Reflective Building Blocks for Modular Systems

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

Abstract

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 therefore worthwhile 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into an environment may cause useful program transformations which rely on static scoping (*e.g.,* α-conversion) to be no longer applicable.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 paper, we propose a treatment of environments and the mechanism by which they are reified and manipulated that addresses these concerns. The language described below permits environments to be reified into data objects, and objects 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.

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[1].

[1] For example, closures, packages, and modules are used for information hiding; objects and classes implement code reuse and sharing; records define a related collection of bindings, etc..

Since the formal definition of all namespace devices is given in terms of transformations 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 extant modularity structures (and to synthesize new ones) simply by transliterating their semantic definitions into the reflective base language of interest.

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 (*e.g.,* [1, 10, 22]), however, provide no mechanism to constrain the effect (or extent) of the reification or reflection process.

In this paper, 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 objects, and data objects to be reflected 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 reflective framework. We argue that the techniques and operators developed define a cohesive basis for building large-scale modular systems using reflective programming techniques.

2 The Language

The syntax and non-reflective dynamic semantics of Rascal is identical to Scheme[8] augmented with a primitive namespace constructor. The other novel aspects of the language deal with reflection.

Namespaces: A namespace is a collection of bindings manipulable as a data object. Superficially, namespaces resemble records; the expression:

```
(make-namespace (a 1) (b 2))
```

creates a namespace with two elements. However, namespaces are more properly regarded as reified images of semantic environments; thus, they may be used as arguments to environment reflection operations.

Public Variables: Variables may be denoted as "public" using the "†" annotation. Such variables may be captured by reification operations performed on the environment in which they are bound. Thus, the program fragment,

```
(lambda (x† y)
   body)
```

defines a procedure whose first argument binding may be captured by an environment reification operation that occurs within *body*; only public variables are targets for environment reification.

Visibility Constraints: Expressions may be annotated by "barriers" using the "|" annotation. Environment reification or reflection operations evaluated within such expressions will not capture any public variables that occur textually outside the barrier. Barriers hide bindings from meta-level environment operations. For example, in the program fragment,

```
(let ((x† e1))
  |(let ((y† e2)
         (z e3))
     body))
```

only y is a target for environment reification in *body* because of the barrier annotating the inner let (assuming that *body* declares no other public variables). We think of barriers as the environment analogue of *prompts*[9] used to constrain the effect of reification operations performed on continuations.

Reflection over Data Structures: *Namespace* data objects may be installed as meta-level environments using the reflect operation. The expression:

```
(reflect e1 e2)
```

creates a new environment that is defined as the functional composition of the operator's evaluation environment (call it ρ_b) and the label bindings defined by e_1. Thus, a free name N referenced in e_2 acquires meaning based on its binding-value in the reflected image of the namespace yielded by e_1 (call this environment ρ_r); if N is not bound in ρ_r, its binding-value is determined from ρ_b.

The reflect operator's evaluation environment is the environment extant upto the closest enclosing barrier. For example, the following code fragment yields an error if r does not define a binding for y in all calls to f:

```
(define f (let ((y 1))
            |(let ((x 2))
               (lambda (r)
                 (+ (reflect r x)
                    (reflect r y))))))
```

Thus, note that

```
|(reflect R var)
```

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.

Environment Customization: The name-lookup rule for an environment can be customized. A *binding procedure* is a meta-level object that given an environment, rho, and an identifier, id, returns an environment that computes a binding value for its argument using bindings defined in rho. The reified representation of id is given as a symbol that acquires meaning when applied to an environment. Thus, we can write:

```
(define (read-only obj)
  (let ((read-only-binder
         (let ((original (copy |(reflect obj x))))
           (binder (rho id)
             (if (and (equal? id 'x)
                      (not (equal? (rho id) original)))
                 (error "variable x has changed")
                 original)))))
    (reflect Env Proc read-only-binder)
```

to make the binding for variable x yielded by reflecting over record Env be read only in the definition of *Proc*.

Reification over Environments: Environments may be reified using the reify operation. In the general case, the expression, "(reify)", returns a namespace data object containing a binding for every public variable in the current environment up to the lexically closest enclosing barrier.

When given a closure as its argument, reify returns the (public) free variables defined within the closure; thus,

```
|(let ((x†  E)
       (y  E1))
   (let ((f (lambda () (x y))))
     (reify f)))
```

returns a record containing the label x bound to the closure yielded by E; y is not included in the record object returned since it is not declared public. The barrier annotating the outer let guarantees that no public bindings defined outside the let are captured by reification operations which occur inside.

2.1 Formal Semantics

We give a formal semantics for Rascal's reflection constructs in Fig. 1. The semantics is defined for a pure subset of the language; the value domain is restricted to integers, Booleans, symbols, namespaces, and procedures. The primitive environment, $\rho_\perp$, maps its input to *undef*. The domain of a binding environment ρ, $Dom(\rho)$, is a set of identifiers such that $\forall x \in Dom(\rho)$, $\rho(x) \neq undef$. $\rho[x \mapsto v]$ defines an extension of environment ρ with a binding of x to v. $\rho_1[\rho_2]$ denotes functional composition of environments ρ_1 and ρ_2.

2.2 Rationale

Barriers and public variables provide a basis for preserving locality and information hiding even in the presence of general environment reification. Only public variables are targets for reification; barriers delimit the extent of a reification operation. Reflection over reified environments permits bindings to be selectively shared across separate evaluation contexts. Both properties are crucial for modularity.

Syntactic Domains

$$\begin{array}{lll}
c & \in \textbf{Const} & \text{Constants} \\
x, y & \in \textbf{Var} & \text{Variables} \\
e & \in \textbf{Exp} & \text{Expressions}
\end{array}$$

Semantic Domains

$$\begin{array}{ll}
b & \in \textbf{Values} = \textbf{Int} + \textbf{Bool} + \textbf{Namespace} + \textbf{Symbol} + \textbf{ProcEnv} \\
\rho, \rho^{\lambda}, \rho^{\dagger}, \rho^{|} & \in \textbf{Env} = \textbf{Var} \rightarrow \textbf{Values} + undef \\
r & \in \textbf{Namespace} = \textbf{Var} \rightarrow \textbf{Values} \\
f & \in \textbf{Fun} = \textbf{Values}^{n} \rightarrow \textbf{Values} \\
\theta & \in \textbf{ProcEnv} = \textbf{Env} \times \textbf{Fun}
\end{array}$$

Valuation Functions

$$\begin{array}{ll}
\mathcal{E} & \textbf{Exp} \rightarrow \textbf{Env} \rightarrow \textbf{Env} \rightarrow \textbf{Env} \rightarrow \textbf{Values} \\
\mathcal{K} & \textbf{Const} \rightarrow \textbf{Values}
\end{array}$$

$$\mathcal{E}[\![(\texttt{make}-\texttt{namespace})]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} = r_{\perp}$$

$$\begin{array}{l}
\mathcal{E}[\![(\texttt{lambda}\ (x_1\ x_2\ \ldots\ x_n)\ e)]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} = \\
\quad \text{let } \{y_1, y_2, \ldots, y_m\} \text{ be public free variables in } (\texttt{lambda}\ (x_1\ x_2\ \ldots x_n)\ e) \\
\qquad \{x_i,\ \ldots,\ x_j\} \text{ be public in } \{x_1, \ldots, x_n\} \\
\quad \text{in } < \rho_{\perp}[y_1 \mapsto \rho^{\dagger}(y_1), \ldots, y_m \mapsto \rho^{\dagger}(y_m)], \\
\qquad\quad (\lambda\ (b_1\ b_2\ \ldots b_n) \\
\qquad\qquad \mathcal{E}[\![e]\!]\rho^{\lambda}[x_1 \mapsto b_1, \ldots, x_n \mapsto b_n]\ \rho^{\dagger}[x_i \mapsto b_i, \ldots, x_j \mapsto b_j]\ \rho^{|}[x_1 \mapsto b_1, \ldots, x_n \mapsto b_n])
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![(e_1\ e_2\ \ldots\ e_n)]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} = \\
\quad \text{let } < \rho, f > = \mathcal{E}[\![e_1]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|}, \\
\qquad b_2 = \mathcal{E}[\![e_2]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|}, \\
\qquad\quad \vdots \\
\qquad b_n = \mathcal{E}[\![e_n]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|}, \\
\quad \text{in } f(b_2\ \ldots\ b_n)
\end{array}$$

$$\mathcal{E}[\![|e]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} = \mathcal{E}[\![e]\!]\rho^{\lambda}\ \rho_{\perp}\ \rho_{\perp}$$

$$\begin{array}{l}
\mathcal{E}[\![(\texttt{reify})]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} = \\
\quad r_{\perp}[x_1 \mapsto \rho^{\lambda}(x_1)\ldots x_n \mapsto \rho^{\lambda}(x_n)] \text{ for } \{x_1, \ldots, x_n\} \text{ in } Dom(\rho^{\dagger})
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![(\texttt{reify}\ e)]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} = \\
\quad \text{let } < \rho, f > = \mathcal{E}[\![e]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} \\
\quad \text{in } r_{\perp}[x_1 \mapsto \rho(x_1)\ldots x_n \mapsto \rho(x_n)] \text{ for } \{x_1, \ldots, x_n\} \text{ in } Dom(\rho)
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![(\texttt{binder}\ (x_1\ x_2)\ e)]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} = \\
\quad (\lambda\ (b_1)\ (\lambda\ (x_2)\mathcal{E}[\![e]\!]\rho^{\lambda}[x_1 \mapsto b_1]))
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![(\texttt{reflect}\ e_1\ e_2)]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} = \\
\quad \text{let } r = \mathcal{E}[\![e_1]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} \\
\quad \text{in } [\![e_2]\!]\rho^{|}[r]\ \rho^{\dagger}\ \rho^{|}
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![(\texttt{reflect}\ e_1\ e_2\ e_3)]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} = \\
\quad \text{let } r = \mathcal{E}[\![e_1]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} \\
\qquad f = \mathcal{E}[\![e_3]\!]\rho^{\lambda}\ \rho^{\dagger}\ \rho^{|} \\
\quad \text{in } \mathcal{E}[\![e_2]\!]\rho^{|}[f(r)]\ \rho^{\dagger}\ \rho^{|}
\end{array}$$

Figure 1: Semantics for a subset of Rascal.

Binding procedures permit customization of a given environment; this allows programmers the flexibility to write daemon procedures that are implicitly invoked whenever a variable is accessed. Such procedures help transparently enforce and strengthen modularity constraints.

Closures are the only structures in Rascal that encapsulate implicit localized environments. The ability to capture public bindings defined in closure maintained environments (via reification) permits lexically disjoint expressions to share bindings; we argue in the following sections that reification of this sort provides precisely the operational requirements for general modular, object-based programming.

There are only two environment (or namespace) manipulating objects in our language: procedures which construct implicit environments captured within closures that may be reified, and namespaces which construct manifest environments that may be reflected. These two types of environments used in conjunction with operations for enhancing and restricting visibility provide the opportunity for supporting various forms of modularity within a unified and self-consistent system.

3 Applications

3.1 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 the lexical binding protocol – since users don't 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 expressivity of the language, and it often necessitates extended dialects to provide either *ad hoc* constructs to realize other binding disciplines (*e.g.,* the `fluid-let`[1] construct 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[2, 4, 3]).

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 λ-expressions (or individual variables) without necessitating any alteration to the base language semantics.

For example, a late-binding binding procedure,

$(\texttt{lambda}_d\ \texttt{(x1 x2 ... xn)}\ \mathit{Exp})$

is equivalent to:

```
|(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. Free variables in the body of the abstraction are evaluated relative to their binding value in `Env`. The barrier preceding the definition ensures that free names referenced in the body of the procedure not captured in the dynamic environment result in an error condition.

An application of such a procedure:

$(e_f\ \texttt{e1 ... en})_d$

is equivalent to:

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

`Env` and `Proc` are assumed to be fresh variables. Note that if a 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.

3.2 Object-Based Programming

Object-based programming is an important domain for meta-programming. The central observation that relates reflection of the kind described here with object-based computing is that objects fundamentally define localized namespaces; inheritance and delegation operations (*e.g.,* [12, 18]) permit manipulation of these 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[2] or Common Lisp[23] that are statically scoped. Support for objects in these systems however involve significant extensions or alterations to a simple language kernel. More significantly, it is non-trivial 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 simple 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 method and instance variables shared among subclasses). Environment reification contributes to a programming methodology that naturally supports such constructions; it does so by permitting environments to be shared across distinct evaluation contexts.

Consider the following simple example: a `color point` is a specialized variant of a `point` object. The point interface provides operations to move and print points, and to test if two points are equal. A color point, in addition to these operations, associates a color with every coordinate. One possible implementation of these two objects is given in Fig. 2.

A `point` abstraction defines an accessor method (`self`) and generic methods. The object returned by instantiating this abstraction contains these methods along with the instance variables denoting coordinates. A `color point` abstraction inherits `move` and `equal` methods from `point` but defines a new `print` routine sensitive to colors; this procedure shares functionality with the procedure method for simple points. The closure reification in its definition captures all public bindings found in the default `print` routine. Points and colored points are structurally identical except for the "super" definition in the latter. `Self` and `super` bindings are ordinary Rascal bindings; there is no special semantics accorded these operations.

The advantage of manipulating environments explicitly in this example lies in added conceptual simplicity and modularity. All class definitions have the same structure: an instance of a point or a color point is defined by composing method definitions, instance variables, and all sub-class definitions into a single record definition. Definitions needed to implement a method, but which are not part of the object's specification need not be explicitly exported. For example, the `print` method for points is presumably closed over output streams, device driver procedures, etc.; these definitions are not defined as part of `point`'s interface. Objects which define new `print` methods can share these common routines transparently *without* violating the specification for points by reifying over `point`'s closure.

This implementation shares characteristics common to both class-based[6, 13] as well as delegation-based[18, 24] systems. As in a class-based system, distinct instances of `points` and `color-points` remain structurally identical, and share common method definitions. Reification over environments permit prototype objects to be constructed as well. There is no *a priori* constraint that requires new instances of points or color points to inherit the method definitions found in their respective generators. For example, we can create an instance of a color point, "clone" it, and then add (or remove) functionality to the cloned instance without requiring modification to the parent class[25]. For example, the following code fragment:

```
(let* ((my-color-point
        (make-color-point default-object x y color))
       (scaled-move†
         (lambda (self scale dx dy)
           (reflect
             (reify (select my-color-point move))
             move point by scale factor)))
  (compose my-color-point scaled-move))
```

defines a variant of a color point object that shares instance variables and method definitions with `my-color-point`, but defines a new method that permits points to be scaled by a given factor; this method, in turn, shares functionality with the default `move` method for ordinary points.

Using binding procedures, we can specialize the behavior of methods over individual instances. For example, different instances of a color point may wish to have points printed differently; one instance may wish to have points printed on a greyscale, another may require points printed within a specific color grid, etc.. To accomodate such flexibility in the absence of binding procedures, we need either to augment the implementation of `print` to handle all these cases, or build specialized classes that duplicate `print`'s basic functionality.

With the use of binding procedures, we can support such functionality seamlessly. Fig. 3 defines a modified version of a color point that uses a binding procedure, and an instance of this class that converts colors to either black or white without requiring alteration to `print`'s basic definition.

To summarize: In a reflective model of the kind described here, the notion of "self" is implemented using record composition, reflection and reification over environments. Classes are namespace generators and a class hierarchy is built by composing new instances of namespaces generated from a set of super-classes; 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.

4 Pragmatics

As a matter of practical convenience, `reflect` and `reify` expressions are clusmy vehicles in which to express inheritance paradigms. Based on the examples given in the previous sections, however, it is clear that there are patterns of usage of these operators that capture common inheritance and delegation-style functionality.

For example, the semantics of the Smalltalk "*self*" pseudo-variable is defined via namespace composition and reification thus:

```
self ≡
(lambda () (set! self (compose (reify) obj)))
```

where `(reify)` captures the local environnment (*e.g.,* instance variables and methods) of the object being defined, and `obj` is the record representation of the caller's environment.

Similarly, to create an instance of a class A that is a superclass of B, we write:

(*A* `self` *args*)

where `self` is the self object denoting B.

We can build syntactic sugar that obviates the need for programmers to refer to the underlying environment structure used to express inheritance or delegation strategies. We envision a library of such macros; Rascal programmers need only have knowledge of the macro interface in order to write programs that have object-based semantics. Thus, while reflection can be used effectively to specify different types of inheritance protocols, they can be effectively subsumed by straightforward syntactic abstractions. Understanding object-based programming techniques in terms of syntactic transformations over environment manipulating expressions is an important property of this model[16].

```
|(define (make-point obj x† y†)
   (letrec ((self (lambda () (set! self (compose (reify) obj))))
            (move† (method (dx dy)
               (set! x (+ x dx))
               (set! y (+ y dy))))
            (print† (method () ...))
            (equal?† (method (p)
              (and (equal? x |(reflect p x))
                   (equal? y |(reflect p y))))))
     (self)))

|(define (make-color-point obj x† y† color†)
   (letrec ((self  (lambda () (set! self  (compose (reify) obj))))
            (super (lambda () (make-point self x y)))
            (print† (method ()
              (reflect (reify (select super print))
                print color point using public procedures defined by point's print procedure))))
     (self)
     (compose (super) self)))

(define-syntax method
  syntax-rules ()
    ((method (arg1 arg2 ...) body1 body2 ...)
     (lambda (self arg1 arg2 ...)
        (reflect self
           (begin body1 body2 ...)))))
```

Figure 2: Implementation of point and color point abstractions. Namespaces are joined using the `compose` procedure: `(compose R1 R2)` returns a namespace in which bindings defined in `R2` shadow those defined in `R1`. `Method` is syntax that expands to a procedure which evaluates its body relative to the environment image of `self`.

```
|(define (make-color-point obj x† y† color†)
   (letrec ((self  (lambda () (set! self  (compose (reify) obj))))
            (super (lambda () (make-point self x y)))
            (print† (method ()
                    (reflect (reify (select super print))
                       print color point using public procedures defined by point's print procedure
                       (send self print-binder)))))
     (self)
     (compose (super) self)))

|(define (grey-scale-point obj x y color)
   (let* ((print-binder† (binder (rho id)
                         (if (equal? id 'color)
                             (compute-grey-scale (rho id))
                             (rho id))))
          (generic-color-point (make-color-point (compose (reify) obj)
                                                 x y color))
          (print† (lambda () (send generic-color-point print))))
      (compose generic-color-point (reify))))

(define-syntax send
  (syntax-rules ()
    ((send (obj message args ...))
     (apply (reflect obj message) obj args))))
```

Figure 3: A modified implementation of color point that uses a binding procedure

4.1 Type Inference

Reflection over namespaces and reification over meta-level environments raise other interesting implementation issues. To build an efficient implementation of a system such as the one proposed here requires techniques to associate statically names with their potential binding values. The problem is a non-trivial one since namespaces can be reflected and used as environments in any expression context.

We have built a static type system for the functional subset of Rascal that implements this analysis[15]. The inference system is patterned after the Hindley-Milner type system used to define a polymorphic type inference algorithm for ML[7, 19]. Space limitations prevent a detailed description of this approach here, but suffice it to say that the analysis is capable of making inferences of the form:

> If a namespace creating expression R *always* defines x, then free occurrences of x found within the scope of the reflective image of R will always refer to the binding value of x in R.

or

> If a namespace creating expression R *never* defines x, then free occurrences of x found within the scope of the reflective image of R will always refer to the binding value of x in its lexical environment.

5 Comparison to Related Work

Pebble[5] and certain dialects of Scheme[1] are two languages that also support environment reification. Pebble 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.

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

In certain dialects of Scheme[1, 20], users are allowed to build customized environment structures via a special environment constructor operation. 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.

The only operation allowable on manifest Scheme environments, however, is `eval`. Thus, unlike Rascal, one cannot reify the current environment (or the environment component of a closure) to get a concrete representation of the bindings it contains, to build binding procedures to alter the name-lookup protocol, to constrain or extend name visibility, or to reflect ordinary record structures into environments.

The treatment of environments in 3-Lisp[21, 22] or Brown[10, 26] is similar to their treatment in Rascal in two important respects: environments can be reified into concrete structures (*e.g.,* procedures in Brown, and lists in 3-Lisp), and data structures can be reflected into environments. Neither Brown nor 3-Lisp, however, fully examined the implications of such reflective capabilities for building modular systems; thus, they do not provide barrier annotations, make no distinction between ordinary variables and those which can be captured by reification, and do not permit environments maintained within closures to be captured. 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.

Symmetric Lisp[11, 14] 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 or closures, nor does it provide customization of binding protocols.

Finally, Lamping[17] 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.

References

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

[2] Norman Adams and Jonathan Rees. Object-Oriented Programming in Scheme. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 277–288, 1988.

[3] Daniel Bobrow, Linda DiMichiel, Richard Gabriel, Sonya Keene, Gregor Kicczales, and David Moon. Common Lisp Object System Specification 1. Programmer Interface Concepts. *Lisp and Symbolic Computation*, pages 245–298, January 1989.

[4] Daniel Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops:Merging Lisp and Object-Oriented Programming. In *OOPSLA'86 Conference Proceedings*, pages 17–30, September 1986.

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

[6] O.J. Dahl, B. Myhruhaug, and K. Nygaard. The Simula67 Base Common Base Language. Technical report, Norwegien Computing Center, 1970.

[7] Luis Damas and Robin Milner. Principle Type-schemes for Functional Programs. In 9^{th} *ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[8] William Clinger *et. al.* The Revised Revised Revised Report on Scheme or An UnCommon Lisp. Technical Report AI-TM 848, MIT Artificial Intelligence Laboratory, 1985.

[9] Matthias Felleisen. The Theory and Practice of First-Class Prompts. In 15^{th} *ACM Symposium on Principles of Programming Languages*, pages 180–190, 1988.

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

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

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

[13] Daniel Ingalls. The Smalltalk-76 Programming System: Design and Implementation. In *Fifth ACM Symposium on Principles of Programming Languages Conf.*, pages 9–16, January 1978.

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

[15] Suresh Jagannathan. Reflective building blocks for modular systems. Technical Report 92-049-3-0050-5, NEC Research Institute, 1992.

[16] Suresh Jagannathan and Gul Agha. A Reflective Model of Inheritance. In *ECOOP'92 European Conference on Object-Oriented Programming*, pages 350–372, 1992. Published as Springer LNCS 615.

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

[18] Henry Liebermann. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA'86 Conference Proceedings*, pages 214–223, 1986. Published as SIGPLAN Notice 21(11), November 1986.

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

[20] Jonathan A. Rees and Norman I. Adams. T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 114–122, 1982.

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

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

[23] Guy Steele Jr. *Common Lisp: The Language, Second Edition.* Digital Press, 1990.

[24] Lynn Stein. Delegation is Inheritance. In *OOPSLA'87 Conference Proceedings*, pages 138–146, 1987. Published as SIGPLAN Notices 22(12), December, 1987.

[25] David Ungar and Randall Smith. SELF: The Power of Simplicity. In *OOPSLA'87 Conference Proceedings*, pages 227–241, 1987. Published as SIGPLAN Notices 22(12), December, 1987.

[26] Mitchell Wand and Daniel Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 298–307, 1986.