Metalevel Building Blocks for Modular Systems

SURESH JAGANNATHAN
NEC Research Institute

The formal definition of any namespace device found in a programming language can be given in terms of transformations on a semantic environment. It is worthwhile, therefore, to consider the implications of incorporating environments as bona fide data objects in a programming system.

Because of their expressive power, environments can be easily abused. Reifying an environment can entail the capture of unwanted bindings, leading to potentially severe violations of lexical abstraction and locality. Reflecting a data structure into an environment may cause useful program transformations which rely on static scoping (e.g., α-conversion) to be no longer applicable. Proposals that have heretofore been suggested for manipulating environments as data objects, however, provide no mechanism to constrain the effect or extent of the reification or reflection process.

In this article, we propose a treatment of environments and the mechanism by which they are reified and manipulated, that addresses these concerns. The language described below (Rascal) permits environments to be reified into data structures, and data structures to be reflected into environments, but gives users great flexibility to constrain the extent and scope of these processes. We argue that the techniques and operators developed define a cohesive basis for building large-scale modular systems using reflective programming techniques.

Categories and Subject Descriptors: D.3 [**Software**]: Programming Languages; D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages; extensible languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*denotational semantics*

General terms: Languages

Additional Key Words and Phrases: Environments, higher-order programming, modularity, reflection

1. INTRODUCTION

Modularity structures in modern programming languages typically address fairly narrow namespace management concerns. Consequently, most languages provide many disparate program and data structuring devices for managing, composing, and conserving bindings; these structures broadly share common functionality, but rarely interact with one another in significant or useful ways. For example, closures [Abelson and Sussman 1985; Steele Jr. and Sussman 1978], packages [U.S. Dept. of Defense 1982], and modules [Cardelli et al. 1989; Wirth 1985] are used for information hiding and data abstraction; objects [Ungar and Smith 1987] and classes [Dahl et al. 1970; Goldberg and Robson 1983] implement code reuse and

Author's address: NEC Research Institute, 4 Independence Way, Princeton, NJ 08540; email: suresh@research.nj.nec.com .

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
©1994 ACM 0164-0925/94/0500-0000 $03.50.

sharing; records define a related collection of bindings, etc.

Since the formal definition of all these structures is given in terms of operations on environments, it is worthwhile to consider the implications of incorporating environments as bona fide data objects in a programming system. Given such a capability, it is possible to capture the behavior of well-known modularity structures (and to synthesize new ones) simply by transliterating their semantic definitions into an equivalent one that operates over first-class environments.

Because of their expressive power, environments can be easily abused. Reifying an environment can entail the capture of unwanted bindings, leading to potentially severe violations of lexical abstraction and locality. Reflecting a data structure such as a record into an environment may cause useful program transformations which rely on static scoping (e.g., α-conversion) to be no longer applicable. Proposals that have heretofore been suggested for manipulating environments as data objects (e.g., Abelson and Sussman [1985], Friedman and Wand [1984], and des Rivières and Smith [1984]), however, provide no mechanism to constrain the effect or extent of the reification or reflection process.

In this article, we propose a treatment of environments, and the mechanism by which they are reified and manipulated, that addresses these concerns. The language described below (Rascal) permits semantic environments to be reified into data objects, and data objects to be transformed into environments, but gives users great flexibility to constrain the extent and scope of these processes. The nature of these constraints does not lead to loss of expressive power, however; to illustrate this point, we define a number of complex and superficially diverse modularity structures within a simple and unified framework. We consider the use of environment objects in defining lexical and dynamic binding protocols, class and prototype object systems, and customizable front-end evaluators. We argue that the techniques developed define a cohesive basis for building large-scale modular systems.

The article is structured as follows. The next section gives an overview of Rascal and discusses the semantics of environments and their reification. Section 5 describes the implications of our design on modularity and program building. Section 6 describes compile-time optimizations to reduce the cost of using first-class environments. Conclusions and comparison to related work are given in Section 7.

2. THE MODEL

In high-level programming languages such as Scheme [Clinger and Rees 1991], ML [Milner et al. 1990], or Haskell [Hudak et al. 1992], the meaning of a variable reference is based solely on the lexical context in which the variable is defined. Lexical binding is a simple rule that reduces implementation complexity and eases reasoning about the operational behavior of a program. By itself, however, lexical binding does not provide any mechanism for the selective import/export of bindings from/to different lexical contexts. Objects, modules, or incrementally constructed programs are several examples where such functionality is important. Lexically scoped languages are typically extended in significant, often ad hoc, ways to support such applications.

The formal meaning of a variable reference is typically defined via an abstract environment manipulated by an interpreter that defines a language's semantics. An environment is a map from names to values. In most languages, this abstract envi-

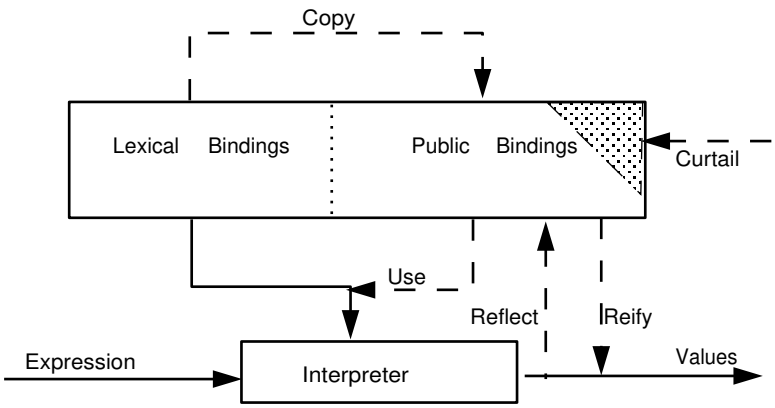

Fig. 1. A programming model that partitions, reflects, and reifies bindings

ronment is hidden from the programmer. Thus, it is often not possible to write expressions that directly access and manipulate environment information built during the evaluation of a program; in other words, most languages prohibit the construction of programmer-specified name lookup protocols. Consequently, abstractions that serve to selectively import or export bindings must be provided as part of the language kernel. These abstractions cannot be easily expressed in terms of other language primitives.

The model described here permits precisely this kind of functionality. The basic idea is to partition a binding environment into two distinct components. The first is a lexical environment that contains *lexical bindings* exclusively; the second is a public environment that contains *public bindings*. Public bindings are bindings that are *exportable* outside the lexical environment in which they are created. Applications which require variables to cross lexical boundaries treat the bindings associated with these variables as public.

There are five basic operations over public bindings and environments provided by the model:

(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 into a public environment. The bindings defined by this environment supersede public bindings in the current public environment.

Figure 1 depicts the relationship between lexical and public environments.

Reflection allows a data object that binds names to values to be transformed into a restricted scope within which other expressions can evaluate. In effect, reflection permits the dynamic construction and injection of new public environments (i.e., scopes) into a program. Expressions can then refer to the values of public bindings within this scope.

The operational inverse of reflection is *reification*. Just as we can transform data objects into public environments, the model also permits public environments to be

captured and transformed into data objects. Thus, environments within this model have a well-defined interpretation.

Central to this model are mechanisms to constrain the effect of the reification and reflection process. Programmers can curtail the set of bindings captured by a reification operation, and may delimit the scope of expressions evaluated within the context of a reflected data object.

Curtailment and public bindings provide a basis for preserving locality and information hiding even in the presence of arbitrary environment manipulation. Only public bindings are targets of reflection and reification operations; curtailment expressions delimit the scope of these operators. Reflection over reified environments permits bindings to be selectively shared across separate evaluation contexts. Both properties are crucial for modularity.

3. THE LANGUAGE

The syntax and nonreflective dynamic semantics of Rascal is identical to Scheme [Clinger and Rees 1991] augmented with operations to explicitly manipulate environments. Scheme is an expressive higher-order lexically scoped dialect of Lisp. We introduce the salient features of Rascal here, and elaborate on their use in the sections following.

Consider the following simple program:

```
(let* ((x 0)
       (y 1)
       (f (lambda (z) (+ y x z))))
  '((x ,x) (f ,f)))
```

In this expression, `x`, `y`, and `f` are lexical variables; their scope is delimited by the scope of the `let*` body. The expression returns a list that binds the symbols `x` and `f` to the lexical bindings of identifiers `x` and `f`. One can use this list as a representation for an environment containing bindings which map `x` to `0` and `f` to a closure in which the free references to `+`, `x`, and `y` are resolved relative to `f`'s definition-time environment.

Reification

In Rascal, environments and bindings may be manipulated directly, thus avoiding the need to construct ad hoc environment representations as in the example above. If we wished to export the binding values of `x` and `f`, i.e., treat their bindings as public, we would write:

```
(let* ((x 0)
       (y 1)
       (f (lambda (z) (+ y x z))))
  (make-public (x f) (reify)))
```

Let the lexical and public environment in which a `make-public` expression is evaluated be ρ_λ and ρ_κ, respectively. The operator takes a list of lexical variables L and an expression E. It 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`) returns the current public environment as a data structure.

The result of this program is an abstraction that represents the reified image of a

```
(let* ((x 1)
       (g (lambda (env) (reflect env (+ (use x) ((use f) 1))))))
  (g (let* ((x 0)
            (y 1)
            (f (lambda (z) (+ y x z))))
       (barrier (make-public (x f) (reify))))))
```

Fig. 2. Reflection gives access to public bindings.

public environment. This environment may contain potentially many bindings; in particular, it will define a public environment that maps x to 0 and f to a closure. The environment captured by this closure maps + to its definition in the lexical context within which the let* expression is evaluated, x to 0, and y to 1.

Curtailing Bindings

We constrain the public bindings that may be captured by reify using barriers. Consider the following expression:

```
(let* ((x 0)
       (y 1)
       (f (lambda (z) (+ y x z))))
  (make-public (y)
    (barrier (make-public (x f) (reify)))))
```

The barrier special form "hides" all public bindings in the current public environment within the extent of its argument expression. The object returned by this expression is thus a public environment with exactly two bindings, one for x and one for f. We think of barriers as the environment analogue of *prompts* [Felleisen 1988] used to constrain the effect of reification operations performed on continuations.

Reflection

Reflection is used to manipulate reified environments. For example, in the expression shown in Figure 2 the object returned by (reify) is passed as an argument to g. The body of this procedure transforms this object into a public environment. The resulting environment is then composed with the public environment extant at the point g is defined. Thus, a public binding in env supersedes a binding of the same variable in the public environment in which g is evaluated. Public bindings are accessed via the use special form. (Use x) returns the binding value of x in the current public environment. The reference to + in g refers to the definition of + in g's lexical environment; public references must be specified explicitly via the use special form.

Note that (make-public (x) (use x)) $\equiv$ x.

We can curtail the bindings injected by reflect by using the restrict operator. In the expression shown in Figure 3, the restrict operator unbinds all its argument variables from its public environment argument, returning the restricted environment as its result. Thus, my-env is bound to an environment in which x and y are unbound; in particular, the binding for x in env is absent in my-env. The

```
(let ((g (lambda (env)
           (let ((my-env (restrict env (x y))))
             (reflect my-env (+ (use x) ((use f) 1)))))))
  (g (let* ((x 0)
            (y 1)
            (f (lambda (z) (+ y x z))))
       (barrier (make-public (x f) (reify))))))
```

Fig. 3. Restriction unbinds names from a public environment.

```
(define f (let ((y 1))
            (make-public (y)
              (barrier (let ((x 2))
                         (make-public (x)
                           (lambda (r) (+ (reflect r (use x))
                                          (reflect r (use y))))))))))
```

Fig. 4. Barriers hide public environments.

public reference to x in g's body refers to x's public binding in g's definition-time (public) environment.

The reflect operator's public binding environment is the environment extant upto the closest enclosing barrier. For example, the code fragment shown in Figure 4 yields an error for any call to f in which r is not bound to a public environment that defines a binding for y. Thus, note that

```
(let ((R (barrier (let ((v1 e1)
                        (v2 e2)
                          ...
                        (vn en))
                    (make-public (v1 v2 ... vn)
                      (reify))))))
  (barrier (reflect R (use vj)))) ;; 1 ≤ j ≤ n,
```

is tantamount to a selection operation on a finite, closed namespace, and behaves identically to the "." operator provided over record objects in many Algol-based languages. Moreover,

(reflect (barrier (reify)) *exp*) ≡ *exp*,

(reflect (reify) *exp*) ≡ *exp*,

and

(barrier (reflect *N* (reify))) ≡ (barrier *N*)

Shared References and Assignment

Public and lexical environments may share references since make-public does not perform a deep copy of the copied binding's value. Side-effects to a heap-allocated

```
(barrier (let ((x (make-vector 10)))
           (vector-set! x 0 1)
           (make-public (x) (vector-set! (use x) 0 100))
           (vector-ref x 0)))
```

Fig. 5. Rascal objects are shared between public and lexical environments.

object made via a public binding may thus be visible in its lexical binding counterpart. For example, the value of the expression shown in Figure 5 is `100`, not `1`.

To allow variables in public environments to be assigned, Rascal contains a special assignment operator. The expression,

```
(set-public! x v)
```

assigns v to the variable x in the public environment in which the `set-public!` form is evaluated. Thus,

```
(let ((z (let ((x 1))
           (make-public (x) (reify)))))
  (reflect z
    (begin
      (set-public! x 100)
      (use x))))
```

returns `100`. Public bindings can be therefore mutated outside the lexical context in which they are defined.

Syntactic Domains

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

Semantic Domains

v	$\in Value = (Int + Bool + Symbol + Environment + Procedure)_\perp$
l	$\in Loc = Locations$
k	$\in Continuation = Value \to Store \to Value$
$r, \rho, \rho_\lambda, \rho_\kappa$	$\in Environment = Var \to Loc + unbound$
f	$\in Procedure = Value \to Continuation \to Store \to Value$
σ	$\in Store = Loc \to Value + undef$
ψ	$\in State = Environment \times Environment \times Store$

Valuation Function

$\mathcal{E}$	$Exp \to Continuation \to State \to Value$

Fig. 6. Syntactic and semantic domains

4. FORMAL SEMANTICS

Figure 6 gives the syntactic and semantic domains for an interesting subset of the language; the semantics is defined over a (lifted) value domain restricted to

$$
\begin{array}{l}
\mathcal{E}[\![c]\!]\,k\,\langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle = \\
\quad k(\ c,\ \sigma\ ) \\
\\
\mathcal{E}[\![x]\!]\,k\,\langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle = \\
\\
\quad k(\ \sigma(\ \rho_\lambda(\ x)),\ \sigma\ )\mathcal{E}[\![(\mathsf{lambda}\ \ (x)\ e)]\!]\,k\,\langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle = \\
\quad \textbf{let}\ \ func = \lambda\,\langle v,\ k',\ \sigma'\rangle.\textbf{let}\ \ \langle l,\ \sigma''\rangle = new(\ \sigma') \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ \mathcal{E}[\![e]\!]\,k'\,\langle \rho_\lambda[x \mapsto l],\ \rho_\kappa,\ \sigma''[l \mapsto v]\rangle \\
\quad \textbf{in}\ k(\ func,\ \sigma\ ) \\
\\
\mathcal{E}[\![(e_1\ e_2)]\!]\,k\,\langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle = \\
\quad \mathcal{E}[\![e_1]\!]\ \lambda\,\langle f,\ \sigma'\rangle. \\
\qquad\qquad \mathcal{E}[\![e_2]\!]\ \lambda\,\langle v,\ \sigma''\rangle.f(\ v,\ k,\ \sigma'') \\
\qquad\qquad\qquad\qquad \langle \rho_\lambda,\ \rho_\kappa,\ \sigma'\rangle \\
\qquad\quad \langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle
\end{array}
$$

Fig. 7. Definitions for functional core.

$$
\begin{array}{l}
\mathcal{E}[\![(\mathsf{make\text{-}public}\ \ (x)\ e)]\!]\,k\,\langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle = \\
\quad \textbf{let}\ \ \langle l,\ \sigma'\rangle = new(\ \sigma) \\
\quad \textbf{in}\ \mathcal{E}[\![e]\!]\,k\,\langle \rho_\lambda,\ \rho_\kappa[x \mapsto l],\ \sigma'[l \mapsto \sigma(\ \rho_\lambda(\ x))\,]\rangle \\
\\
\mathcal{E}[\![(\mathsf{use}\ \ x)]\!]\,k\,\langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle = \\
\quad k(\ \sigma(\ \rho_\kappa(\ x)),\ \sigma\ ) \\
\\
\mathcal{E}[\![(\mathsf{barrier}\ \ e)]\!]\,k\,\langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle = \\
\quad \mathcal{E}[\![e]\!]\,k\,\langle \rho_\lambda,\ \rho_\perp,\ \sigma\rangle \\
\\
\mathcal{E}[\![(\mathsf{restrict}\ \ e\ (x))]\!]\,k\,\langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle = \\
\quad \mathcal{E}[\![e]\!]\ \lambda\,\langle r,\ \sigma'\rangle.k(\ r[x \mapsto unbound\,],\ \sigma'\ ) \\
\qquad\quad \langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle \\
\\
\mathcal{E}[\![(\mathsf{reify})]\!]\,k\,\langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle = \\
\quad \textbf{let}\ \ r = \rho_\perp[x_1 \mapsto \rho_\kappa(\ x_1)\ldots x_n \mapsto \rho_\kappa(\ x_n)], \text{for } x_1,\ldots,x_n \in EnvDomain(\ \rho_\kappa) \\
\quad \textbf{in}\ k(\ r,\ \sigma\ ) \\
\\
\mathcal{E}[\![(\mathsf{reflect}\ \ e_1\ e_2)]\!]\,k\,\langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle = \\
\quad \mathcal{E}[\![e_1]\!]\ \lambda\,\langle r,\ \sigma'\rangle. \\
\qquad\qquad \mathcal{E}[\![e_2]\!]\,k \\
\qquad\qquad\qquad\qquad \langle \rho_\lambda,\ \rho_\kappa[r],\ \sigma'\rangle \\
\qquad\quad \langle \rho_\lambda,\ \rho_\kappa,\ \sigma\rangle
\end{array}
$$

Fig. 8. Definition of operations over public environments.

integers, Booleans, symbols, locations, environments, and procedures. We give a formal semantics for Rascal's reflection constructs in Figures 7, 8, and 9.

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 $StoreDomain(\ \sigma)$ 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 $l' \neq l$, $l \notin StoreDomain(\ \sigma)$, and $l \in StoreDomain(\ \sigma')$.

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

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

Fig. 9. Definition of assignment.

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 $EnvDomain(\,\rho)$ 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 [Steele Jr. 1978; Strachey and Wadsworth 1974], and models evaluation in terms of two environments. The first maps λ-bound variables, and the second maps public variables.

5. APPLICATIONS

Modern languages provide a gamut of modularity structures from simple records to complex objects and data abstractions. In most languages, these constructs rarely interact with one another in obvious or useful ways even though they may share close similarities in their formal definitions. The metalevel environment operations provided in Rascal are intended to make such similarities manifest. The goal is to allow the expression of progressive module elaboration with minimal program rewriting and minimal conceptual overhead. As a programmer preserves his idea of a natural module unit, but expands its functionality, it should be unnecessary to switch from one module construct to another. In this section, we examine a number of modularity structures found in modern languages and argue that incorporating environments as bona fide objects in a programming language (along with constraints on their reification and use) provides an *expressive* and *unified* framework for building a host of superficially diverse modularity devices. We consider a language feature expressive if it is simple (i.e., relatively few primitive operations applicable on it) *and* its "phrasebook" is short—little verbiage is required to define all the programming idioms a programmer is likely to express using this feature.

5.1 Static Environments

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,

```
(barrier (let ((x 1)
               (y 'bar)
               (z (list 1 2 3)))
          (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 same lexical context 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.

As a counterpoint to Rascal's treatment of records, consider Scheme's treatment of heterogeneous named objects. Like Rascal, Scheme also does not provide a primitive record constructor. In Scheme, the simplest way of providing records is in terms of an association list structure, or in some cases as a vector. Record labels are typically represented using symbols or integers that are used as keys into the appropriate data structure. In Rascal, there is no programmer-chosen representation for records; records are derived from application of procedures. Symbols or integers are not used as labels; record labels are simply public λ-bound variables. Thus, programmers need not devise an arbitrary program representation in order to define a record abstraction. Moreover, record labels are λ-bound variables visible to and manipulable by the Rascal compiler and runtime system; unlike records whose labels are represented in terms of quoted symbols, Rascal-style records are amenable to compile-time analysis. For example, the following expression can be flagged as potentially erroneous by a moderately sophisticated compiler:

```
(barrier (reflect (let ((x 1)
                        (y 'bar)
                        (z (list 1 2 3)))
                    (reify)))))
         (use foo))))  ;; no binding for a in R.
```

Furthermore, representing records in terms of reified environments makes it easy to define different kinds of record objects without having to construct new representations. For example, given a record R, we can construct a new record containing an extra field z thus:

```
(define (extend-record record field)
   (lambda (v)
     (barrier (reflect record (reflect (field v) (reify))))))

(extend-record  R (lambda (z) (make-public (z) (reify)))).
```

The call to extend-record returns a procedure. When applied, this procedure composes R with a reified representation of a namespace containing a single binding for z; the value bound to z is the value passed as the procedure's argument.

This definition does not raise an error if a record is extended with a field it already contains. We can ensure that such an error is raised by slightly rewriting the call to extend-record:

```
(extend-record  R (lambda (z)
                    (let* ((dummy-proc (lambda () 0))
                           (new-record (barrier (let ((z dummy-proc))
                                                  (make-public (z) (reify))))))
                      (reflect new-record
                         (reflect  R
                           (if (eq? (use z) dummy-proc)
                               (make-public (z) (reify))
                               (error "Can't extend record."))))))).
```

In this expression, a public environment is created in which z is bound to a locally defined dummy procedure, dummy-proc. This environment is then composed with R. If the public binding for z in this namespace is *not* dummy-proc, an error is raised. Otherwise, an abstraction is returned that is defined to be the composition of R with a record containing a field for z. As we show in the next section, simple syntactic forms make it is easy to abstract these operations over any name; using such forms would enable extend-record to perform error checking of this kind directly, rather than requiring it to be performed by its callers.

5.1.1 *Modules.* In addition to providing record types, languages such as Ada [U.S. Dept. of Defense 1982], Clu [Liskov et al. 1977], and Modula-2 [Wirth 1985] 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 not 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

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

```
(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)))))
```

Fig. 10. A stack abstraction in Rascal.

records are. To illustrate, consider the definition of a stack shown in Figure 10. 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, 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 [MacQueen 1988] – as in Rascal, there is no special runtime representation for modules in ML; modules are represented 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 as shown in Figure 11.

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

```
(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)))))
```

Fig. 11. A simple stack abstraction in Scheme.

```
(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)))))
```

Fig. 12. A stack client interface procedure.

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 reimplementation of a module will not require reimplementation of clients using this module.

Unfortunately, the stack program shown in Figure 10 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 (see Figure 12).

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`:

```
(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)))))
```

Fig. 13. The implementation interface for stacks.

```
(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 Figure 13.

The test procedure returns true if the implementation provides procedures named create, push, and pop. The install procedure reassigns 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 Figure 14.

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

```
(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) 1))
                              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)))))
```

Fig. 14: 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.

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 which clients 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 [1990] system, 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 firstclass, 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 [1993] module description. 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 7.

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 languages in which modules are defined as part of the language kernel, e.g., Ada [U.S. Dept. of Defense 1982] or Modula-2 [Wirth 1985], 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., Kohlbecker and Wand [1987] and Clinger and Rees [1993]), 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.

[2]In most implementations of ML, a functor is treated as a first-order procedure used to link modules together and to enforce type constraints among modules that share bindings. Harper and Lillibridge [1994] discuss a type framework for higher-order (first-class) modules in SML; MacQueen and Tofte [1994] and Tofte [1992] suggest a semantics of higher-order functors.

```
(define-syntax make-impl-interface
  (syntax-rules ()
    ((make-impl-interface ((b1 b2 ...))
      (let ((test (lambda (impl)
                    (let ((b1 (lambda () b1))
                          (b2 (lambda () b2))
                          ...)
                      (reflect (barrier (make-public (b1 b2 ...)
                                                     (reify)))
                               (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)))))))
```

Fig. 15. A syntactic abstraction for implementation interfaces.

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

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

Both make-client-interface and make-impl-interface take the names of the interface procedures as arguments.

The implementation of make-impl-interface using a define-syntax macro specification form of the kind available in Scheme [Clinger and Rees 1991] is given in Figure 15.

Similar syntactic forms can be built for abstracting other details of implementations, and installation procedures. Application programmers then would 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.

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 illustrate, consider the Scheme 48 package definitions shown in Figure 16; Scheme 48 is a byte-code implementation of Scheme with several major extensions, one of which is a sophisticated module system [Rees 1993].

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

```
(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))))
```

Fig. 16. Some simple package definitions in Scheme 48.

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 a 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 take place only 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 Figure 17). 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 Scheme 48'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.

5.2 Dynamic Environments

Dynamic environments, i.e., environments which are incrementally created, are usually built and maintained by a language's front-end (FE). The FE typically acts as an evaluator that repeatedly reads a new input expression, evaluates it on the basis of the current internal environment structure, augments the environment if necessary, and prints the result. Users usually do not have access to the internal

```
(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)))))))))
```

Fig. 17. Implementing modules as structures using public environments.

state of the FE, and programs to access and manipulate the environment image of an interpreter session must usually be provided as part of the evaluator package.

On the other hand, the introduction of metalevel environments as bona fide data objects has interesting ramifications for the construction of front-end's. In particular, programmability of metalevel environments makes it possible for users to write customized functions that inspect, coalesce, or maintain multiple name spaces; the environment maintained by the front-end now becomes an object that may be freely examined and manipulated.

For example, we might choose to define a top-level environment initially as an empty namespace:

```
(define top-level (barrier (reify))).
```

We can then proceed to define a "FE-define" operator that adds a new binding to top-level; its definition is given in terms of a more fundamental composition operator on environments (see Figure 18).

Adding a new definition returns a new public environment which is composed of the old top-level with an environment containing a single binding, B. B's name is Name, and its binding value is determined by evaluating Exp relative to top-level.

We can easily provide other kinds of functionality usually not available in most languages that provide implicit dynamic environments. (Lisp, ML, and Prolog are three notable examples.) For example, we can "undefine" a binding in our top-level

```
(define (compose Env1 Env2)
  (let ((a Env1) (b Env2))
    (barrier (reflect a  (reflect b (reify))))))

(FE-define Name Exp) ≡
  (set! top-level
        (compose top-level
          (barrier (let ((Name (reflect top-level Exp)))
                     (make-public (Name) (reify)))))))
```

Fig. 18. A front-end "define" operator.

environment:

(FE-undefine *Name*) ≡ (set! top-level (restrict top-level (*Name*))).

The object assigned to top-level is a namespace that contains all bindings previously input to top-level except the binding for *Name*. Previously input definitions that refer to *name* are unaffected by this operation. Thus, consider the following script:

(with-top-level x) ≡ (barrier (reflect top-level (with-top-level x)))

```
(FE-define x 1)
(FE-define f (lambda (y) (+ y (use x))))
(FE-undefine x)
((with-top-level f) 1)  ⟹  returns 2
(with-top-level x)      ⟹  returns undef
```

The application of f to 1 returns 2 despite the fact that x has been undefined by the previous operation. F is closed over an environment that contains a binding for x; the effect of an undefine operation is visible only to those expressions subsequently evaluated.

We can reinitialize top-level via the procedure close:

```
(define (close)
  (let ((NewEnv top-level))
    (set! top-level (barrier (reify)))
    NewEnv)).
```

Interpreted languages often provide an eval procedure that evaluates a textual representation of an expression in the context of a metalevel environment. The functionality of eval is subsumed in Rascal by reflect. Unlike eval, the power of reflect is constrained by public variables, barriers, and restrictions on the variables which are captured. Furthermore, the eval procedure typically evaluates expressions via an environment which it maintains, and which is hidden from the expressions being evaluated. This is not the case in Rascal. Thus, one can define multiple dynamic environments within which evaluation may occur, and programmers have the flexibility to freely add new operations to these environments.

In most interpreted languages, the definition of the FE constrains an expression

to be evaluated relative to either its lexical or dynamic environment; the choice is usually hard-wired as part of the top-level evaluation procedure. For example, free variables in ML procedures are resolved relative to the environment extant at the time the procedure is evaluated; in Common Lisp or Scheme, top-level redefinitions are implemented in terms of assignment (not rebinding), thus giving the semblance of dynamic binding at the toplevel. The interpretation choice in these systems cannot easily be altered by their users. According to the definition of define given above, Rascal expressions input to a top-level environment are evaluated relative to the environment extant at the point of definition, not at the point of use. As we discuss in the following section, however, Rascal programmers are free to change this protocol without having to reimplement a new evaluator.

5.3 Binding Protocols

Most languages that come equipped with a default binding protocol rarely provide facilities by which this protocol can be overridden cleanly. Lexical binding languages with higher-order procedures are a good case in point. The primary environment-building structure in these languages is typically a closure that is built and maintained by the underlying interpreter: users cannot write down an expression that defines the representation of a closure, nor can they examine a closure object from within the language.

It therefore becomes problematic to implement variations on a given binding protocol – since users do not have access to the binding environment within which expressions are evaluated, they cannot alter the environment in any way not originally prescribed by the language design. This is an important limitation in the expressiveness of the language, and it often necessitates extended dialects to provide either ad hoc constructs to realize other binding disciplines (e.g., fluid-let [Abelson and Sussman 1985] in Scheme to achieve dynamic binding), or to implement significant extensions to the base language (e.g., extensions for supporting late-binding and object-based programming[Adams and Rees 1988]).

Rascal, like many other higher-order lexically scoped languages, also represents procedures in terms of closures. The fact that the language provides operations to explicitly capture an environment, however, makes the lexical binding rule logically unnecessary. In other words, we could ascribe a late-binding semantics to procedures (or individual variables) without necessitating any alteration to the base language semantics. For example, a late-binding procedure,

(lambda$_d$ (x1 x2 ... xn) *Exp*)

is equivalent to

```
(barrier
  (lambda (Env)
    (reflect Env (lambda (x1 x2 ... xn) Exp'))))
```

We rewrite a late-binding procedure into a higher-order early binding one that takes as its argument the reified image of the dynamic environment and returns a procedure that evaluates in the context of this environment. *Exp′* is identical to *Exp* except that all free variables referenced in *Exp* are marked public in *Exp′*. Thus, public free variables in the body of the abstraction are evaluated relative to their binding value in Env. The barrier preceding the definition ensures that

public names referenced in the body of the procedure not captured in the dynamic environment defined by Env raise an error. An application of such a procedure,

(ef e1 ... en)$_d$

is equivalent to

```
(let ((Env (reify))
      (Proc ef))
  ((Proc Env) e1 ... en)).
```

Env and Proc are assumed to be fresh variables. Note that if a public variable occurs free in Exp, but is not defined to be public in the dynamic environment, the application is considered ill defined; it is straightforward to weaken this behavior if desired.

The ability to freely combine lexical and dynamic environments leads to useful functionality. For example, reflective environments in conjunction with an expressive macro facility can be used to build a simple optional keyword argument facility similar to that found in Common Lisp [Steele Jr. 1990] without requiring any change to the underlying runtime interpreter. An expression of the form,

```
(lambda_k (arg_1 arg_2 ... arg_n
           &keyword (karg_1 d_1)
                    (karg_2 d_2)
                        ...
                    (karg_k d_k))
  E)
```

defines a procedure that takes arguments arg_1, arg_2, ... arg_n and a series of optional arguments passed by keywords karg_1, karg_2,..., karg_k with default values d_1, d_2, ..., d_k. Thus, given the procedure,

```
(define key (lambda_k (a b c &keyword (d 1) (e 2))
              (list a b c d e))).
```

we observe that

```
(key 1 2 3) yields (1 2 3 1 2).
(key 1 2 3 (:e 4)) yields (1 2 3 1 4). and
(key 1 2 3 (:e 4) (:d 5)) yields (1 2 3 5 4).
```

We can transform a keyword procedure of the form shown above to the Rascal expression shown in Figure 19. All free occurrences of karg_i, $1 \leq i \leq k$ in E are transformed to (use karg_i) in E'.

An application of a keyword procedure,

```
(apply-keyword f arg_1 arg_2 ... arg_n
               (:karg_1 v_1)
               (:karg_2 v_2)
                   :
               (:karg_j v_j)).
```

is equivalent to

```
(lambda (Env arg1 arg2 ... argn)
  (let ((karg1 d1)
        (karg2 d2)
          ...
        (kargk dk))
    (barrier
      (make-public (karg1 karg2 ... kargn)
        (reflect Env E')))))
```

Fig. 19. A keyword abstraction implemented in Rascal.

```
(f (let ((karg1 v1)
         (karg2 v2)
           ...
         (kargj vj))
      (barrier
        (make-public (karg1 karg2 ... kargj) (reify))))
    arg1 arg2 ... argn)
```

Keywords argument are encapsulated in a namespace that is passed as the first argument to the procedure; the body of the procedure is evaluated relative to this namespace. This namespace contains bindings only for the keyword arguments. References to keyword arguments that are not provided by the caller resolve to their default value.

5.4 Object-Based Programming

Object-based computing is an important domain for environment metaprogramming. We relate operations of the kind provided by Rascal with those provided in object-oriented programming languages by observing that objects define localized namespaces; thus, inheritance and delegation operations over objects (e.g., Goldberg and Robson [1983] and Liebermann [1986]) require manipulation of namespaces in ways that utilize the functionality provided by reflect and reify.

Simple procedural abstractions by themselves are a poor choice for building expressive object systems. Of course, it is possible to express object-based programming in languages like T [Adams and Rees 1988] or Common Lisp [Steele Jr. 1990] that are lexically scoped. Support for objects in these systems however involve significant extensions or alterations to a simple language kernel. More significantly, it is nontrivial to understand the semantics of objects in these languages based only on an understanding of the primitive operations that define the kernel.

We argue that procedural abstractions serve a purpose orthogonal to the needs of a general object-based system. A procedure parameterizes an expression across a collection of different inputs. When building objects with inheritance or delegation semantics, however, one needs to construct a collection of distinct abstractions which share access to a common set of inputs (e.g., superclass methods and instance variables shared among subclasses). Environment reflection and reification contribute to a programming methodology that naturally supports such constructions; it does so by permitting environments to be shared across distinct evaluation contexts.

```
(define MakeBox
  (let ((grid shape of grid on which boxes are printed))
    (lambda (x y length width)
      (letrec ((move (lambda (self new-x new-y)
                       (barrier
                         (let ((new-box ((reflect self (use clone)))))
                           (reflect new-box
                             (begin
                               (set-public! x new-x)
                               (set-public! y new-y)
                               new-box))))))
               (print (lambda ()
                        print box using grid's coordinates))
               (clone (lambda ()
                        (make-public (x y length width grid
                                        clone move print)
                          (reify)))))
        (clone)))))
```

Fig. 20. A simple box object using delegation.

5.4.1 *Delegating Operations to Objects.* Reflection and reification of public environments permit abstractions to share free variable bindings even if they do not exist in the same scope. This capability permits objects (or procedures) to share operations and data defined within other objects (or procedures). Given this functionality, instances of an object may have different behaviors based on how they choose to delegate operations [Chambers and Ungar 1989; Ungar et al. 1992].

To illustrate, consider the familiar problem of specifying geometric objects. A box object consists of the x and y coordinates of its left corner, a length, a width, a method to move boxes from one coordinate to another, a method to clone a new instance of a box, and a print method to print box objects on a grid of some dimension. Other implementations of box objects may define new methods or provide alternative implementations of existing ones, but will still wish to share bindings for these methods and variables. The outline of a simple box generator implemented in Rascal is shown in Figure 20.

The clone procedure creates a new instance of a box. Note that a new environment for the cloned object is constructed containing separate bindings for its instance variables and methods, but the values of these bindings are shared with the original.

By making the operations and variables associated with a box public, instantiations of other kinds of boxes (with possibly different behaviors) can be generated from any instance of a simple box. A new instance of a box is created by evaluating

(define MyBox (MakeBox *arguments*)).

Suppose we now define a specialized kind of box called a colored box. A colored box, in addition to containing box shape information, also contains a color that associates coordinates to distinct colors, and a redefined print method sensitive to colored boxes. Given the existence of a box object P, we wish to avoid respecifying

```
(define MakeColorBox
  (lambda (box color)
    (letrec ((print (lambda ()
                      (barrier (reflect box
                                 new print routine that references grid))))
             (clone (lambda ()
                      (compose box (make-public (color clone print)
                                                (reify))))))
      (clone))))
```

Fig. 21. A color box object.

the initial coordinates and the move method when creating a new instance of a colored box; instead, we would like to treat P as a prototype object upon which colored boxes can be defined. Color box instances delegate requests for moving colored boxes and determining current coordinates to P; in other words, the operation of moving colored boxes is delegated to ordinary box objects. We define one implementation of a color box below in Figure 21.

To create an instance of a color box parameterized from MyBox, we write

```
(define MyColorBox (MakeColorBox MyBox initial_color)).
```

The new print method found in MyColorBox uses the public binding value of grid defined by MyBox. The color print method is a modified implementation of the print method defined for a simple box. MyBox's print method is shadowed by MakeColorBox's definition.

The public environment returned by MakeColorBox also contains the public binding for move as defined by MakeBox; both boxes and color boxes share this method. For example, to move a color box, we write

```
(barrier
  ((reflect MyColorBox (use move)) MyColorBox new-x new-y)).
```

The move procedure for MyBox clones a new color box, and sets the x and y coordinates of that new box to *new-x* and *new-y*, respectively; this new color box inherits methods from MyBox, the prototype box object used to create MyColorBox.

First-class environments permit the expression of *dynamic inheritance* [Chambers and Ungar 1989]. A system that supports dynamic inheritance allows new methods to be incorporated into the object hierarchy dynamically, or existing methods to be removed. For example, we can construct a stationary box object from a prototype such as MyBox thus:

```
(let ((NewBox ((reflect MyBox (use clone)))))
  (restrict NewBox (move)))
```

This expression returns a box object with no move procedure. The coordinates of new objects cloned from this box will all be immutable.

Critics might argue that building modified versions of object generators is possible even in the absence of reification and environment-based reflection. For example, rather than using (reify) to capture the public bindings defined by a prototype

object generator, we could structure our program such that all related versions of an abstraction (e.g., boxes and color boxes) reside in the same lexical context. This obviates the need to explicitly package and unpackage bindings via environment-manipulating operations. The approach has the significant limitation, however, of requiring the original prototype environment to be altered whenever a modification is made. Modularity is significantly reduced as a result.

An alternative solution would be for all instances derived from a prototype object to specify explicitly the data exported by the prototype in their interface specification. Such a solution comprises modularity since it forces data relevant only to the implementation of boxes to become visible in the interface specification of any abstraction that is derived from it. Packaging details of a box prototype in the specification of a colored box would be an unfaithful characterization of its specification. In the example given above, all relevant binding information for boxes as viewed by a colored box is encapsulated within the namespace returned by (clone); instances of color boxes need not be aware of any bindings defined by the box prototype that are not explicitly required in their definition; thus, changes to the prototype that do not affect the interface are immediately visible to objects instantiated from it.

Reification contributes to a programming methodology that is the operational inverse of ordinary function abstraction: abstraction parameterizes an expression over a set of presumably different inputs; reification parameterizes a set of inputs over presumably different expressions. In the above example, the input coordinates for MyBox are used in the definition of MyColorBox – the same arguments are used to construct two different abstractions.

5.4.2 *Inheritance.* Code reusability is a form of incremental programming: new programs can be generated by specifying how they differ from existing ones. Incremental programming techniques are complicated by the fact that a modified structure may contain mutually recursive components. (Neither the stack nor the colored-box example highlighted this issue.) Free references occurring within the recursive components of such structures must be resolved relative to the state of the modified object and not the original.

Cook and Palsberg[1989], Reddy[1988], and Kamin[1988] discuss how to build class-based inheritance systems that permit construction of modified versions of recursive structures using explicit fixpoint notation. In essence, a fixpoint semantics is used to give a nonoperational definition of the *self* pseudovariable found in Smalltalk-style languages.

Building modified versions of recursive structures is also possible using operations over metalevel environments. Reflection and reification permit modified versions of objects to be created while still allowing access to the component elements found in the original. The modified version might define new definitions for bindings found in the original; the old definitions are still accessible, however, since environments can be projected and captured. Rascal is distinguished from these other proposals insofar as it provides a concrete self-contained *linguistic framework* within which *both* class-based and delegation-style inheritance strategies can be expressed.

The notion of *self* is implemented using namespace composition, reflection, and reification. Classes are namespace generators, and a class hierarchy is built by composing new instances of namespaces generated from a set of superclasses; these namespaces are composed with the bindings found in the current evaluation envi-

ronment. Reification gives access to this environment.

To illustrate how to use reflection to build inheritance systems that have recursive components, consider an example discussed in Cook and Palsberg [1989] and Kamin [1988]. A circle is a subclass of a point. The point definition contains instance variables x and y to specify its location and defines two methods: DistfromOrig computes the distance of a point from its origin, and ClosertoOrig takes another point object as its argument and returns true if the point is closer to the origin than its argument, and false otherwise; it uses DistfromOrig to compute its answer. A circle is defined in terms of points. Because circles have a radius, they have a different meaning of distance from the origin. The notion of distance from origin for circles is given in terms of the definition of DistfromOrig found in point objects: if l is the distance from the origin to the circle's center and r is the circle's radius, then $l - r$ gives the distance from the origin of the circle object. If this difference is negative, the distance is assumed to be 0. Figure 22 gives a definition of a point class, and Figure 23 gives a definition of a circle class.

Methods defined in point must evaluate relative to the bindings defined by point's callers. Typically, one abstracts the desired evaluation environment into **self** and **super** pseudovariables. We can treat these variables as ordinary user-defined objects using operations on environments. The value of self with respect to point is a namespace that contains bindings defined by the (a) point's caller, (b) instance variables and methods of the current point instance, and (c) methods and instance variables defined by point's superclass (in this order). In this example, point's superclass is the null object represented as an empty environment.

Because Rascal is a strict language, we use abstraction and assignment to model nonstrict mutual recursion. Method is a syntactic form that expands to a procedure which evaluates its body relative to the environment image of its self argument. To send a message to a method, we use the syntactic abstractions send and delegate. Given an object *obj*, a method *message*, and an argument list *args*, both send and delegate resolve the definition of *message* relative to *obj*; send, however, causes the "self" argument of the method to be bound to *obj*, whereas delegate binds "self" to its definition in the caller object. In this implementation, each instance of a point is closed over its own copy of a point's methods; this is in contrast to, e.g., Smalltalk where all instances derived from the same class share the same methods. Thus, different point instances may modify their method definitions locally.

Methods evaluate their body using the reflected environment image of their self argument. Late-binding (i.e., the definition of self) and method sharing are realized by environment reification. For example, the first reference to DistfromOrig in ClosertoOrig refers to DistfromOrig's binding-value in self. The definition of this method is determined by structure of point's caller, C; if C does not define such a method, the binding value of point's local definition is used. The object returned by the point generator contains bindings for x and y as well as the public bindings for DistfromOrig and ClosertoOrig.

Circles are structurally identical to points except that they have a superclass. A superclass instance is typically accessed via the **super** pseudovariable. In Rascal, however, super acquires legitimacy as a namespace denoting instance bindings of the current object's superclass. Unlike a point instance, circle instances contain method definitions and instance variables for both points and circles.

The code for the circle generator is given in Figure 23. Note that since circle

```
(define make-point
  (barrier (lambda (x y)
             (letrec ((super (lambda ()
                               (set! super (barrier (reify)))))
                      (DistfromOrig (method ()
                                      (sqrt (+ (sqr (use x)) (sqr (use y))))))
                      (ClosertoOrig (method (p)
                                      (< (send self DistfromOrig)
                                         (send p DistfromOrig)))))
               (let ((local (make-public (x y DistfromOrig
                                          ClosertoOrig))))
                 (reify))))
             (super)
             (compose super local))))))

(method (arg_1 arg_2 ... arg_n) body) ≡
  (lambda (self arg_1 arg_2 ... arg_n)
    (reflect self body))

(send obj message . args) ≡
  (let ((m (barrier (reflect obj (use message)))))
    (apply m (cons obj args)))


(delegate obj message . args) ≡
  (let ((m (barrier (reflect obj (use message)))))
    (apply m (cons self args)))
```

Fig. 22. Implementation of a point class.

instances inherit the ClosertoOrig method defined by points, the free reference to DistfromOrig in ClosertoOrig defined by point instances created by circles must refer to its definition in the circle instance, and not its definition in the point one.

Circles inherit properties of points. The superclass of a circle is a point; thus, circle instance's define their super binding in terms of new point instances. The bindings used to create a new point contain a definition for DistfromOrig as defined by the circle instance; moreover, the definition of DistfromOrig in circles refers to the binding value of DistfromOrig as defined by point.

Besides containing bindings for DistfromOrig, x, y and r, the object returned by applying a circle generator contains the binding for ClosertoOrig as found in the point instance associated with this new circle.

A slight generalization of the technique outlined above can be used to support a simple form of multiple inheritance. Suppose that C_1 and C_2 are disjoint generators that are intended to be used as superclasses of C_3. We can specify the superclass methods available to C_3 by composing the environment representation of C_1 and C_2; this image would contain the method definitions for objects instantiated from C_1 and C_2.

To summarize, in a reflective model of the kind described here, the notion of

```
(define make-circle
  (barrier (lambda (x y r)
             (letrec ((super (lambda ()
                               (set! super (make-point x y))))
                      (DistfromOrig (method ()
                                      (max (- (delegate super DistfromOrig)
                                              (use r)) 0))))
               (let ((local (make-public (x y r DistfromOrig)
                                         (reify))))
                 (super)
                 (compose super local))))))
```

Fig. 23. A circle class.

```
(define circle-class (make-class
                       (instance-variables x y r)
                       (methods
                         ((DistfromOrig
                           (method ()
                             (max (- (delegate super DistfromOrig)
                                     (use r))
                                  0)))))
                       (super-class point-class (x y))))
```

Fig. 24. A syntactic abstraction of a circle class.

self is implemented using namespace composition, reflection, and reification over public environments. Classes are namespace generators, and a class hierarchy is built by composing new namespace instances generated from a set of superclasses; these namespaces are composed with the bindings found in the current evaluation environment. Reification gives access to method definitions found in this environment. Reflection captures the notion of code sharing: by using reflection, we can specify that a binding not defined within one environment be resolved in another.

As is the case with modules, the namespace operators found in Rascal do not provide syntactic abstractions to relieve programmers from constructing implementations of classes and instances whenever a new object is defined. However, examination of make-circle and make-point reveals a great deal of common structure; this structure can be captured by simple syntactic abstractions. For example, one can specify the semantics of circle classes using a make-class macro (see Figure 24). The definition of make-class is given in Figure 25.

6. IMPLEMENTATION

The ability to capture metalevel environments and apply them in different contexts raises implementation issues similar to those faced by implementors of dynamically typed, late-binding languages such as Smalltalk [Deutsch and Schiffman 1983] or Self [Chambers and Ungar 1989]. To build an efficient implementation of Rascal requires techniques to associate (in as many cases as possible) names with the

```
(define-syntax make-class
  (syntax-rules (instance-variables methods super-class)
    ((make-class (instance-variables x1 ...)
                 (methods ((m1 d1) ...))
                 (super-class class (arg1 ...)))
     (barrier (lambda (x1 ...)
                (letrec ((super (lambda ()
                                  (set! super (class arg1 ...))))
                         (m1 d1)
                         ...)
                  (let ((local (make-public (x1 ... m1 ...)
                                            (reify))))
                    (super)
                    (compose super local))))))))
```

Fig. 25. Syntactic abstractions can simplify the construction of classes in Rascal.

locations containing their potential binding values. The problem is nontrivial since reified environments can be used in contexts far removed from their point of capture.

To illustrate the problem, consider the following Rascal program fragment:

```
(define f (barrier (let ((id 1))
                     (make-public (id)
                       (lambda (R) (reflect R (use id)))))))
```

F is bound to a procedure that takes a namespace as its argument and treats it as a scope within which id is evaluated. However, the semantics of reflect dictates that R need not define a binding for id — if it does not, the meaning of id is taken to be its binding value in the procedure's barrier bounded public environment. Thus, the following applications of f are both valid:

```
(f (barrier (reify)))
(f (barrier (let ((id 'foo))
              (make-public (id) (reify)))))
```

In the first application, f's formal argument R is bound to an empty namespace. The result in this case is 1. In the second application, R is bound to a namespace that defines a binding for id; the result here is foo.

A naive implementation of Rascal would represent public environments as a chain of frames in which each frame contains the bindings of by a public environment. Reification captures this chain; the use operator searches the chain for the binding value corresponding to its argument variable; reflect simply adds one chain onto the end of the other; and restrict and barrier delete bindings and frames, respectively.

In this implementation, a significant cost is paid whenever the binding value of a public variable is referenced. This is because a runtime search down a potentially large number of environment frames must be performed for every such reference. For example, in the expression,

```
(lambda (env) (reflect env (begin ... (use y) ...)))
```

the reference to y will not be compiled as a reference or index into a known data

structure. Instead, a search down the chain represented by public environment env will be initiated; this is a potentially expensive operation if the binding for y is defined deep in the chain. Note that a shallow binding implementation would eliminate an expensive chain walk, but would complicate the restoration of public bindings upon exit from a public scope.

A more clever strategy will attempt to optimize the implementation of reification and reflection operations in two basic ways: (1) finding all contexts in which a reified environment is used and (2) minimizing the runtime cost of references to public variables by analyzing call and barrier points. We enumerate some of the optimizations that derive from this basic strategy below:

(1) *Compiling Namespaces as Records:* If the contexts in which a reified environment is used are all known, the namespace can be compiled as a record R, and reflect operations and public variables referenced within reflect operations using this namespace can be compiled as offsets into R. Consider the following fragment:

```
(barrier
  (let ((R (barrier
             (let ((x E1)
                   (y E2))
               (make-public (x y) (reify)))))
    ...
    (reflect R (use x))
    ...
    (reflect R (use y))
    ...)))
```

Assuming all use points of R are known, it can be compiled as a two-field record with labels x and y. The references to x and y in the reflect expressions found in the let body can be implemented as offsets into this record.

(2) *Efficient Environment Representation:* The overheads in capturing an environment can be reduced by recording references to public variables in the body of reflect operations. For example, in the following fragment, x need not be recorded in the reified environment image captured by reify.

```
(reflect (barrier (let ((x E1)
                        (y E2))
                    (make-public (x y) (reify))))
  (use y))
```

The above optimization can be restated as follows. Let P be the set of all program points that use a public environment R. R need contain bindings only for the public variables referenced explicitly by expressions in P even though there may be many more public bindings defined in R. Consider the following fragment.

```
(let ((R (barrier
          (let ((a1 E1)
                (a2 E2)
                ...
                (an En))
            (make-public (a1 a2 ... an)
              (reify))))))
  ...
  (reflect R (use a1))
  ...
  (reflect R (use a2)))
```

Assuming R does not escape the let body, only bindings for a_1 and a_2 need to be captured by reify; all other public variables bound in the let can be treated as ordinary lexical variables. Thus, R can be implemented as a single frame containing bindings for a_1 and a_2.

In certain instances, we can avoid constructing explicit representation of public environments altogether. For example, consider procedure f as defined below.

```
(define (f z)
  (barrier
    (reflect z
      ... (use x) ... (use y) ...)))
```

If all calls to f are known, we can eliminate the reflect operation and "lift" references to x and y as arguments to the procedure,

```
(define (f x y)
   ... x ... y ...)
```

An application of f originally structured as (f r) would be rewritten as

```
(f (barrier (reflect r (use x)))
   (barrier (reflect r (use y)))).
```

(3) *Customization:* If all call points of a procedure abstracted over a public environment are known, its implementation can be specialized for each call [Chambers and Ungar 1989; Ungar et al. 1992]. Consider the expression below.

```
(let ((f (lambda (R)
           ...
           (reflect R (use x))
           ...)))
  ...
  (f (barrier
       (let ((x E))
         (make-public (x) (reify)))))
  ...
  (f (barrier (reify))))
```

If f does not escape, and its body is relatively small, this fragment can be translated to read,

```
(let ((f1 (lambda (R)
            ...
            (barrier
              (reflect R (use x)))
            ...))
      (f2 (lambda (R)
            ...
            (use x)
            ...)))
  (f1 (barrier
        (let ((x E))
          (make-public (x) (reify)))))
  ...
  (f2 (barrier (reify)))).
```

The reference to x in f_1 is implemented as an offset into the record implementation bound to R; the reflect operation can be eliminated altogether in f_2.

(4) *Local Caching:* Even if customization and general transformation of references to offsets is not possible, the overhead of runtime name lookup can be minimized by building an intermediate representation structure used to "cache" references to public variables within reflect operations. To illustrate, consider the following program fragment:

```
(reflect (f args)
  ... (use x) ... (use y) ...).
```

Suppose x and y are the only public variables referenced in the reflect body. If f is not known, the address of x and y's binding value cannot be determined until runtime; to avoid performing runtime name lookup on every reference, however, this fragment can be implemented as follows:

(a) Construct a record R containing fields labeled x and y.
(b) Translate all public references to x and y in the body of reflect as appropriate offsets into R.
(c) Evaluate (f *args*) and call the reified object returned O.
(d) Store in the slot for x and y in R their value in O (if one exists); otherwise, store their value as determined by their public lexical environment.

In this translation, runtime lookup occurs only when the label slots in R are initially filled; all subsequent references within the scope of the enclosing reflect expression are implemented as ordinary references off a known offset in a record structure.

Many of these optimizations have been applied within other systems. Self [Ungar et al. 1992] implements customized versions of methods based on type information of the receiver. Orbit [Kranz et al. 1986] builds specialized representations of closures based on the structure of the closure's lexical environment and the contexts in which the closure is applied. The process of transforming free references in procedures to bound ones is known as λ-lifting and is described in Johnsson [1985] and Peyton-Jones [1987]. The SML/NJ compiler [Appel 1992] uses an optimization called inverse η-reduction to generate specialized versions of functions whose calls sites are all known.

7. COMPARISON TO RELATED WORK

Pebble [Burstall and Lampson 1984] is a kernel language designed to facilitate the construction 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 [Abelson and Sussman 1985], 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. Additionally, 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 [Smith 1982; des Rivières and Smith 1984] or Brown [Friedman and Wand 1984; Wand and Friedman 1988] 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, and make no distinction between ordinary variables and those which can be captured by reification. 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 [Goldberg and Robson 1983] and Self [Ungar et al. 1992] 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. Furthermore, the implementation of objects and methods in these languages does not rely significantly on contexts. In Rascal, object-based programming is expressed entirely via operations on meta-level environments; no additional machinery beyond that provided for reflection of namespaces and reification of environments is used to implement object-based protocols.

Lee and Friedman [1993] propose an alternative treatment of names that manipulates variables rather than identifiers. 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 bindings. The use of a different metaobject to implement various modularity devices, however, clearly separates their work from ours. Control over environments is obtained in Rascal using operations 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 [Gelernter et al. 1987; Jagannathan 1989] 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 [1988] 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 Henry Cejtin, Shinnder Lee and the anonymous referees for their many useful comments and suggestions.

REFERENCES

Abelson, H. and Sussman, G. 1985. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass.

ADAMS, N. AND REES, J. 1988. Object-oriented programming in Scheme. In *Proc. of the ACM Symp. on Lisp and Functional Programming*, 277–288. ACM, New York.

APPEL, A. 1992. *Compiling with Continuations*. Cambridge University Press, Cambridge, U.K.

BURSTALL, R. AND LAMPSON, B. 1984. A kernel language for modules and abstract data types. In *International Symp. on Semantics of Data Types*, Volume 173, Springer-Verlag, New York. Lecture Notes in Computer Science.

CARDELLI, L., DOHAHUE, J., JORDAN, M., KALSOW, B., AND NELSON, G. 1989. The Modula-3 type system. In *the* 16^{th} *ACM Symp. on Principles of Prog. Languages*, ACM, New York, 202–213.

CHAMBERS, C. AND UNGAR, D. 1989. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN '89 Conference on Prog. Language Design and Implementation*, ACM, New York, 146–160.

CLINGER, W. AND REES, J. 1991. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers 4,* 3 (July), 1–55.

CLINGER, W. AND REES, J. 1993. Macros that work. In *the* 18^{th} *ACM Symp. on Principles of Prog. Languages*, ACM, New York. 155–163

COOK, W. AND PALSBERG, J. 1989. A denotational semantics of inheritance and its correctness. In *OOPSLA'89 Conference Proceedings, ACM SIGPLAN Not. 24(10)*, (Oct. 1989), 433–444.

CURTIS, P. AND RAUEN, J. 1990. A module system for Scheme. In *Proc. of the ACM Symp. on Lisp and Functional Programming*, ACM, New York. 13–19.

DAHL, O., MYHRHAUG, B., AND NYGAARD, K. 1970. The Simula67 base common base language. Tech. rep., Norwegien Computing Center.

DES RIVIÈRES, J. AND SMITH, B. 1984. The implementation of procedurally reflective languages. In *Proc. of the ACM Symp. on Lisp and Functional Prog.*, ACM, New York, 331–347.

DEUTSCH, P. AND SCHIFFMAN, A. 1983. Efficient implementation of the Smalltalk-80 system. In *the* 10^{th} *ACM Symp. on Principles of Prog. Languages*, ACM, New York, 297–302.

FELLEISEN, M. 1988. The theory and practice of first-class prompts. In *the* 15^{th} *ACM Symp. on Principles of Prog. Languages*, ACM, New York, 180–190.

FRIEDMAN, D. AND WAND, M. 1984. Reification: Reflection without metaphysics. In *Proc. of the ACM Symp. on Lisp and Functional Programming*, ACM, New York, 348–355.

GELERNTER, D., JAGANNATHAN, S., AND LONDON, T. 1987. Environments as first-class objects. In *the* 14^{th} *ACM Symp. on Principle of Prog. Languages Conference*, ACM, New York.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Press. Reading, Mass.

HARPER, R. AND LILLIBRIDGE, M. 1994. A type-theoretic approach to higher-order modules with sharing. In *the* 19^{th} *ACM Symp. on Principles of Prog. Languages*, ACM, New York, 123–137.

HUDAK, P., PEYTON JONES, S. L., AND WADLER, P. 1992. Report on the functional programming language Haskell, version 1.2. *ACM SIGPLAN Not. 27,* 5 (May).

JAGANNATHAN, S. 1988. A programming language supporting first-class, parallel environments. Ph. D. thesis, MIT, Cambridge, Mass. (Also published as Tech. rep. LCS-TR 434.)

JOHNSSON, T. 1985. Lambda lifting: Transforming programs to recursive equations. In *Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science. Vol. 201. Springer-Verlag, New York.

KAMIN, S. 1988. Inheritance in Smalltalk-80: A denotational definition. In *the* 15^{th} *ACM Symp. on Principles of Prog. Languages*, ACM, New York, 80–87.

KOHLBECKER, E. AND WAND, M. 1987. Macro-by-example: Deriving syntactic transformations from their specifications. In *the* 14^{th} *ACM Symp. on Principles of Prog. Languages*, ACM, New York, 77–85.

KRANZ, D., KELSEY, R., REES, J., HUDAK, P., PHILBIN, J., AND ADAMS, N. 1986. Orbit: An optimizing compiler for Scheme. *ACM SIGPLAN Not. 21,* 7 (July), 219–233.

LAMPING, J. 1988. A unified system of parameterization for programming languages. In *Proc. of the ACM Symp. on Lisp and Functional Programming*, ACM, New York, 1–11.

LEE, S.-D. AND FRIEDMAN, D. 1993. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *the 20^{th} ACM Symp. on Principles of Prog. Languages*, ACM, New York, 479–492.

LIEBERMANN, H. 1986. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPSLA'86 Conference Proceedings, SIGPLAN Not. 21(11)*, (Nov. 1986), 214–223.

LISKOV, B., SYNDER, A., ATKINSON, R., AND SCHAFFERT, C. 1977. Abstraction mechanisms in CLU. *Commun. ACM 20,* 8 (Aug.), 564 – 576.

MACQUEEN, D. 1988. An implementation of standard ML modules. In *Proc. of the ACM Symp. on Lisp and Functional Programming*, ACM, New York, 212–223.

MACQUEEN, D. AND TOFTE, M. 1994. A semantics for higher-order functors. In *Proc. European Symp. on Programming (ESOP'94)*, Springer-Verlag, New York.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML.* MIT Press, Cambridge, Mass.

PEYTON-JONES, S. L. 1987. *The Implementation of Functional Programming Languages.* Prentice Hall.

REDDY, U. 1988. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. of the ACM Symp. on Lisp and Functional Programming*, ACM, New York, 289–297.

REES 1993. Another Module System for Scheme. Scheme48 documentation.

SMITH, B. 1982. Reflection and Semantics in a Procedural Language. Ph. D. thesis, MIT, Cambridge, Mass. (Also published as Tech. rep. LCS TR-272.)

STEELE JR., G. 1978. Rabbit: A Compiler for Scheme. Master's thesis, Massachusetts Institute of Technology, Cambridge, Mass.

STEELE JR., G. 1990. *Common Lisp: The Language.* Digital Press, Bedford, Mass.

STEELE JR., G. AND SUSSMAN, G. 1978. The art of the interpreter, or the modularity complex. Tech. rep. AI-TM 453, MIT Artificial Intelligence Laboratory, Cambridge, Mass.

STRACHEY, C. AND WADSWORTH, C. 1974. Continuations: A mathematical semantics for handling full jumps. Technical Report PRG-11, Programming Research Group, Oxford University, Oxford, U.K.

TOFTE, M. 1992. Principal signatures for higher-order program modules. In *the 19^{th} ACM Symp. on Principles of Prog. Languages*, ACM, New York, 189–199.

UNGAR, D. AND SMITH, R. 1987. SELF: The power of simplicity. In *OOPSLA'87 Conference Proceedings, SIGPLAN Notices 22(12)*, (Dec. 1987), 227–241.

UNGAR, D., SMITH, R. B., CHAMBERS, C., AND HÖLZLE, U. 1992. Object, message, and performance: How they co-exist in Self. *IEEE Comput. 25,* 10 (Oct.), 53–65.

U.S. Dept. of Defense 1982. *Reference Manual for the ADA Programming Language.* U.S. Dept. of Defense, Washington, D.C.

WAND, M. AND FRIEDMAN, D. P. 1988. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbol. Comput. 1,* 1 (June), 11–38.

WIRTH, N. 1985. *Programming in Modula-2.* Springer-Verlag, Berlin.

Received January 1991; revised February 1993 and August 1993; accepted October 1993.