Dynamic Modules in Higher-Order Languages

Suresh Jagannathan
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
suresh@research.nj.nec.com

Abstract

Providing programmers the ability to construct meaningful abstractions to help manage complexity is a serious language design issue. Many languages define a module system that can be used to specify distinct namespaces, and build user-defined data abstractions; however, few languages support *dynamic modules*, *i.e.,* modules which are true first-class objects. In this paper, we define a module semantics for a dialect of Scheme called Rascal. Modules are defined in terms of *reified environments*, and are first-class objects which may be dynamically created, freely assigned, used as arguments to procedures, etc.. If defined naively, however, implementing modules using environments can entail the capture of unwanted bindings, leading to potentially severe violations of lexical abstraction and locality.

We address these concerns by giving users great flexibility to manipulate environments, and to constrain the extent and scope of the environment reification process. We argue that the techniques and operators developed define a cohesive and semantically sound basis for building large-scale modular systems in dynamically-typed higher-order languages. Using meta-level environments for implementing modules and complex data abstractions requires few syntactic extensions to the base language, is amenable to aggressive compile-time analysis, and permits the description of a number of modularity issues within a well-defined formal framework.

1 Introduction

Providing programmers the ability to construct meaningful abstractions to help manage complexity is a serious language design issue. Many languages define a module system that can be used to specify distinct namespaces, and build user-defined data abstractions; however, few languages support *dynamic modules*, *i.e.,* modules which are true first-class objects. In this paper, we define a module semantics for a dialect of Scheme [4] called Rascal. Modules are defined in terms of *reified environments*, and are first-class objects which may be dynamically created, freely assigned, used as arguments to procedures, etc.. If defined naively, however, implementing modules using environments can entail the capture of unwanted bindings, leading to potentially severe violations of lexical abstraction and locality. Languages that permit programmer manipulation of meta-level environments (*e.g.,* [6, 25]) often are unable to preserve or apply useful program transformations which rely on static scoping (*e.g.,* α-conversion). This is because proposals that have heretofore been suggested for manipulating environments as data objects provide no mechanism to constrain the effect (or extent) of the reification process.

Scheme [4] is a higher-order dynamically-typed language that does not support a module facility; its utility in building large-scale systems is thus severely undermined. There have been several proposals, however, to add a module facility to Scheme (*e.g.,* [12, 5]), and many dialects of Scheme implement some form of modules [20, 12]. Several of these implementations constrain modules to be statically defined [26]; thus, modules serve primarily as a mechanism for separate compilation, and are cumbersome to use as a data abstraction facility. Other implementations permit module-like structures to be dynamically created, but do not integrate modules cleanly with the rest of the language making their liberal use expensive or difficult to reason about [20]. None of these proposals define a formal semantic treatment of modules, and all these systems incorporate modules as a new primitive datatype orthogonal to other kinds of modularity structures such as records or objects. Languages such as Modula-3 [3], or Ada [24] also provide support for modules; however, the semantics of modules in these systems requires them to be statically defined at the top-level and necessitates non-trivial syntactic and semantic extensions to express parameterized

To appear in the Proceedings of the 1994 IEEE International Conference on Computer Languages.

modules. In the case of languages such as Haskell [10] or SML [19, 18], the semantics and implementation of modules is also closely tied to a static type system.

We address these concerns by giving users great flexibility to manipulate environments, and to constrain the extent and scope of the environment reification process. We argue that the techniques and operators developed define a cohesive and semantically sound basis for building large-scale modular systems in dynamically-typed higher-order languages. Using meta-level environments to implement modules and complex data abstractions requires few syntactic extensions to the base language, is amenable to aggressive compile-time analysis, and permits the description of a number of diverse modularity issues within a well-defined formal framework.

2 Motivation

A module is commonly understood to define a separate namespace. Modules may control visibility of bindings defined by its namespace by specifying a *signature* that defines the set of names (potentially along with their types) that are visible to the outside. The set of bindings associated with a given signature defines the module *interface*. Depending on the semantics desired, a module could be instantiated multiple times (*e.g.,* as in Scheme-48 [12] or ML [18]) or only once (*e.g.,* as in Ada [24] or Clu [17]).

In high-level programming languages such as Scheme [4], ML [19] or Haskell [10], the exclusive use of lexical binding and the absence of abstractions to construct and manipulate namespaces necessitates the inclusion of a module facility as part of the language kernel. This is because lexical binding does not provide any mechanism for the selective import (and export) of bindings to (and from) different lexical contexts. A module system is an important example where such functionality is required.

An alternative approach to implementing modules (and related structures) is to implement environments as a basic data abstraction. Environments define finite functions that map names to values. Using environments to implement module-like structures is attractive because namespace related abstractions have a formal semantic definition or denotation closely related to their implementation. Just as significantly, other kinds of namespace management structures (such as records, objects, classes, etc.) which have similar denotations but which are used in different application contexts can be also implemented using the same framework. First-class environments can easily express dynamic modules; dynamically instantiated modules can be parameterized and generated in a number of interesting and important ways.

To make environments tractable and useful, however, it is important to constrain their effects. Failure to do so might make it difficult to optimize (and reason about) programs. Since environments deal with names at a semantic level, program transformations (such as α-conversion) that rely on the syntactic property of names to effect optimizations may no longer be valid in the presence of a language that permits unconstrained manipulation of reified environment objects.

In this paper, we define a programming language based on a model in which the effects of first-class environments are suitably constrained not to violate useful program transformations and optimizations that rely on lexical scoping. The model views a program's namespace as consisting of two two distinct component. The first component consists exclusively of *lexical* bindings (*i.e.,* bindings introduced by λ-abstraction and application); the second consists exclusively of *public* bindings (*i.e.,* bindings that are captured and exported outside their defining lexical scope). Environment operations available to the user only deal with public bindings; lexical bindings are constructed and manipulated in the usual way via λ-abstraction. Lexical environments only contain lexical bindings; public environments only contain public bindings.

Public environments are built by reifying pieces of a lexical environment. By carefully constructing the operations allowed on such environments, it is possible to retain the program reasoning capabilities afforded by lexical scoping, while also gaining expressive power through the use of bindings constructed and shared across disjoint lexical contexts. We argue that the ability to realize such sharing is the central issue in the design of a useful module system.

The rest of the paper develops the model and the language and highlights its utility in the context of a module system. The next section describes the operations allowed on reified environments; Section 4 gives a formal semantics for the language. Section 5 constructs a module system based on public environments and bindings. Section 6 gives comparison to related work.

3 Public Environments

There are six basic operations over public bindings and environments that we consider in this paper:

1. *Copy* a binding in the lexical environment into the public environment.

2. *Use* the binding value of a public binding.

3. *Curtail* or remove bindings from a public environment.

4. *Reify* the bindings in a public environment into a data object.

5. *Reflect* a data object representation of a reified environment into a public environment ρ. The bindings defined by ρ supersede public bindings in the current public environment.

6. *Mutate* a binding in a public environment.

There are seven operations provided in Rascal that are used to implement the operations found in the abstract model:

- `(make-public` *(vars) body*`)`. **[special form]**. Given a list of lexical variables L and an expression E, `make-public` evaluates E in the context of ρ_λ, and a new public environment defined to be the join of ρ_κ and the lexical bindings of all variables in L.

- `(reify)`. [**procedure**]. `Reify` returns the current public environment as an object.

- `(barrier` *e*`)`. **[special form]**. The `barrier` special form "hides" all public bindings in the current public environment within the dynamic extent of its argument expression.

- `(reflect` *env body*`)`. **[special form]**. Given an environment *env*, `reflect` composes the public bindings found in *env* with the current public environment to yield a new public environment ρ and evaluates *body* in the context of ρ.

- `(use` *var*`)`. **[special form]**. `(Use x)` returns the binding value of `x` in the current public environment. Note that `(make-public (x) (use x))` $\equiv$ `x`. Moreover,

  `(reflect (barrier (reify))` *exp*`)` $\equiv$ *exp*,

  `(reflect (reify) (use x))` $\equiv$ `(use x)`,

  and

  `(barrier (reflect` N `(reify))` $\equiv$
    `(barrier` N`)`

- `(set-public!` *var e*`)`. **[special form]**. Bindings in a public environment can be mutated via this operator. Thus, public bindings can be mutated outside the environment in which they were originally captured and defined.

Note that these operations specialize functionality that may be available in a more general framework. For example, a programming language that supports extensible, first-class environments, might choose to provide procedures that reify not only bindings, but names as well. Consider a `lookup` procedure that takes an environment *env* and a name *var* as its arguments, and returns the value of *var* in *env*. Because it is implemented as an ordinary procedure, `lookup` must treat *var* as any expression that yields a reified representation of a variable (*e.g.,* in terms of a symbol or string).

Rascal does not permit such functionality. By allowing reification only on bindings, the set of public names accessed and exported from a lexical environment is statically known. Thus, the environment constructed via evaluation of a `make-public` form consists of statically known bindings, and the names referenced from a public environment via `use` and `reflect` are also statically known – `use` is a special-form that requires its argument to be a name, and not a general Rascal expression.

The tradeoffs between a fully general environment abstraction that supports name reification (in terms of operations such as `lookup` and `eval`), and the more constrained operations provided in Rascal is not merely an efficiency vs. expressivity one. Although the environment abstractions provided in Rascal cannot be used to implement the `lookup` procedure described above, the separation of lexical and public environments isolates and identifies uses and effects of reified environments within a program. This capability gives programmers the ability to reason about code in terms of its lexical and non-lexical components; such reasoning capability is difficult in the presence of unconstrained reified environments.

4 Formal Semantics

We give a formal semantics for Rascal in Figs. 2, 3, and 4. The semantics is defined for a subset of the language; the (lifted) value domain is restricted to integers, Booleans, symbols, locations, namespaces, and procedures.

The primitive store, $\sigma_\perp$, maps to *undef*. $\sigma[l \mapsto v]$ defines an extension of store σ in which location l is bound to v. $\sigma_1[\sigma_2]$ denotes functional composition of stores σ_1 and σ_2. We define *StoreDomain*(σ) to be the set of locations in σ that are not *undef*. We also define an auxiliary procedure New where $New(\sigma)$ returns a pair, $\langle l, \sigma' \rangle$ where $\sigma = \sigma'$ for all $l' \neq l$, $l \notin$ *StoreDomain*(σ), and $l \in$ *StoreDomain*(σ').

The primitive environment, $\rho_\perp$, maps to *unbound*. $\rho[x \mapsto v]$ defines an extension of environment ρ in which variable

x is bound to v. $\rho_1[\rho_2]$ denotes functional composition of environments ρ_1 and ρ_2. We define *EnvDomain*(ρ) to be the set of variables in ρ that are not *unbound*. ρ_λ denotes lexical environments, and ρ_κ denotes public environments.

The semantics is given in continuation-passing style [14, 22], and models evaluation in terms of two environments. The first maps λ-bound variables, and the second maps public variables.

5 Records and Modules

5.1 Records

As a first example, consider the definition of a simple record structure. Rascal does not provide a primitive record constructor, but nonetheless permits the dynamic construction of namespaces which may be used to express the functionality of records. Records are implemented in terms of reified static environments; record templates are implemented as procedures that return a new namespace when applied.

The expression:

```
(let ((x 1)
      (y 'bar)
      (z (list 1 2 3)))
  (barrier (make-public (x y z)
             (reify))))
```

returns a namespace object with bindings for `x,y` and `z`. The `reify` operation used in the above expression returns the bindings of three λ-bound variables in the procedural environment within which it is evaluated; the barrier preceding the `let` form ensures that other public bindings found in the current public environment are not captured. The object returned can be freely bound, passed as an argument to (or returned as a result from) other procedures. If this object is called `R`, the expression,

```
(barrier (reflect R (use x)))
```

returns `1`. The above expression acts as a selector operation on `R`.

We can build a simple record constructor mechanism as follows:

```
(define (make-R x y z)
  (barrier (make-public (x y z) (reify))))
```

When applied, `make-R` returns a three element namespace binding its argument names to the values supplied.

5.2 Modules

In addition to providing record types, languages such as Ada [24], Clu [17], and Modula-2 [27] also define a module or data abstraction facility[1]. There are four salient features of modules common to all these languages:

1. modules are the fundamental unit of compilation and cannot be generated dynamically,
2. modules are the fundamental mechanism for building user-defined data abstractions,
3. modules are non first-class, and thus cannot be built into data structures, passed as arguments to ordinary procedures, etc.,
4. module creation and manipulation is syntactically distinct from record creation and manipulation.

A Simple Example A module in Rascal is an environment with procedure and data bindings; some of these bindings are visible to expressions outside the environment. They are created and manipulated in fundamentally the same ways that records are. To illustrate, consider the definition of a stack shown in Fig. 5. When applied, `stack` returns a namespace containing procedures `push` and `pop`. These procedures are closed over `max`, `s`, and `ptr`. Rascal's definition of a stack behaves as a module – it defines a data abstraction whose interface is specified by the public bindings it defines.

Unlike packages or clusters, however, a Rascal-style module is structurally no different from a record: it is defined in terms of a collection of bindings projected from a dynamically instantiated environment. In this respect, Rascal modules shares some functionality with modules defined in ML [18] – as in Rascal, there is no special runtime representation for modules in ML; modules are typically implemented in terms of ordinary records and function closures. However, a Rascal-style module is a first-class object; both the `stack` procedure as well its instances are treated no differently from any other object in the system.

The basic approach to building abstract data types in higher-order languages such as Scheme which do not have specialized constructs for this purpose is to use closures, and a dispatch procedure. The stack example can be rewritten in Scheme thus:

[1]Data abstractions are defined in Ada using packages, in Clu using clusters, and in Modula-2 using modules.

Syntactic Domains

c	$\in$ **Const**	Constants
x, y	$\in$ **Var**	Variables
e	$\in$ **Exp**	Expressions

Semantic Domains

$$
\begin{array}{ll}
v & \in Value = (Int + Bool + Symbol + Environment + Procedure)_\perp \\
l & \in Loc = Locations \\
k & \in Continuation = Value \rightarrow Store \rightarrow Value \\
r, \rho, \rho_\lambda, \rho_\kappa & \in Environment = Var \rightarrow Loc + unbound \\
f & \in Procedure = Value \rightarrow Continuation \rightarrow Store \rightarrow Value \\
\sigma & \in Store = Loc \rightarrow Value + undef \\
\psi & \in State = Environment \times Environment \times Store
\end{array}
$$

Valuation Function

$$
\mathcal{E} \qquad Exp \rightarrow Continuation \rightarrow State \rightarrow Value
$$

Figure 1: Syntactic and Semantic Domains

$$
\begin{array}{l}
\mathcal{E}\,[\![c]\!]\,k\,\langle \rho_\lambda,\,\rho_\kappa,\,\sigma\rangle = \\
\quad k(\,c,\,\sigma\,)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![x]\!]\,k\,\langle \rho_\lambda,\,\rho_\kappa,\,\sigma\rangle = \\
\quad k(\,\sigma(\,\rho_\lambda(\,x)))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![(\texttt{lambda}\;(x)\;e)]\!]\,k\,\langle \rho_\lambda,\,\rho_\kappa,\,\sigma\rangle = \\
\quad \textbf{let}\; \mathit{func} = \lambda\,\langle v,\,k',\,\sigma'\rangle.\,\textbf{let}\;\langle l,\,\sigma''\rangle = new(\,\sigma') \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\;\mathcal{E}\,[\![e]\!]\,k'\,\langle \rho_\lambda[x \mapsto l\,],\,\rho_\kappa,\,\sigma''[l \mapsto v\,]\rangle \\
\quad \textbf{in}\; k(\mathit{func},\,\sigma\,)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![(e_1\;e_2)]\!]\,k\,\langle \rho_\lambda,\,\rho_\kappa,\,\sigma\rangle = \\
\quad \mathcal{E}\,[\![e_1]\!]\;\lambda\,\langle f,\,\sigma'\rangle. \\
\qquad\qquad \mathcal{E}\,[\![e_2]\!]\;\lambda\,\langle v,\,\sigma''\rangle.f(\,v,\,k,\,\sigma'') \\
\qquad\qquad\qquad \langle \rho_\lambda,\,\rho_\kappa,\,\sigma'\rangle \\
\qquad\quad \langle \rho_\lambda,\,\rho_\kappa,\,\sigma\rangle
\end{array}
$$

Figure 2: Definitions for functional core.

$$
\begin{aligned}
&\mathcal{E}\,[\![(\texttt{make}-\texttt{public}\ (x)\ e)]\!]\,k\,\langle\rho_\lambda,\,\rho_\kappa,\,\sigma\rangle =\\
&\quad \textbf{let}\ \langle l,\,\sigma'\rangle = \mathit{new}(\,\sigma)\\
&\quad \textbf{in}\ \mathcal{E}\,[\![e]\!]\,k\,\langle\rho_\lambda,\,\rho_\kappa[x\ \mapsto l],\,\sigma'[l\ \mapsto \sigma(\,\rho_\lambda(\,x))\,]\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,[\![(\texttt{use}\ x)]\!]\,k\,\langle\rho_\lambda,\,\rho_\kappa,\,\sigma\rangle =\\
&\quad k(\,\sigma(\,\rho_\kappa(\,x)),\,\sigma\,)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,[\![(\texttt{barrier}\ e)]\!]\,k\,\langle\rho_\lambda,\,\rho_\kappa,\,\sigma\rangle =\\
&\quad \mathcal{E}\,[\![e]\!]\,k\,\langle\rho_\lambda,\,\rho_\perp,\,\sigma\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,[\![(\texttt{reify})]\!]\,k\,\langle\rho_\lambda,\,\rho_\kappa,\,\sigma\rangle =\\
&\quad \textbf{let}\ r\ =\ \rho_\perp[x_1\ \mapsto\ \rho_\kappa(\,x_1)\ldots x_n\ \mapsto\ \rho_\kappa(\,x_n)], \text{for } x_1,\ldots,x_n \in\ \mathit{EnvDomain}(\,\rho_\kappa)\\
&\quad \textbf{in}\ k(\,r,\,\sigma\,)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,[\![(\texttt{reflect}\ e_1\ e_2)]\!]\,k\,\langle\rho_\lambda,\,\rho_\kappa,\,\sigma\rangle =\\
&\quad \mathcal{E}\,[\![e_1]\!]\ \lambda\,\langle r,\,\sigma'\rangle.\\
&\qquad\qquad \mathcal{E}\,[\![e_2]\!]\,k\\
&\qquad\qquad\qquad \langle\rho_\lambda,\,\rho_\kappa[r],\,\sigma'\rangle\\
&\qquad \langle\rho_\lambda,\,\rho_\kappa,\,\sigma\rangle
\end{aligned}
$$

Figure 3: Definition of operations over public environments.

$$
\begin{aligned}
&\mathcal{E}\,[\![(\texttt{set!}\ x\ e)]\!]\,k\,\langle\rho_\lambda,\,\rho_\kappa,\,\sigma\rangle =\\
&\quad \mathcal{E}\,[\![e]\!]\ \lambda\,\langle v,\,\sigma'\rangle.\\
&\qquad\qquad k(\,v,\,\sigma[\rho_\lambda(\,x)\ \mapsto v\,]\,)\\
&\qquad \langle\rho_\lambda,\,\rho_\kappa,\,\sigma\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,[\![(\texttt{set}-\texttt{public!}\ x\ e)]\!]\,k\,\langle\rho_\lambda,\,\rho_\kappa,\,\sigma\rangle =\\
&\quad \mathcal{E}\,[\![e]\!]\ \lambda\,\langle v,\,\sigma'\rangle.\\
&\qquad\qquad k(\,v,\,\sigma[\rho_\kappa(\,x)\ \mapsto v\,]\,)\\
&\qquad \langle\rho_\lambda,\,\rho_\kappa,\,\sigma\rangle
\end{aligned}
$$

Figure 4: Definition of assignment.

```
(define (stack)
  (barrier (let* ((max 100)
                  (s (make-vector max))
                  (ptr -1)
                  (push
                    (lambda (x)
                      (cond ((= ptr max) (error "Stack Overflow"))
                            (else (set! ptr (+ ptr 1))
                                  (vector-set! s ptr x)))))
                  (pop
                    (lambda ()
                      (cond ((= ptr -1) (error "Stack underflow"))
                            (else (let ((result (vector-ref s ptr)))
                                    (set! ptr (- ptr 1))
                                    result))))))
              (make-public (push pop)
                (reify)))))
```

Figure 5: A stack abstraction in Rascal.

```
(define (make-stack)
  (let* ((max 100)
         (ptr 0)
         (stack (make-vector max))
         (push (lambda (x) code for push))
         (pop (lambda ()   code for pop)))
     (lambda (op)
       (cond ((eq? op 'push) push)
             ((eq? op 'pop)  pop)))))
```

To push an object O onto a stack instance S, we write:

((*S* 'push) *O*)

There is a significant difference between Rascal and Scheme in this exercise – the Scheme solution relies on labels and a dispatch procedure to associate an operation label with the operation itself; the Rascal solution does not. This dispatch procedure mimics an environment since it associates labels with values. The Rascal solution manipulates environments directly, and does not require the presence of an intermediate dispatch routine.

Building Interfaces The way stacks are defined and used in the above example is clearly too simplistic for programming in the large. For any module system to be useful it must permit programmers to:

- cleanly separate a module interface from its implementation,
- dynamically link new implementations of a module with new clients,
- incrementally define a module's exported procedures, and
- ensure that re-implementation of a module will not require re-implementation of clients using this module.

Unfortunately, the stack program shown above meets none of these requirements. To allow interfaces to be freely constructed, and to allow dynamic linking of module implementations with an interface specification, we need to provide a module system that handles several kinds of objects:

1. A client interface that specifies the bindings imported by a user from a module.
2. An implementation interface that contains bindings used by the implementation to test the specification of the implementation and to make it visible to clients.
3. An implementation that exports bindings to clients.

We now reconsider the stack example in light of these new requirements. In order to permit interfaces to be separated from implementations, we first define a client interface for stacks:

```
(define (stack-client-interface)
  (let ((create (lambda ()
          (error "undefined procedure")))
        (push (lambda (rep x)
          (error "undefined procedure")))
        (pop  (lambda (rep)
          (error "undefined procedure"))))
    (barrier (make-public (create push pop)
               (reify)))))
```

The interface is defined as a procedure which returns a namespace that contains bindings for procedures `create`, `push` and `pop`; these procedures are initially defined to return an error if applied.

A client who wishes to use a stack module imports the module using the syntactic form, `import`:

```
(import stack-interface E) ≡
   (let ((stack-interface
          (stack-client-interface)))
     (reflect stack-interface E))
```

Note that clients can be compiled with references to stack procedures even if a stack implementation has not yet been defined.

Having defined a client interface for stacks, we now define the implementation interface. Implementors of stacks use an interface that provides procedures for testing whether the implementation meets a desired specification, and for making the implementation available to the client. This interface is shown in Fig. 6.

The `test` procedure returns true if the implementation provides procedures named `create`, `push` and `pop`. The `install` procedure re-assigns the bindings defined by its interface argument to the values specified by the new implementation.

Finally, we define a stack installation routine. This procedure, given an implementation, tests the implementation based on the `test` procedure provided by the implementation interface, and if successful, installs the implementation. We show this routine in Fig. 7.

We install a stack module thus:

```
(let ((impl-interface
        (stack-impl-interface))
      (client-interface
        (stack-client-interface)))
  ((install-stack impl-interface)
    client-interface stack))
```

The object returned by evaluating this expression is a namespace containing the implementation of `push`, `pop` and `create`; these procedures are closed over a stack representation. Multiple instances of a stack can be created by calling the `create` procedure defined as part of the module interface.

Clients can refer to stacks even if the implementation is not available. Moreover, clients need not be recompiled even after an implementation is specified since the public bindings defined in the object returned by a call to `stack-interface` are modified directly by the installation procedure of an implementation. Implementations can be constructed without knowing the clients which will use it. Both the client and implementation interface are called at runtime; thus, this system permits modules to be dynamically constructed and linked.

Operations on environments and namespaces are used in two distinct ways in this example. Reification is used to access interface procedures found in an implementation. Reflection operations expose bindings to clients and implementors. When used by clients, a reflect operation imports bindings from the stack interface; when used by implementors, it exports procedures to clients (*e.g.,* as used in the body of `install`). Clients see the interface procedures provided by the implementation; implementations see the interface procedures provided by clients.

Rascal's treatment of modules differs in important ways from other module systems proposed for Scheme and related languages. Unlike modules in Curtis and Rauen's system [5], interfaces are not compile-time objects in Rascal. A module's interface can be generated dynamically, and can be incrementally defined. ML's module system uses functors[2] to implement interfaces; the tight-coupling of ML's module system with its type system distinguishes it from Rascal in obvious ways. Moreover, since functors are not first-class, dynamic installation of implementation interface procedures such as `test` and `install` is difficult to express. Our formulation shares much in common with Lee and Friedman's module description given in [16]. Their system also supports runtime linking. The differences between the two systems lie primarily in the way modules are realized. Rascal relies on constrained reification of environments to specify selective import and export of interface procedures; their approach relies on "quasi-static variables" to achieve a similar effect. We defer discussion of their proposal to Section 6.

The specification of a module in Rascal is closely correlated to the module's implementation. This is an important distinction between Rascal and other languages in which modules are defined as primitive objects. For example, to specify a stack module, one was required to the specify significant details of the module implementation. In

[2] A functor is a first-order procedure used to link modules together, and to enforce type constraints among modules that share bindings.

```
(define (stack-impl-interface)
  (let ((test (lambda (impl)
                (let ((create (lambda () 'create))
                      (push   (lambda () 'push))
                      (pop    (lambda () 'pop)))
                   (reflect (barrier (make-public (create push pop) (reify)))
                     (reflect impl
                       (and (not (eq? (use create) create))
                            (not (eq? (use push) push))
                            (not (eq? (use pop) pop))))))))
        (install (lambda (client-interface impl)
                   (reflect client-interface
                     (begin
                       (set-public! create
                                    (barrier (reflect impl (use create))))
                       (set-public! push
                                    (barrier (reflect impl (use push))))
                       (set-public! pop
                                    (barrier (reflect impl (use pop))))))))))
     (barrier (make-public (test install) (reify)))))
```

Figure 6: The implementation interface for stacks.

languages in which modules are defined as part of the language kernel, *e.g.,* Ada [24] or Modula-2 [27], programmers are free from thinking about the actual implementation of the module facility. In this sense, we should consider Rascal as a high-level implementation substrate for various modularity structures. However, when used in conjunction with expressive macro systems (*e.g.,* [13, 26]), much of the complexity in implementing modules could be alleviated. For example, we can define macros called `make-client-interface` and `make-impl-interface` that can be used to define stack clients and implementations:

```
(define (stack-client-interface)
  (make-client-interface (push pop create)))

(define (stack-impl-interface)
  (make-impl-interface (push pop create)))
```

The implementation of `make-client-interface` using a `define-syntax` macro specification form of the kind available in Scheme [4] could be given thus:

```
(define-syntax make-client-interface
  (syntax-rules ()
    ((make-client-interface  (p1 p2 ...))
       (let ((p1 (error
                   "undefined procedure"))
             (p2 (error
                   "undefined procedure"))
                 ...)
         (make-public (p1 p2 ...)
           (reify))))))
```

As another example, we can abstract details of an implementation interface using the macro shown in Fig. 8:

Similar syntactic forms can be built for abstracting other details of implementations, and installation procedures. Application programmers would then use these syntactic abstractions as the module implementation mechanism; the actual specification of the module that needs to be constructed would be defined using such abstractions. Language implementors would use environment operations in the ways described above to provide the feature implementation.

5.3 Modules as Structures

One obvious advantage of reified environments is that new interfaces can be constructed without requiring reimplementation or modification to the language kernel. To

```
(define (install-stack impl-interface)
  (barrier (reflect impl-interface
             (let ((test-proc (use test))
                   (install-proc (use install)))
               (lambda (client-interface impl)
                 (cond ((test-proc impl)
                        (install-proc client-interface impl)
                        'installed)
                       (else 'rejected)))))))

(define stack
  (let ((max 100)
        (passwd (list 'password)))
    (let ((push (lambda (rep x)
                  (barrier (reflect (rep passwd)
                             (cond ((= (use ptr) max)
                                    (error "Stack Overflow"))
                                   (else (set-public! ptr (+ (use ptr) 1))
                                         (vector-set (use s) (use ptr) x)))))))
          (pop (lambda (rep)
                  (barrier (reflect (rep passwd)
                             (cond ((= (use ptr) -1)
                                    (error "Stack Underflow"))
                                   (else (let ((result (vector-ref (use s) (use ptr))))
                                           (set-public! ptr (- (use ptr)))
                                           result)))))))
           (create (lambda ()
              (let ((impl (let ((s (make-vector max))
                                (ptr -1))
                            (barrier (make-public (s ptr) (reify))))))
                (lambda (pass)
                  (if (eq? pass passwd)
                      impl
                      (error "Can't access representation")))))))
        (make-public (push pop create)
          (reify)))))
```

Figure 7: `Install-stack` installs a new stack module provided the module meets the specifications dictated in the implementation interface. There are many possible implementations of stacks; the one shown above permits only `push` and `pop` to access components of the representation generated by `create`. The object returned by `create` is a procedure that takes a password (represented as a list) as an argument. It returns a namespace containing the representation (in this case a vector and a pointer defining the top of stack) *only if* its argument is `eq?` to the password associated with `stack`. Since only calls made by `push` and `pop` satisfy this constraint, no client will have access to a stack's internal representation.

```
(define-syntax make-impl-interface
  (syntax-rules ()
    ((make-impl-interface ((b1 b2 ...))
      (let ((test (lambda (impl)
                    (let ((b1 (lambda () b1))
                          (b2 (lambda () b2))
                             ...)
                      (barrier (reflect impl
                                 (and (not (eq? (use b1) b1))
                                      (not (eq? (use b2) b2))
                                          ...))))))
            (install (lambda (client-interface impl)
                       (reflect client-interface
                         (begin
                           (set-public! b1 (barrier (reflect impl (use b1))))
                           (set-public! b2 (barrier (reflect impl (use b2))))
                                ...)))))
        (barrier (make-public (test install) (reify))))))))
```

Figure 8: A syntactic abstraction for implementation interfaces.

illustrate, consider the following Scheme48 package definitions:

```
(define-signature scheme (export + * cons))
(define-signature foo (export a c cons))
(define-signature bar (export d))

(define-package ((foo foo-signature))
  (open scheme)
  (begin (define a 1)
         (define (b x) (+ a x))
         (define (c y) (* (b a) y))))

(define-package ((bar bar-signature))
  (open scheme foo)
  (begin (define d w) (+ a (c w))))
```

This configuration defines two structures. `Foo` is a structure that exports bindings for `a`, `c`, and `cons`; `bar` is a structure that exports a binding for `d`. Both `foo` and `bar` import bindings from the `scheme` structure. The binding for `cons` exported by `foo` is defined by `scheme`. The `define-signature` form serves as useful declaration indicating the bindings exported by a structure.

This module specification is very different from the specification used to implement stacks. As used in this style, there is no separation between a module client's and its implementation; a package's signature specifies the interface, and its implementation is provided directly within the package body. The expressions found within a "`begin`" are evaluated when the structure is instantiated; at structure instantiation time, all imported bindings are known. Thus, evaluation of forms within a package P can only take place when all packages it imports are known and instantiated; compilation of P's body is deferred until the bindings found in all packages P accesses are known.

A similar module structure can be expressed in Rascal (see Fig 9). In this implementation, bindings from imported packages specified as exportable in their signature are spliced into the package body as lexical bindings. Thus, the package specification for `bar` locally binds `+`, `*` and `cons` to their binding definition in a `scheme` package, and `a`, `c` and `cons` to their definition in an instance of `foo`. Unlike Scheme48's module specification, imported bindings must be explicitly recorded in the Rascal version. This is because any bindings exported from a lexical scope can only be accessed via operations on public environments. As was the case with the earlier examples, it is straightforward to construct syntactic abstractions that obviate the need for programmers to explicitly specify such bindings; information found in package signatures and `open` declarations provide the necessary details to construct such abstractions.

6 Comparison to Related Work

Pebble [2] and certain dialects of Scheme [1] are two languages that also support environment reification. Pebble is a kernel language designed to facilitate the construction

```
(define (foo scheme)
  (reflect scheme
    (let ((+     (use +))
          (*     (use *))
          (cons  (use cons)))
      (barrier
        (let ((a 1)
              (b (lambda (x) (+ a x)))
              (c (lambda (y) (* (b a) y))))
          (make-public a c cons)
            (reify))))

 (define (bar scheme foo)
   (reflect foo
     (reflect scheme
       (let ((+    (use +))
             (*    (use *))
             (a    (use a))
             (c    (use c))
             (cons (use cons)))
         (barrier
           (let ((d (lambda (w) (+ a (c w)))))
             (make-public (d)
               (reify))))))))
```

Figure 9: Implementing modules as structures.

and maintenance of large modular programs. It does so by providing a semantics for data types, abstract data types and modules within the framework of the second-order lambda calculus. Among its many interesting features (*e.g.,* dependent types, polymorphism, types as values, etc.), Pebble also treats bindings (and naming environments) as values.

A Pebble binding `x~3` binds the value of the name `x` to 3. The role of bindings in Pebble is to provide a basis for a semantics of modules given in terms of functions that map bindings to bindings. Pebble modules are treated as functions that map environments containing the bindings of module interfaces to environments containing the operations the module defines.

Rascal can also be used to implement Pebble-style modules. On the other hand, there is no notion of reification in Pebble; bindings found within an evaluation environment cannot be captured into a data structure. Moreover, the operations allowable on bindings are not the same as those allowable on records; to build complex sets of bindings one needs to use a pairing operation to build a tuple. Arbitrary elements of tuples cannot be projected, however, nor can tuples themselves be composed.

In certain dialects of Scheme [1], users are allowed to build customized environment structures through the `make-environment` special-form. `Make-environment` constructs an environment object, evaluates a sequence of expressions within this environment, and returns the new environment as its result. The `define` special form installs a binding within the environment in which it is evaluated; users access bindings defined in a particular environment by passing the environment as the second argument to the primitive `eval` operator. Thus, given an environment, `E`, containing definitions `x` and `y`, the expression: `(eval '(x y) E)` evaluates the application `(x y)` in the context of `E`. In addition, these dialects also provide a procedure ("`the-environment'` ") to capture the current environment.

The `make-environment` construct makes it possible for Scheme programmers to define local namespaces that encapsulate a related collection of data and procedures. The environment object yielded by `make-environment` or `the-environment` is a true first-class object – it can be passed as arguments to procedures or returned as a result. The only operation allowable on these environments, however, is `eval`. Unlike Rascal, one cannot restrict bindings captured by `the-environment` or annotate those variables that are valid targets for environment capture within a Scheme program.

The treatment of environments in the reflective languages 3-Lisp [21, 6] or Brown [7, 25] is similar to their treatment in Rascal in two important respects: environments can be reified into concrete structures (*e.g.,* procedures in Brown, and lists in 3-Lisp), and data structures can be reflected into environments. Neither Brown nor 3-Lisp, however, fully examined the implications of such reflective capabilities for building modular systems; thus, they do not provide barrier annotations, make no distinction between ordinary variables and those which can be captured by reification, and do not distinguish between lexical and public bindings. Because both Brown and 3-Lisp also address reification of continuations and stores, their emphasis is markedly different from ours. This results in a fairly sharp departure of our work from theirs both in the semantics of the reflective operators developed, and in the application domains addressed.

Smalltalk [9] and Self [23] are two late-binding languages that permit programmers to examine the context in which message evaluation occurs. The notion of a "context" in these languages is very similar to the notion of a reified environment in Rascal. In effect, contexts are first-class environments. However, neither Smalltalk nor Self provides mechanisms to constrain the bindings found within a context by restricting the extent and scope within which a context object is constructed.

Lee and Friedman propose an alternative treatment of names that manipulates variables rather than identifiers [16]. Their proposal consists of a new abstraction called a quasi-static variable that defines a mapping from a syntactic identifier to a semantic variable; identifiers in different scopes can share objects via an external name associated with a quasi-static variable. Their proposal shares many of the same goals as ours: enhancing modularity of higher-order languages such as Scheme by defining expressive (and safe) operations over names. In particular, their proposal retains full lexical scoping since syntactic names are never shadowed; a Rascal program is also lexically scoped in the absence of public variable annotations. The use of a different meta-object to implement various modularity devices, however, clearly separates their work from ours. Control over environments is obtained in Rascal using annotations to restrict name visibility and scope; no new semantic domains are constructed to support this functionality. Furthermore, quasi-static variables themselves provide no mechanism similar to barrier annotations, `reflect` or `reify`.

Symmetric Lisp [8, 11] is a parallel programming language whose fundamental data and program structure is an environment object. Although the language provides a unified treatment of program and data structures, Symmetric Lisp provides no reification operations on environments, nor does it provide customization of binding protocols.

Finally, Lamping [15] describes a modularity technique based on transparent parameterization of data objects, *i.e.,* objects which resemble procedural abstractions but which may be freely combined, and whose data parameters may be instantiated in arbitrary order. Although data parameters can be used to express a number of diverse modularity structures, there is no explicit notion of reflection or reification in the model presented; consequently, there is no direct analogue to the operations or annotations developed here.

Acknowledgments

Thanks to Richard Kelsey, Shinnder Lee and Henry Cejtin for many useful comments and suggestions.

References

[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.

[2] Robert Burstall and Butler Lampson. A Kernel Language for Modules and Abstract Data Types. In *International Symposium on Semantics of Data Types*. Springer-Verlag, 1984. Lecture Notes in Computer Science, Number 173.

[3] Luca Cardelli, Jim Dohahue, Mike Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 Type System. In 16^{th} *ACM Symposium on Principles of Programming Languages*, pages 202–213, 1989.

[4] William Clinger and Jonathan Rees, editors. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[5] Pavel Curtis and James Rauen. A Module System for Scheme. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 13–19, 1990.

[6] J. des Rivières and Brian Smith. The Implementation of Procedurally Reflective Languages. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 331–347, 1984.

[7] Daniel Friedman and Mitchell Wand. Reification: Reflection without Metaphysics. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 348–355, 1984.

[8] David Gelernter, Suresh Jagannathan, and Thomas London. Environments as First-Class Objects. In *14th ACM Symposium on Principle of Programming Languages Conference*, 1987.

[9] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Press, 1983.

[10] Paul Hudak *et.al.* Report on the Functional Programming Language Haskell, Version 1.2. *ACM SIGPLAN Notices*, May 1992.

[11] Suresh Jagannathan. A Programming Language Supporting First-Class, Parallel Environments. Technical Report LCS-TR 434, Massachusetts Institute of Technology, December 1988.

[12] January. Another Module System for Scheme, 1993. Scheme48 documentation.

[13] Eugene Kohlbecker and Mitch Wand. Macro-by-Example: Deriving Syntactic Transformations from their Specifications. In *14th ACM Symposium on Principles of Programming Languages*, pages 77–85, 1987.

[14] Guy L. Steele. Rabbit: A Compiler for Scheme. Master's thesis, Massachusetts Institute of Technology, 1978.

[15] John Lamping. A Unified System of Parameterization for Programming Languages. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 1–11, 1988.

[16] Shinn-Der Lee and Daniel Friedman. Quasi-Static Scoping: Sharing Variable Bindings Across Multiple Lexical Scopes. In *20th ACM Symposium on Principles of Programming Languages*, January 1993.

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

[18] David MacQueen. An Implementation of Standard ML Modules. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 212–223, 1988.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[20] Mark Sheldon and David Gifford. Static Dependent Types for First-Class Modules. In *ACM Conference on Lisp and Functional Programming*, pages 20–29, 1990.

[21] Brian Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, 1982.

[22] C. Stratchey and C.P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. Technical Report PRG-11, Programming Research Group, Oxford University, 1974.

[23] David Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle. Object, Message, and Performance: How They Co-Exist in Self. *IEEE Computer*, 25(10):53–65, October 1992.

[24] United States Dept. of Defense. *Reference Manual for the ADA Programming Language*, 1982.

[25] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–38, June 1988.

[26] William Clinger and Jonathan Rees. Macros that Work. In *18th ACM Symposium on Principles of Programming Languages*, pages 155–163, January 1993.

[27] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, 1985.