environment of a module is allocated when an expression containing the free variable is linked. The linker associates each binding in the free-variable environment with the location of the appropriate local or imported variable, if any. A run-time error is signaled if a free variable is not associated with a location by the time it is referenced. Figures 11 and 12 depict the implementation using two modules, *cpu* and *memory*, that import from each other. (The module *primitives*, which imports system primitives such as *expt*, is assumed to be implicitly imported by both *cpu* and *memory*.)

Interfaces among modules can be modified freely. Possible modifications include changing a module's imports, adding a binding, and modifying an existing binding. To support these modifications, the free-variable environments of affected modules must be updated after every user interaction that changes the dependencies among modules. For example, if a local definition

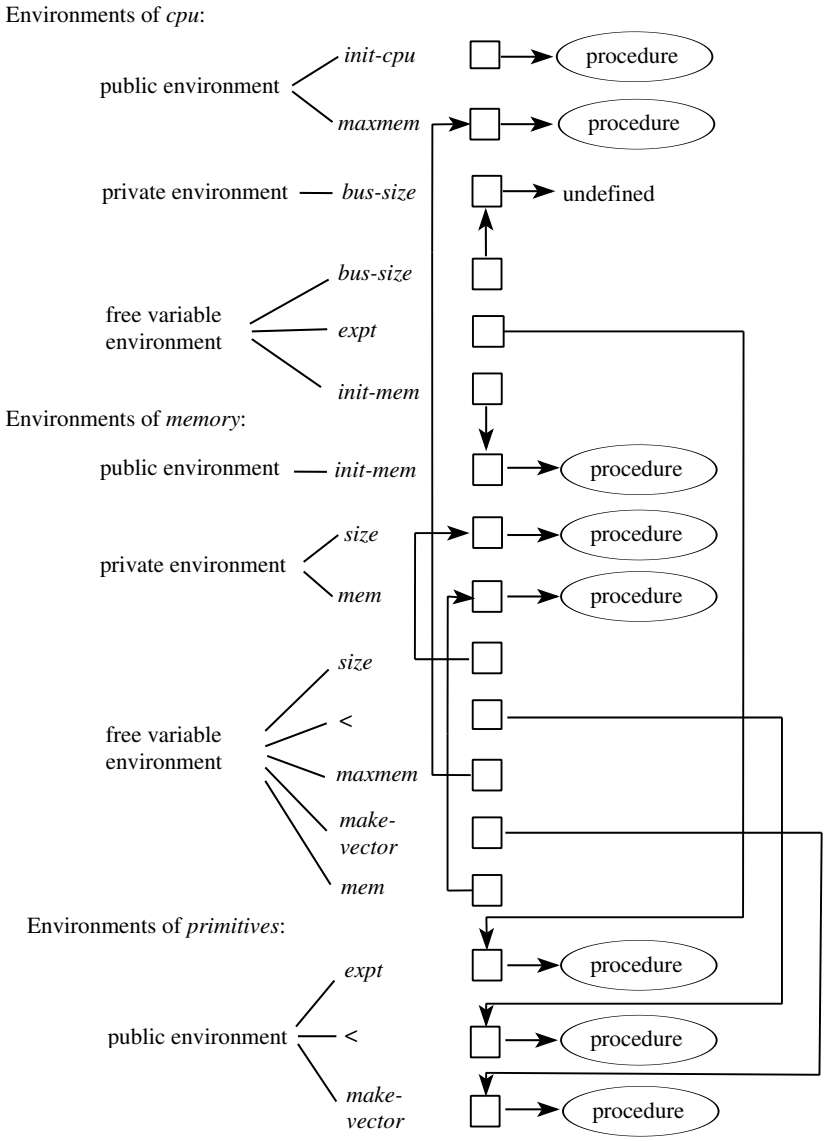

Figure 12: Module linkage

for *maxmem* is added to *memory*, the pointer to *cpu*'s *maxmem* must be replaced with a pointer to the local *maxmem*. In extreme circumstances, the amount of relinking required could be high. In practice, however, this does not appear to be a problem.

5.4. Compiling Developed Programs

When an application has been fully developed and is ready to be put into production use, the meta-level language and the internal module structure need no longer be supported. In particular, the double indirections required to support interactive modifications can be eliminated.

Compiling developed programs is straightforward. Each public and private variable is renamed to a composite name reflecting both the defining module and the original variable name. For example, a variable v in a module m might be renamed m-v. Variables so named become top-level variables, and since they are the only top-level variables, all name conflicts are avoided. Similarly, each free variable reference is converted into a reference to the appropriate top-level variable. Calls to small local or imported procedures should be integrated at this time; this is especially important to avoid the apparent overhead resulting from the use of reference and assignment procedures as described in Section 4. Code for separate modules can be compiled separately as long as sufficient information is recorded for each module to determine the set of variables it exports. It is straightforward to extend this to allow separately compiled, fully developed modules to be combined with modules still under interactive development [13].

6. Conclusion

Interactive programming systems offer convenience and flexibility and can greatly speed program development. Modular programming allows the programmer to create well-structured programs with abstract interfaces among modules that simplify the coding and increase the reliability of each program part. The static nature of traditional module systems and the free-wheeling nature of traditional interactive programming systems, however, are inherently at odds.

The module system described in this article is intended to support interactive modular programming. An important feature of the module system is that a module's free variable bindings are automatically updated to reflect interactive modifications to the module's local and imported variables. Together with restrictions disallowing definition-time variable references and requiring assignable private variables to be initialized explicitly, this feature allows the interactive environment to accurately reflect the current

program text and to support load-order independence and reinitialization without reloading.

The language described in this article is separated into a meta level used for defining modules and their interfaces and a base level used for defining the executable code. Because the meta-level features are not available in the base-level language, the performance of fully developed programs is unaffected by the existence of these features.

Acknowledgements: The authors would like to thank Oscar Waddell for comments on an earlier draft of this article and Carl Bruggeman for the use of his Scheme TeXer for typesetting code.

References

1. Harold Abelson and Gerald J. Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press and McGraw-Hill, 1985.

2. Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report. Technical Report 31, DEC Systems Research Center, 1988.

3. Pavel Curtis and James Rauen. A module system for Scheme. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, 1990.

4. Harley Davis, Pierre Parquier, and Nitsan Séniak. Talking about modules and delivery. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 113–120, 1994.

5. S. I. Feldman. Make: A program for maintaining computer programs. unpublished article.

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

7. Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML. Technical Report ECS-LFCS-89-81, Department of Computer Science, University of Edinburgh, 1989.

8. David MacQueen. Modules for Standard ML. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 198–207, 1984.

9. Christian Queinnec and Julian Padget. Modules, macros and lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123. Plenum Publishing Corporation, 1991.

10. René G. Rodríguez, Bruce F. Duba, and Matthias Felleisen. Can you trust your read-eval-print loop? unpublished manuscript.

11. Guy L. Steele Jr. *Common Lisp, the Language*. Digital Press, second edition, 1990.

12. Warren Teitelman and Larry Masinter. The Interlisp programming environment. *IEEE Computer*, 14(4):25–34, 1981.

13. Sho-Huan Simon Tung. *Merging Interactive, Modular, and Object-Oriented Programming*. PhD thesis, Indiana University, Bloomington, 1992.

14. US Government Department of Defense. The programming language ADA: Reference manual. *Lecture Notes in Computer Science, Vol. 106*, 1981.

15. Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 1983.