TECHNICAL REPORT NO. 305

Towards a System for Interactive Modular Programming

by

Sho-Huan Simon Tung and R. Kent Dybvig

February 1990

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

Towards a System for Interactive Modular Programming*

Sho-Huan Simon Tung and R. Kent Dybvig
Computer Science Department
Indiana University
Bloomington, Indiana 47405

February 23, 1990

1. Introduction

Modular programming is an important programming paradigm for large-scale program development. Well modularized programs are easier to understand and maintain, thereby reducing program development cost. Many programming languages provide facilities for modular programming. These facilities give programmers direct control over the visibility of names among modules using import and export mechanisms.

Interactive programming is an important technique for reducing program development time. An interactive programming system allows a programmer to enter programs or program fragments directly into the system and to receive the output from that program or fragment immediately, reducing the usual compile-link-execute step conceptually to a single evaluate step.

On the surface, it appears that modular programming and interactive programming are inherently incompatible. Interactive programming relies on the ability to make changes easily and dynamically. Modular programming, on the other hand, provides a rigid structure with sometimes extreme requirements for function and variable declarations, especially declarations regarding imported or exported items. However, we believe that these two programming paradigms can be merged. This paper presents an approach toward *interactive modular programming* for the Scheme programming language [RC86,Dyb87b]. Scheme already supports interactive programming but currently does not support modular programming.

In order to support interactive modular programming in Scheme, we must design a user interface that allows interaction among modules. Currently, Scheme programmers interact with the Scheme system through a program known as the "read-eval-print" loop. However, this read-eval-print loop is designed to evaluate ordinary expressions rather than expressions defined in modules. One simple design of a user interace for modular programming is to allow programmers to switch the "current" modules interactively in a read-eval-print loop. This approach is adopted by Common Lisp's package system [Jr84]. However, this approach is not satisfactory since we cannot interact with more than one module at a time.

*This material is based on work supported by Sandia National Laboratories under contract number 75-5466 and by the National Science Fundation under grant number CCR-8803432.

Unlike traditional Lisp systems, which offer limited user interface facilities for modular programming, IMP is a programming environment that supports interactive modular programming through multiple read-eval-print windows. Each read-eval-print window in IMP corresponds to a module. Information defined in modules can be exchanged through interactive import and export.

In order to allow the user to test programs in any window concurrently displayed on the screen, IMP's user interface must allow the read-eval-print windows to run concurrently. This concurency can be implemented with *engines* [HF87,DH89]. A prototype system supporting multiple concurrent read-eval-print windows has been implemented.

In addition to the user interface, we must also take care to design a name space management strategy on which to base the implementation of IMP's module system. Many Scheme systems provide first class environments as a general tool for name space management [AS84]. However, the concept of first class environments is too dynamic to adequately support modular programming. Furthermore, compiling programs that use first class environments efficiently is difficult or impossible. These problems prompted us to design *module environments* to implement IMP's modules.

A computer program is not just written for machines to execute. It must also be readable in order to be maintained. Literate programming [Knu84] is an approach to programming which encourages programmers to write more readable programs. A literate programming system provides tools for organizing a program into a structure that best illustrates the program. These tools encourage programmers to change their programming attitude from the traditional *written-for-machine* attitude to the new *wirtten-for-people* attitude. Using modules as base units to organize Scheme programs suggests an attractive style of literate programming for Scheme.

This paper suggests possible solutions to problems involved with interactive modular programming and outlines research directions. In the next section, we present the user's view of IMP. Section 3 describes module environments and their use in supporting interactive modular programming. We then present an extension to the basic module system to support mutually recursive import and export. Section 5 describes the implementation of IMP using the X window system [SG86] and engines [HF87,DH89].

2. User's View

An IMP program is composed of a collection of modules. Modules can be loaded interactively. Loading a module also creates a window with a read-eval-print loop. The read-eval-print loops run concurrently to allow the user to interact with more than one module. Imports and exports among modules are handled interactively with necessary relinking or reevaluation hidden from the user. The cost of running multiple concurrent read-eval-print windows is high but justifiable in order to provide a flexible development environment for interactive modular programming. However, for fully developed programs, efficiency is more important than flexibility. IMP therefore provides code generation capabilities for fully developed program to run independently. In addition, IMP supports high quality document generation which allows IMP programmers to practice literate programming [Knu84,Tun89]. The remainder of this section describes a typical programming session with IMP.

IMP is invoked by typing imp in the operating system's command shell. For example, typing

```
% imp sample
```

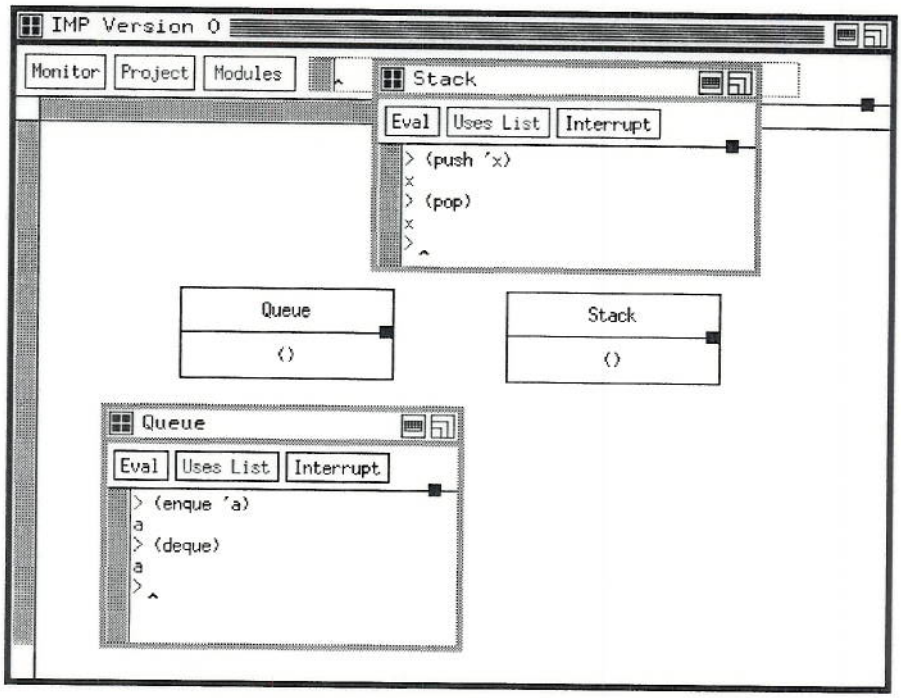

Figure 1: User interface

starts a programming session for the project sample. If the file sample.sp exists in the current directory, IMP loads modules used by the sample project and displays them graphically in the project's window. If sample.sp does not exist, IMP creates an empty project window.

A module is loaded into the system by selecting the Load command from the Module menu. For example, loading the module Stack specified as:

```
(module Stack ()
  (private the-stack '())
  (public empty? <code for empty?>)
  (public push <code for push>)
  (public pop <code for pop>)
  (public print <code for print>))
```

creates a window with a read-eval-print loop for the module Stack. Stack imports no other modules and exports empty?, push, pop, print, with the-stack being kept as a private variable. Loading the module Queue specified as:

```
(module Queue ()
  (private the-queue '())
  (public empty? <code for empty?>)
  (public enque <code for enque>)
  (public deque <code for deque>))
```

causes similer effects. Figure 1 shows IMP's user interface after loading Queue and Stack.

In addition to displaying the names of modules and their imports, the small boxes in Figure 1 also serve as control panels for the corresponding read-eval-print windows. Clicking the box causes the read-eval-print window to disappear, and clicking it again causes it to reappear. The displayed "()" in the lower half of the box indicates the module imports no other modules. Multiple read-eval-print windows allow the user to test the implementations of Queue and Stack concurrently as if he or she has two Scheme systems running in two independent terminals.

Suppose the Main program module of sample is defined as:

```
(module Main (Queue Stack)
    <other code of Main>)
```

then loading Main into the system makes public definitions in both Queue and Stack available to the read-eval-print window associated with the Main module. However, the name empty? is used in both Queue and Stack. Additional mechanisms are needed to resolve this name clashing problem among imported modules.

The easiest solution is to use "qualified" names such as Stack.empty? or Queue.enque for every imported identifier. However, we consider such a solution as "unfriendly" since it requires extra key strokes for imported identifiers. Another solution is to require that the user explicitly rename identifiers that are involved in name clashes. However, in some applications name clashes can be useful in shadowing undesired bindings. Simply renaming every identifier is not sufficient for those applications.

To solve this problem, we choose to employ an interactive renaming/shadowing mechanism. For example, at the time when module Main is loaded, a dialogue is displayed:

```
Name clashes: empty? exported from both: Stack and Queue
   1. Choose empty? from:
   2. Rename empty? of Stack as:
      Rename empty? of Queue as:
```

The user can choose to answer either the first question or the second question but not both. Answering the first question with Stack shadows the empty? defined in the queue. Answering the second question with s.empty? and q.empty? makes the empty? defined in both Queue and Stack available with local names s.empty? and q.empty?.

As another example, let us consider the situation when the user discovers the need to add a print procedure to Queue and export it dynamically to Main. The user can modify the file associated with Queue and then load the modified module into the system, or he or she may choose to enter the code directly into the read-eval-print loop. Suppose the user did the latter by typing:

```
(public print <code of print>)
```

in the Queue module. As soon as the evaluation of (public print ...) is complete, IMP attempts to export the procedure print to Main and every other module that imports Queue. Since Main has already imported print from Stack, a dialogue window similer in nature to the dialogue presented previously would appear on the screen for additional information to resolve name clashes. In addition to its flexibility, interactive renaming/shadowing also provides a good warning mechanism for "surprising" name clashes.

To allow the user to review the set of imported bindings, the Uses List button of a module causes the display of a list which associates the names of local identifiers and their sources of imports. Figure 2 is the *uses-list* for Main after shadowing Stack's print and renaming Queue's empty? to q.empty? and Stack's empty? to s.empty?. Uses lists are editable.

[s.empty? Stack.empty?] [push Stack.push] [pop Stack.pop] [q.empty? Queue.empty?]
[enque Queue.enque] [deque Queue.deque] [print Queue.print]

Figure 2: Uses list

At some point in the programming process the user may wish to modify the push procedure for debugging purposes. In traditional modular programming systems this would require the Stack module be recompiled and relinked with other modules before testing can begin. In IMP, such recompilation and relinking is kept to a minimum and is hidden from the user. The user is able to use the newly defined push procedure in the Main module after it is changed.

In addition to examples described above, many other scenarios of user interaction are also possible. For example, the user may choose to "delete" a module from a project or import additional modules in an already loaded module. One challenge in the design of IMP is to maintain the consistency of the system and to choose the most meaningful strategies for handling all possible user interactions.

3. Module Environments

In Scheme, as in many other programming languages, the set of all possible identifiers is treated as a flat set. For example,

Stack Stack.push Stack.pop

are treated as three distinct identifiers even though they are related in an obvious manner. The design of module environments attempts to use this natural source of structure for name space management.

The set of all possible identifiers in Scheme can be effectively classified into two disjoint sets. One set is all possible identifiers for naming module environments. Another set is all identifiers other than that for naming module environments. We choose to define the subset of identifiers for naming module environments as identifiers formed without the "." character and the subset of any other identifiers as identifiers containing at least one "." character. To ensure that the classification of identifiers is properly observed and to save the user from typing identifier names such as Stack.push, the system reader converts every free identifier id encountered in the module associated with a module environment E into E.id. Note that in our definitions, "." and ".push" are possible names for identifiers; however, in practice, they can never be formed in IMP.

A module environment named E denotes a set of bindings or (name . value) pairs where value is the value of the global identifier E.name. In IMP, a module environment is created by

(module *mod* (*im-mod* ...) *exp* ...)

where *mod* is the name of the module environment, *im-mod* ... are the names of imported module environments, and *exp* ... are expressions in the module. The expressions in the module E are

evaluated in an environment (in traditional sense) composed of a fresh copy of the module's top level bindings (a module E has its own E.+, E.−, ... E.car, E.cdr ...etc.) and bindings imported from other module environments. Importing a module environment with bindings ([n1 v1] [n2 v2] ...) to a module E binds global identifers E.n1 to v1, and E.n2 to v2 ...(assuming no renaming is necessary).

New bindings are added to a module environment with the public syntactic form:

(public *id* *exp*)

A public definition (public a *exp*) occurring in a module E defines a top level binding E.a and extends the module environment E with the pair (a . <E.a's value>). Conversely the private syntactic form:

(private *id* *exp*)

establishes a top level binding without extending the module environment.

Only one evaluation environment exists for every module in the system. E.+, E.− ...etc. are defined globally. The reason they are only "local" to E is that no module except E can form a name such as E.+ to access E's global bindings. Separate name spaces for modules is effectly achieved.

On the surface, it appears that module environments only allows importing values. However, the box [Dyb87b,Dyb87a,KKR+86] object can be used to implement other import semantics that require locations.

Module environments are not first class because they can be used only as *im-mod* in the module special form. However, the properties of module environments are sufficient to support interactive modular programming. In addition, with information stored in the *uses-list*, programs using module environments can be compiled efficiently.

4. Recursive Modules

Modules that import directly or indirectly from each other are resursively defined. IMP supports recursive modules as long as there are no cyclic initialization dependencies of exported objects among the modules. As an example,

```
(module A (B)          (module B (A)
  (public x y))          (public y x))
```

are two resursively defined modules with a cyclic initialization dependency.

Two approaches are possible to load resursively defined modules. One approach is named *block-on-unbound*. The other is named *multi-pass-loader*. Block-on-unbound is suitable for loading modules interactively into the IMP development environment. Multi-pass-loader is suitable for loading a project as a collection of modules into the system.

Consider the following example where modules A and B export to each other:

```
(module A (B)          (module B (A)
  (public m 1))          (public p 3))
  (public n (+ 2 p)))    (public q (+ m n)))
```

Let's assume A is loaded first into IMP. Since no binding is yet available in the module environment B, IMP proceeds with loading the body of A. Without special treatment, the evaluation of (public n

(+ 2 p)) will cause an error, since, A.p is an unbound identifier. However, with block-on-unbound, "process" A is blocked on identifier A.p. Suppose the user chooses to load B after seeing that A is blocked, then after B.p is defined and exported to define A.p, process A is unblocked. Since A.n is defined and exported to define B.n before B.n is used, no further blocking occurs.

The multi-pass-loader works as follows: All expressions to be loaded are stored in a pool, and the multi-pass-loader traveres the pool attempting to load expressions. If an expression can be evaluated without causing an "undound identifier" error, then the expression is loaded and removed from the pool. The multi-pass-loader continues to walk through the pool until the pool becomes empty or no progress can be made. The multi-pass-loader is capable of detecting the existence of cyclic initialization dependencies. However, using block-on-unbound to load modules with cyclic initialization dependencies results in deadlock.

The performance of loading can be improved by requiring the user to specify the dependency relations among modules. Special "loader directories" can be used to trigger the multi-pass-loader to load recursive modules. The dependency relations among modules can also be used to organize the structure of the documentation of the program.

5. Implementation

From the implementation's point of view, IMP is a special purpose operating system which is designed to run multiple Scheme processes concurrently. These Scheme processes have their own "address spaces" (the scope of modules), their own I/O devices (keyboard, mouse, and window), and their own "command" interpreters (the read-eval-print loop). The IMP user issues "system calls" by selecting menu items and clicking command buttons. The implementation of IMP is therefore responsible for providing the following:

- the user interface,
- process scheduling and I/O handling, and
- intermodule linkage, *i.e.*, shared address space management.

5.1 User Interface

The IMP user interface is implemented using the X window system [SG86], the object-oriented X toolkit [SA88], and user interface components (scroll bars, buttons, etc.) defined in the Athena widgets set [MA88,SW]. We have developed a set of Scheme procedures to access these tools.

The base system of X can be called from Scheme with the use of the foreign function interface. However, only low level routines are available at this level. For most applications, it is desirable to use high level user interface tools defined in the X toolkit and widget sets.

X window applications are *event-driven* programs. Any X application always has an *event loop*. This loop receives user actions as events and dispatches them to event handlers for further processing. In addition to defining an object oriented programming style for writing and extending widgets in C, one of the most important features of the X toolkit is that it frees programmers from doing direct event handling using an event loop. Instead, events and their handlers are registered internally and the handlers are automatically invoked by the X toolkit when the corresponding

events occur. This presents a problem in writing callback routines in Scheme, since X toolkit routines (written in C) can not invoke Scheme functions directly when an event occurs.

We have found a method to get around this problem. This method uses Scheme to implement its own callback dispatcher which communicates with the X toolkit using an index into a callback vector maintained in Scheme. The following is a brief description of the method:

- A vector is used to store the Scheme callback functions and client_data. The client_data is passed to the callback when it is invoked.
- The index to the vector is used to communicate between Scheme and C.
- A "standard_callback" function written in C is used to register widgets (which will be the source of events) and the indexes into the X toolkit. If a callback event associated with a widget occurs, a flag is set to the value of the index (which is previously registered and passed back from X toolkit). Other information (widget, call_data) passed from the X toolkit are also stored for Scheme to retrieve.
- The Scheme callback dispatcher polls the flag during every iteration of the event dispatch loop and calls the associated Scheme callback function stored in the vector.

In addition, we have also used similer approaches to implement translation manager (associates key event to actions) and other event handling mechanisms.

5.2 Process Scheduling and I/O Handling

Scheme treats continuations as first class objects. First class continuations can be used to implement engines [HF87,DH89], a high-level abstraction for timed preemption. Continuations and engines play a major role in implementing IMP's scheduler.

The simple read-eval-print procedure in Figure 3 illustrates the uses of continuations. The read-eval-print procedure uses call/cc twice to capture two continuations:

- *quit* accepts the passed value end-of-input and returns it as the result of applying read-eval-print.
- k represents what to do next after the (call/cc (lambda (k) ...)) expression, that is, executing (print (format "~ s") prompt) then (loop (read)). Note that k "remembers" the prompt passed to the read-eval-print procedure. If end is read, k is created and saved in the ready-queue. The saved k can be called later to continue the read-eval-print loop. (See the right column of Figure 3.)

The continuation saved in the ready-queue is almost a process except that it only relinquishes control of the "CPU" when end is read. In order to support multiple read-eval-print processes, some additional mechanism is needed to prevent a process from occupying the Scheme system indefinitely.

Engines are a high-level abstraction for timed preemption [HF87], providing the means for a computation to be interrupted if it does not complete in a specified amount of time. Engines can be implemented in terms of first-class continuations and timer interrupts [DH89]. In IMP, processes and their scheduler are implemented with engines. Figure 4 is an example illustrating how we use engines to implement a round-robin scheduler.

```
(define ready-queue (make-queue))
(define read-eval-print
  (lambda (prompt)
    (call/cc (lambda (quit)
      (printf (format "~s " prompt))
      (let loop ([obj (read)])
        (if (eq? 'end obj)
          (call/cc (lambda (k)
            (ready-queue 'enque k)
            (quit 'end-of-input)))
          (begin
            (write (eval obj))
            (newline)))
        (printf (format "~s " prompt))
        (loop (read)))))))
```

```
> (read-eval-print ':)
(read-eval-print ':)
: (+ 1 2)
3
: end
end-of-input
> ((ready-queue 'deque) 'ignored)
: (list 'a 'b)
(a b)
: (let loop () (loop))
```

Figure 3: Implementing read-eval-print processes with continuations

The make-process procedure takes a prompt as its argument, creates an engine using the make-engine procedure, and puts the new engine in the ready-queue. The argument to make-engine is a *thunk* representing the computation to be performed when the engine is invoked. In our example, the computation is the read-eval-print loop.

The scheduler procedure cycles through the ready-queue and executes each engine in a round-robin fashion. An engine starts executing when it receives a positive integer (ticks) representing the amount of "fuel" the engine can consume, a *complete* procedure that specifies what to do if the computation finishes, and an *expire* procedure that specifies what to do if the fuel runs out before the computation finishes. The *complete* procedure is a procedure of two arguments that expects to receive the result of the engine computation and the amount of fuel remaining. The *expire* procedure is a procedure of one argument that expects to receive a new engine capable of continuing the unfinished computation.

In our example, each engine's computation is an infinite loop. However, an engine should be stopped if no expression is available for the engine to evaluate. engine-return serves this purpose. engine-return causes the running engine to stop as if the computation had finished. Its argument is passed to the *complete* procedure along with the number of ticks remaining. In the make-process procedure, this argument is the message "stop" and a thunk capable of continuing the stopped computation. Upon receiving the arguments, the *complete* procedure creates a new engine from the thunk, puts the new engine on the tail of the ready-queue, and starts another engine from the head of the ready-queue.

Engines provide an excellent mechanism for implementing multi-tasking with processes that do not perform I/O. However, in our example, each process must read expressions to evaluate and print results. In traditional "single terminal" systems, this means that the whole Scheme system is stopped until an expression is read, defeating the possibility of implementing "real" time-sharing

```
(define make-process
  (lambda (prompt)
    (ready-queue 'enque
      (make-engine
        (lambda ()
          (printf (format "~ s " prompt))
          (let loop ([obj (read)])
            (if (eq? 'end obj)
              (call/cc
                (lambda (k)
                  (engine-return (list 'stop (lambda () (k 'ignored))))))
              (begin
                (write (eval obj))
                (newline)))
            (printf (format "~ s " prompt))
            (loop (read)))))))))
(define ticks 500)
(define scheduler
  (lambda ()
    (let loop ([running (ready-queue 'deque)])
      (running
      ticks
      (lambda (trap-msg ticks-remaining)
        (record-case trap-msg
          [stop (thunk) (ready-queue 'enque (make-engine thunk))]
          [else (error 'scheduler "bad trap message: ~ s" trap-msg)])
        (loop (ready-queue 'deque)))
      (lambda (new-engine)
        (ready-queue 'enque new-engine)
        (loop (ready-queue 'deque)))))))
```

Figure 4: Implementing read-eval-print processes with engines

systems. IMP solves this problem by providing each process its own *window ports* to read from and write to. A process is blocked if an unsatisfied read request is made from its *window input port.*

One major type of events are keyboard events. The keyboard is used to enter expressions and may also be used to provide input for expressions such as (read) or (read-char). If no expression is available in a window then the read-eval-print process associated with the window should be blocked. Similarly, if no input is available after a (read) or (read-char) request, the associated process should be blocked. A process blocked with no expression to evaluate becomes ready to execute after an *eval event*[1] is received. It remains ready until evaluation has completed or the evaluated expression has issueed another read request to the *window input port* of the process. In order to handle unsatisfied read requests, keyboard events must be used to signal the blocking and unblocking of processes from their *window input ports.* Each read-eval-print process also has a *window output port.* Unlike read requests, write requests can proceed without blocking. They can be handled outside the scheduler. The structure of IMP's scheduler is thus:

```
(let loop ()
  (obtain-x-event)
  <handle keyboard event for window input ports>
  (dispatch-x-event)
  <run a ready process>
  (loop))
```

IMP's scheduler needs to schedule both X events and Scheme processes. The number of ticks given to each process must be small enough for reasonable response time. However, there is a trade-off between fast "X response" and fast "Scheme response" since a smaller number of ticks means more context switches, resulting in slower execution of user programs. Fortunately, fully developed programs can run independently without the context switch overhead, so this cost is incurred only during interactive development.

5.3 Inter Module Linkage

A few alternatives are available to implement importing and exporting operations among modules [BH], including *import-by-value*, which imports values, and *import-by-reference*, which imports locations. The problem of import-by-value for interactive modular programming is that it is inefficient to support automatic import or export. That is, in order to "spread" the effect of interactively changing the value of an exported object, every importing module must be relinked or reinitialized in order to receive the new value. Import-by-reference avoids the need for relinking. However, the problem with import-by-reference is that too much control is given up by the exporting module: any importing module can change the value of an imported object with the result seen by both the exporting module and other importing modules. *Import-from-owner* is a variant of import-by-reference in which locations are imported, however, importers are prohibited from changing the value of imported objects. The remainder of this section discusses the implementation of import-from-owner. The implementation uses *boxes* to export "locations" [Dyb87b,Dyb87a,KKR+86].

[1] Eval event can be generated in one of two ways: 1. Click the Eval button. 2. Press the return key following a "parentheses balenced" expression.

Assuming module O is the exporting module with its module environment initialized as the null list. A public identifier *id* defined in O is translated as:

$$\begin{array}{rl} \text{(public } id\ exp\text{)} \Rightarrow & \text{(begin} \\ & \quad \text{(set! O.}id\text{ (box } exp\text{))} \\ & \quad \text{(set! O (cons } id\text{ O)))} \end{array}$$

The translation defines a top level binding[2] O.*id* and extends the module environment O with the name of the exported identifier[3]. Since *O.id* is bound to the value of *exp* through an extra level of indirection, accessing or changing the "value" of *id* in O requires an additional translation:

$$\begin{array}{rl} id & \Rightarrow \text{ (unbox O.}id\text{)} \\ \text{(set! } id\ exp\text{)} & \Rightarrow \text{ (set-box! O.}id\ exp\text{)} \end{array}$$

Suppose EXP is an expression occurring at the top level of O, then the previous two rules apply to those *id*s that occur free in EXP only. The following example, which assumes that a is defined as a public identifier in O, illustrates this point:

$$\text{((lambda (a) a) a)} \Rightarrow \text{((lambda (a) a) (unbox O.a))}$$

Note that "O." need not precede the bound variables. The translation from set! to set-box! broadcasts the side effect to every module that imports id.

Suppose module U imports (id_1 id_2 ...) from module O. Then:

$$\begin{array}{l} \text{(set! U.}id_1\text{ O.}id_1\text{)} \\ \text{(set! U.}id_2\text{ O.}id_2\text{)} \\ \ldots \end{array}$$

is performed. Imported identifiers are unboxed to access their values:

$$id \Rightarrow \text{ (unbox U.}id\text{)}$$

In order to localize the side effect of the "set!" operation on an imported identifier, a new box is allocated:

$$\text{(set! } id\ exp\text{)} \Rightarrow \text{ (set! U.}id\text{ (box } exp\text{))}$$

Of course, the previous two rules apply only to those *id*s that occur free in expressions occurring at the top level of U.

Many other import/export mechanisms are possible, such as restricting exported objects to be procedures or supporting *range-check* on the arguments of exported procedures [BH]. We have not yet decided which particular import/export semantics is most appropriate for our module system.

[2] We assume every identifier is initially bound to a location.

[3] The implementation need not represent the second class environment O as ([id value] ...). The value can be obtained by evaluating O.id.

6. Conclusion

We have implemented a prototype system for interactive modular programming. However, many problems remain to be solved. One of them is to support import and export of macros. The other one is to extend our module system to support object oriented programming. We believe interactive modular programming is a promising research area which may completely change the traditional method for interacting with Lisp systems.

References

[AS84] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1984.

[BH] C. Bruggeman and R. Hieb. Private communication.

[DH89] R. K. Dybvig and R. Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[Dyb87a] R. Kent. Dybvig. *Three Implementation Models for Scheme.* PhD thesis, University of North Carolina, Chapel Hill, 1987.

[Dyb87b] R.K. Dybvig. *The Scheme Programming Language.* Prentice-Hall, 1987.

[HF87] C. T. Haynes and D. P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):109–121, 1987.

[Jr84] G. L. Steele Jr. *Common Lisp.* Digital Press, 1984.

[KKR+86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: An optimizing compiler for Scheme. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, 1986.

[Knu84] D.E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[MA88] Joel McCormack and Paul Asente. Using the X toolkit or how to write a widget. In *Proceedings of the Summer, 1988 USENIX Conference*, pages 1–13, 1988.

[RC86] J. Rees and W. (Eds.) Clinger. *Revised*^{3} report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[SA88] Ralph R. Swick and Mark S. Ackerman. The X toolkit: More bricks for building user interfaces. In *in Proceedings of the Winter, 1988 USENIX Conference*, pages 221–233, 1988.

[SG86] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[SW] R. Swick and T. Weissman. X Toolkit Athena Widgets - C Language Interface. Distributed with X Version 11, Release 3.

[Tun89] S.H. Tung. A structured method for literate programming. *Structured programming*, 10(2):113–120, 1989.