A Reflective Model of Inheritance

Suresh Jagannathan[1] *and Gul Agha*[2]

[1] Computer Science Research, NEC Research Institute, 4 Independence Way, Princeton, NJ 08540, *suresh@research.nj.nec.com*
[2] Dept. of Computer Science, 1304 W. Springfield Ave., University of Illinois, Urbana, IL 61801, *agha@cs.uiuc.edu****

1 Introduction

Inheritance is widely regarded as a central feature in modern object-oriented programming. Despite its perceived importance, however, there is still no universal consensus on the definitions or mechanisms which should be used to support it. Current proposals consider it either as an operational technique for code sharing and reuse, or as a structuring mechanism for reasoning about programs. Depending on the motivation for its introduction, the techniques to support inheritance that are incorporated in language designs often appear to have little (if any) semantic traits in common.

Regardless of how inheritance is implemented or used, it is clear that a central issue in any object-oriented language is namespace management; in this sense, inheritance maybe regarded conceptually as a tool for conserving names within a program. Formal definitions of programming languages typically refer to namespaces as *environments*, and represent them in terms of finite sets (or functions) that bind (or map) program variables to values. Environments are usually built and maintained by an abstract interpreter that implements the language's semantics; the structure of this interpreter prohibits programmers from gaining direct access to environments. Thus, the rules governing the definition and management of namespaces are invariably "hard-wired" as part of the language definition. As a result, it is often problematic to manipulate namespaces in ways not originally prescribed by the language designers. This restriction has significant ramifications for the design of object-oriented languages: to build a variation of an inheritance structure or to define a different one altogether in effect requires implementing a new language or constructing a new interpreter sensitive to the desired requirements.

In this paper, we present an alternative treatment of namespace construction and manipulation. The *reflective* model is based on a semantic transformation technique that

*** The second authors' work has been made possible by support provided by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195).

† Appeared in the Proceedings of the European Conference on Object-Oriented Programming, June 1992. Published as Springer-Verlag LNCS 615.

provides flexible *mechanisms* for managing namespaces. We argue that given the ability to manipulate environments directly, a variety of different object-oriented paradigms can be realized within a unified and simple framework. Starting from a kernel language whose foundation is the simply typed λ-calculus, we develop a small collection of environment manipulating primitives. These primitives provide an expressive platform to express a number of inheritance-related abstractions.

Our model is distinguished from other efforts that provide a formal semantic treatment of inheritance and delegation[11, 19]. In these systems, objects are represented as records, with fixpoints and record composition used to realize late-binding. Our work generalizes this approach in some important respects. Most notably, we define specific linguistic mechanisms to express inheritance and delegation that are couched in terms of *reflective* operations over environments. In the presence of reflection, pseudo-variables such as "*super*" or "*self*" are now interpreted as ordinary data objects. Message passing and method dispatch are simple function applications that evaluate relative to a user-generated environment. Inheritance is not a fundamental component in a reflective language. Its semantics is given in terms of composition and source-to-source transformation on environment manipulating operations.

While the bulk of the paper is confined to a foundational description of the model and its expressivity gains, we also address the question of developing 'syntactic sugar' for abstractions which capture common inheritance strategies. We argue that inheritance and delegation paradigms can be succinctly described in terms of syntactic program transformations once a framework for managing environments is developed.

2 The Model

We can think of an environment as one piece of context or state information that is manipulated by the abstract interpreter (or virtual machine) that defines a language's semantics. In most languages, state information is hidden from the programmer. Thus, it is often not possible to write expressions that directly access and manipulate context information built during the evaluation of a program.

The model described here permits precisely this kind of functionality. The essence of the model revolves around two transformation operators. The first is a "reflective" operator that allows data objects to be treated as though they were binding environments. Any data structure that binds names to values (*e.g.,* records, hash tables, association-lists, etc.) can be transformed into a scope within which other expressions can evaluate. In effect, reflection permits the dynamic construction and injection of new scopes into a program. An object which can be treated as a scope defines a binding environment.

The operational inverse of reflection is "reification". Just as record-like structures can be transformed into environments, environments can be captured and transformed into data objects. Thus, environments within this model have a well-defined concrete representation.

Figure 1 depicts the interaction between a language interpreter and the transformation operators described.

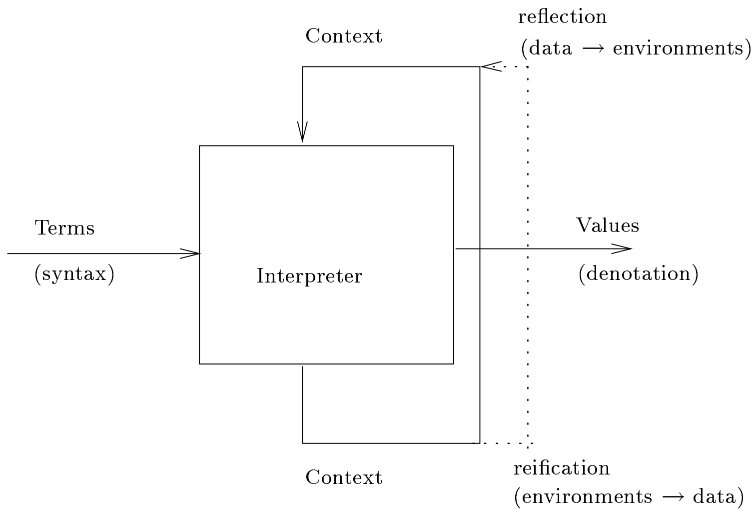

Fig. 1. The reflective model permits transformations between environments and ordinary data objects.

Given a way to explicitly manipulate environments, inheritance based abstractions are implemented in terms of creation, composition, and reification of environments. In this regard, the model differs from traditional class-based inheritance systems (*e.g.,* Simula[12] or Smalltalk[13]) in several important respects. First, it provides a consistent semantics of all elements which comprise a given object; there are no specific rules governing the manipulation of elements that serve as instance variables and those that serve as methods. Second, and more importantly, the model imposes no specific policy on how an inheritance hierarchy should be constructed. Decisions regarding the structure of a class hierarchy, the notion of *self* or *super*, early or late binding of free variables occurring in class definitions, or the semantics of message passing are not built-in characteristics of languages built on this model. Inheritance is viewed as a programming method, not a fundamental property of a programming language.

The model is also distinguished from delegation based systems[22, 31] insofar as (a) objects are implemented in terms of ordinary function abstractions and data structures, (b) there is no *a priori* static structure for an object that indicates its parent in the delegation hierarchy, and (c) objects instantiated from classes can be freely intermixed with objects instantiated via delegation.

The next section describes a reflective kernel language. A series of applications relating the utility of the model to object-based paradigms is sketched Section 5. Simple syntactic sugar that abstracts the complexity of manipulating environments is presented in Section 6; different inheritance mechanisms can be specified succinctly given the macro facility described. Although comparisons to related work are provided throughout the paper, a summary is given in Section 7.

3 A Kernel Language

To make our discussion more concrete, we define a kernel language called $\mathcal{L}$ that we will use in the examples developed in later sections. The non-reflective operators in $\mathcal{L}$ form a strongly-typed, non-strict (*i.e.,* lazy), higher-order lexically-scoped functional language. While austere, $\mathcal{L}$ provides all the essential ingredients necessary to building various inheritance protocols.

$\mathcal{L}$'s kernel term set is defined by the grammar given in figure 2. We use x to range over identifiers, s to range over strings, n to range over the natural numbers and b to range over Booleans. We describe the other constructs in the language below[5].

$$
\begin{array}{lll}
\mathrm{E} & ::= & x \mid n \mid s \mid b \mid \\
 & & \texttt{!}\,\mathrm{E} \mid \\
 & & (\lambda\ (\{x^{\dagger} \mid x\}^{*})\ \mathrm{E}) \mid (\mathrm{E}\ \{\mathrm{E}\}^{*}) \mid \\
 & & \texttt{letrec}\ \{\{x\{^{\dagger}\} = \mathrm{E}\}^{+}\ \mathrm{in}\ \mathrm{E} \mid \\
 & & \mathrm{E} \rightarrow \mathrm{E};\mathrm{E} \mid \mathrm{E} \rightarrow \mathrm{E} \mid \\
 & & [\ \{x = \mathrm{E}\}^{*}\ ] \mid \mathrm{E}.x \mid (\bullet\ \mathrm{E}\ \mathrm{E}) \mid \\
 & & (\texttt{reflect}\ \mathrm{E}\ \texttt{in}\ \mathrm{E}) \mid (\texttt{reify}\ \{\mathrm{E}\})
\end{array}
$$

Fig. 2. Grammar for $\mathcal{L}$.

We discuss the significance of the † annotation on `letrec-` and λ-bound variables in Section 3.3, and the "`!`" (read "prompt") prefix in Section 3.3.

Abstractions are introduced using λ notation; conditionals are written using $\rightarrow$; application is expressed by juxtaposition of the function being applied with its arguments. Recursion is expressed using `letrec`.

3.1 Records

Records are non-strict finite associations of labels to values. The constituent expressions in a record are evaluated relative to the record's evaluation environment.

The value of a record field can be retrieved using the "." operator: if r is a record, then evaluating $r.x$ returns the binding value of x as defined in r.

We provide one other operation over records. Let r_1 and r_2 be two records and let $Dom(r)$ be the set of names defined within record r. The join or composition of r_1 and r_2 (written $(\bullet\ r_1\ r_2)$) is now defined as follows:

[5] Besides these basic syntactic forms, we introduce various syntactic extensions (or abbreviations) throughout the paper; these extensions are best thought of as macros that expand into elements of the base term set.

$$(\bullet\ r_1\ r_2).x = \begin{cases} r_2.x \text{ if } x \in\ Dom(r2) \\ r_1.x \text{ otherwise} \end{cases}$$

3.2 Reflection

Record objects are transformed into environments using the `reflect` operator. This operator permits record labels to be treated as program variables. Given an expression e_1 that yields a record, we evaluate an expression e_2 in the context of the bindings defined by e_1 by writing:

```
(reflect e1 in e2)
```

The record object r yielded by the evaluation of e_1 is transformed into an environment that contains a binding for each label found in r. The binding value of a *potentially* free identifier found in the body of e_2 *not* defined by this environment is resolved within the current evaluation environment. We give a precise definition for potential free variables in Fig. 3.

Since records are non-strict, the environment image of a record captured using `reflect` may consist of unevaluated (closed) bindings. Thus, expressions in e_2 that access a binding B in r force the evaluation of the deferred expression associated with B's binding value.

The reflect operator is similar to the dot operator discussed by Gordon in [14] and to the *let* construct found in Pebble[6], a higher-order language that treats bindings as first-class values.

3.3 Reification

The transformation of an environment into a data object is accomplished using the `reify` operator. In its most simple form, `reify` takes no arguments, and when evaluated returns a record containing a binding for each †-suffixed variable found in its evaluation environment. Annotating a `letrec`- or λ-bound variable with a † marks that variable as *public* insofar as its binding can be captured and exported using a reification operation. Thus, the result of evaluating the expression[6]:

```
let a† = 1,
    b† = 2,
    c  = 3
in (reify)
```

[6] The syntactic form:

$$\texttt{let}\ x_1\ =\ e_1,\ x_2\ =\ e_2,\ \ldots,\ x_n\ =\ e_n\ \mathit{in}\ E$$

is equivalent to:

$$((\lambda(x_1\ x_2\ \ldots\ x_n)\ E)\ e_1\ e_2\ \ldots\ e_n)$$

is a record binding `a` to `1` and `b` to `2`. The binding value of `c` is not captured by evaluating `reify`. Unlike other language definitions that permit the explicit capture of bindings or environments[1, 6], the semantics of `reify` permits selective capture of bindings found in its environment. This capability is crucial to maintain object encapsulation and information hiding.

`Reify` permits any local binding environment to be transformed into a module; the names visible at the interface of a `reify`-generated module are precisely the public bindings found within its evaluation environment.

Since the environment image of a record used as the argument to `reflect` never contains public bindings (record labels cannot be suffixed with "†"), `reify` is insensitive to the bindings injected by `reflect`. Thus,

(reflect E in (reify)) $\equiv$ (reify)

Reification of Closures. In its more general form, `reify` takes a single argument. This argument must evaluate to a closure. The value yielded by `reify` in this case is a record containing the binding values of all †-suffixed potential free variables that occur in the body of the abstraction associated with the closure.

Definition 1. Let x and y range over identifiers and let E range over expressions. We define the notion of a potential free variable inductively as follows: (The abbreviation "x *PF* E" reads "x occurs potentially free in E.")

- x *PF* $x^{\dagger}$.
- x *PF* $(E_1\ E_2)$ if x *PF* E_1 and x *PF* E_2.
- x *PF* $(\lambda\ (y_1^{\dagger})\ E)$ if $x \neq y_1$ and x *PF* E.
- x *PF* `letrec` $y^{\dagger} = E_1$ `in` E_2 if x *PF* E_1 or $x \neq y$ and x *PF* E_2.
- x *PF* $[x_1 = E_1,\ x_2 = E_2,\ \ldots,\ x_n = E_n]$ if it occurs potentially free in any of the E_i, $1 \leq i \leq n$.
- x *PF* $E_1.y$ if x *PF* E_1.
- x *PF* $(\bullet\ E_1\ E_2)$ if x *PF* E_1 or x *PF* E_2.
- x *PF* (`reflect` E_1 `in` E_2) if x *PF* E_1 or x *PF* E_2.
- x *PF* (`reify` E) if x *PF* E.
- x *PF* $E_1 \rightarrow E_2; E_3$ if x *PF* E_i, $1 \leq i \leq 3$.

Fig. 3. Definition of *potential free variables* in $\mathcal{L}$. This definition is weaker than the definition of free variables found in *e.g.,* the λ-calculus. If x is free in E, then x is a bound variable in $(\lambda\ (x)\ E)$; on the other hand, this need not be the case if x were potentially free in E. Suppose E is of the form (reflect E_1 in x). If E_1 defines a field for x, then x will not be a bound variable in $(\lambda\ (x)\ E)$ since its binding value is determined from the record object denoted by E_1.

Thus, evaluating the expression:

```
letrec x† = 1
```

```
        f = (λ (y) (reflect y in x))
in  (reify f)
```

yields the record `[x = 1]`.

The only construct in $\mathcal{L}$ that manipulates binding environments implicitly is function abstraction: a lexical closure contains an environment that binds each free variable occurring in the body of the abstraction to its binding value in the function's lexical environment. The general form of `reify` permits certain elements of this environment to be accessed and manipulated freely.

Prompts. The visible scope within which a reification operation evaluates is delimited using a prompt facility. Any expression, E, may be prefixed by a prompt (written " `!` "). All public variables found in the lexical environment outside of the contour specified by the prompt are hidden from any reification subexpression of E. Thus, assuming primitive operations (*e.g.,* `+`,`-`,`*`) are not public, evaluating the expression,

```
let a† = 1
in
    !let b† = 2,
         c† = (+ a 10)
     in  let d = (λ (f) (- (* f (+ b c)) a))
         in   (reify d)
```

yields (when fully evaluated):

```
[ b = 2, c = 11 ]
```

The binding value of `a` is not captured in the record representation of `d`'s closure because of the prompt prefixing the inner `let`.

4 Formal Semantics

The semantics of $\mathcal{L}$ is given in terms of a set of Plotkin-style rewrite rules[25]. The semantics of reflection is captured in three rules that manipulate environments. Applications augment the current evaluation environment with a binding for the λ-bound variable defined by the abstraction; prompts remove bindings from an evaluation environment; public variables project bindings outside the environment in which they were defined. The semantics of expression evaluation is given relative to these three categories; each category is represented as a binding environment:

- ρ^{λ} defines the binding environment used to evaluate non-reflective operations (*i.e.,* abstraction, application, letrec, conditionals, records, etc.)
- $\rho^{!}$ defines the binding environment used to evaluate prompts.
- $\rho^{\dagger}$ defines the binding environment used to evaluate `reify` expressions.

The empty binding environment $\rho_{\perp}$ maps its input to the special symbol *undef.* The domain of a binding environment ρ, written $Dom(\rho)$, is a set of identifiers such that $\forall x \in Dom(\rho)$, $\rho(x) \neq$ *undef.*

Definition 2.

- The special symbol *undef* is a value.
- Constants (*e.g.,* integers, Booleans, and strings) are values.
- A *closure* is a value. A closure is a pair $(< \rho^{\lambda}, \rho^{!}, \rho^{\dagger} >, e)$ that associates an expression e with the binding environment triple used to evaluate it. Closures are used to specify the semantics of non-strictness and abstraction.
- A binding environment is a value.

Fig. 4. Definition of *value* in $\mathcal{L}$.

In general, we shall use the notation:

$$\rho[x_1 \mapsto v_1, x_2 \mapsto \ v_2, \ldots, x_k \mapsto \ v_k]$$

to indicate the environment that maps x_i to v_i, $i = 1, 2, \ldots, k$ and any other identifier y to $\rho(y)$. Similarly, we write $\rho[\rho']$ to indicate the binding environment yielded by composing ρ with ρ'; a binding for variable x found in ρ' supersedes its binding value in ρ provided that $x \in \ Dom(\rho')$.

4.1 Semantic Rules

To express the fact that expression e evaluates (or "reduces") to expression e' with respect to ρ^{λ},$\rho^{!}$, and $\rho^{\dagger}$, we write:

$$\rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ e \Longrightarrow \ e'$$

We omit the rules for conditionals, record selection and composition; their definitions are standard. The definition of `letrec` follows from the definition of application and the assumption of a least fixpoint operator[4]; its definition is omitted as well.

(Constants)

$$\rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ v \ \Longrightarrow \ v$$

(Identifiers)

$$\frac{\begin{array}{c}\rho^{\lambda}(x) = (< \rho_i^{\lambda}, \rho_i^{!}, \rho_i^{\dagger} >, e) \\ \rho_i^{\lambda}, \rho_i^{!}, \rho_i^{\dagger} \ \vdash \ e \ \Longrightarrow \ v\end{array}}{\rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ x \ \Longrightarrow \ v}$$

Since $\mathcal{L}$ is a non-strict language, the binding values of identifiers are always closures. Closures are dereferenced only when the value of an identifier is required.

(Record Introduction)

$$\rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ [\ x = e\ ] \Longrightarrow \ \rho_{\perp}[x \mapsto \ (< \rho^{\lambda}, \rho^{!}, \rho^{\dagger} >, e)]$$

Records themselves are defined in terms of a binding environment in which its constituent expressions are closed within the current evaluation environment.

(Abstraction)

$$\rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ (\lambda \ (x) \ e) \implies (< \rho^{\lambda}, \rho^{!}, \rho^{\dagger} >, (\lambda \ (x) \ e))$$

(Application)

$$\frac{\begin{array}{c} \rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ e_1 \implies (< \rho^{\lambda}_a, \rho^{!}_a, \rho^{\dagger}_a >, (\lambda \ (x) \ e)) \\ \rho = \rho_{\perp}[x \mapsto \ (< \rho^{\lambda}, \rho^{!}, \rho^{\dagger} >, e_2)] \\ \rho^{\lambda}_a[\rho], \rho^{!}_a[\rho], \rho^{\dagger}_a \vdash \ e \implies v \end{array}}{\rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ (e_1 \ e_2) \implies v}$$

$$\frac{\begin{array}{c} \rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ e_1 \implies (< \rho^{\lambda}_a, \rho^{!}_a, \rho^{\dagger}_a >, (\lambda \ (x^{\dagger}) \ e)) \\ \rho = \rho_{\perp}[x \mapsto \ (< \rho^{\lambda}, \rho^{!}, \rho^{\dagger} >, e_2)] \\ \rho^{\lambda}_a[\rho], \rho^{!}_a[\rho], \rho^{\dagger}_a[\rho] \vdash \ e \implies v \end{array}}{\rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ (e_1 \ e_2) \implies v}$$

Abstractions evaluate to closures in which the binding environment component is the current evaluation environment and the expression component is the λ phrase. Application of an abstraction involves constructing a closure for the argument (thereby delaying its evaluation), and evaluating the body of the abstraction in an environment in which the λ-bound variable of the abstraction is bound to the argument closure.

(Prompt)

$$\frac{\rho^{\lambda}, \rho_{\perp}, \rho_{\perp} \vdash \ e \implies v}{\rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ !e \implies v}$$

A prompt expression, $!\,e$, causes e to evaluate relative to fresh prompt and public binding environments. Thus, bindings found "outside" the prompt are invisible to any `reflect` and `reify` sub-expressions of e.

(Reflection)

$$\frac{\begin{array}{c} x_1, x_2, \ldots, x_n \ \text{ potentially free in } e_2 \\ \rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ e_1 \implies \rho \\ \rho_r(x) = \begin{cases} \rho(x) & \text{if } x = x_1, x_2, \ldots, x_n \\ \mathit{undef} & \text{otherwise} \end{cases} \\ \rho^{\lambda}[\rho_r], \rho^{!}[\rho_r], \rho^{\dagger} \vdash \ e_2 \implies v \end{array}}{\rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ (\texttt{reflect } e_1 \texttt{ in } e_2) \implies v}$$

(Reification – Simple)

$$\rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ (\texttt{reify}) \implies \rho^{\dagger}$$

(Reification – General)

$$\frac{\begin{array}{c} \rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ e \implies (< \rho^{\lambda}_f, \rho^{!}_f, \rho^{\dagger}_f >, (\lambda \ (x) \ e_b)) \\ x_1, x_2, \ldots, x_n \ \text{ potentially free in } (\lambda \ (x) \ e_b) \\ \rho_r(x) = \begin{cases} \rho^{\dagger}_f(x) & \text{if } x = x_1, x_2, \ldots, x_n \\ \mathit{undef} & \text{otherwise} \end{cases} \end{array}}{\rho^{\lambda}, \rho^{!}, \rho^{\dagger} \ \vdash \ (\texttt{reify } e) \implies \rho_r}$$

5 Applications

It is widely agreed that central themes in object-based programming are issues of code reusability and name overloading. Class-based inheritance[12, 13, 21], delegation[22, 30, 31], and subtyping[8] are among competing proposals that address these issues.

Code reusability and name overloading techniques are expressible given a mechanism that permits the free capture and projection of environments. An object O that is to reuse names or methods defined by O' needs to access the environment within which O' evaluates. Reification implements this capture operation; reflection provides the capability to use captured environments to evaluate other expressions.

5.1 Building Prototype Objects

Object Generators. As a first example, we discuss the utility of environment-based reflection in building modified versions of existing generators or modules. In the context of a reflective programming model, records are properly viewed as objects, and record-yielding abstractions are regarded as object generators.

Consider the simple problem of implementing a stack object; we might choose to structure such an implementation thus:

```
StackGen = let make_rep = (lambda (args) return a representation),
               overflow = (lambda (rep elt) handle stack overflow)
           in  (λ (size)
                   let Rep = (make_rep args),
                       top = top of stack counter
                   in  [ Push   = (λ (elt) ...),
                         Pop    = (λ () ...) ]
```

`StackGen` is an object generator; when applied, it returns a new stack instance:

```
MyStack = (StackGen MySize)
```

A stack instance is a record containing operations allowable on stacks; these operations are closed over the `overflow` exception handler and `make_rep`. By making various `let`-bound identifiers public, `Stack` can be transformed into a prototype object generator:

```
let make_rep† = (lambda (args) return a representation),
    overflow† = (lambda (rep) handle stack overflow)
in  (λ (size)
          ⋮)
```

To define a new kind of stack that uses the same representation as `StackGen` objects but which implements different `Push` and `Pop` operations, we write:

```
let NewStack = (λ (ProtoTypeStack)
                   (reflect (reify ProtoTypeStack) in
                      (λ (size)
                           let my_rep = (make_rep args),
                               top = top of stack counter
```

```
                         in [ Push = (λ () ...),
                              Pop  = (λ () ...) ]))
in (NewStack StackGen)
```

Free references to make_rep and overflow that occur within the body of the reflect expression refer to their binding values in StackGen's closure; if NewStack is not closed over a particular binding which happens to be referenced within the body, the binding value is determined from the current evaluation environment.

Thus, suppose the Push operation defined by NewStack refers to overflow. The value of overflow is the closure defined in StackGen. In other words, instances of StackGen and NewStack share the same stack overflow handler; overflow is a default handler for any stack. To define a specialized handler local to NewStack, the generator is restructured:

```
let NewStack = (λ (ProtoTypeStack)
                  (reflect (reify ProtoTypeStack) in
                    (λ (size)
                         let my_rep = (make_rep args),
                             top = top of stack,
                             overflow† = (λ (rep) new stack overflow handler)
                          in [ Push = (λ () ...),
                               Pop  = (λ () ...) ]))
in (NewStack StackGen)
```

The specialized version of overflow defined by NewStack may refer to the prototype definition. Moreover, if it is declared as public, and occurs free in the definition of Push, it becomes available to any object which is instantiated using the environment captured by Push's closure. For example, if the abstraction returned by the above expression is called NewStackGen, evaluating:

```
let MyNewStack = (NewStackGen size)
in  (reify MyNewStack.Push)
```

returns a record containing the bindings of all public variables that occur free in Push's closure.

Reification permits the expression of *dynamic inheritance*[9]. A system that supports dynamic inheritance allows new methods to be incorporated into the object hierarchy dynamically. If the definition of the overflow exception handler defined in StackGen was changed[7], modified stack implementations that use this procedure as the default exception handler would see the change.

One limitation in this formulation is its inability to permit general delegation of operations to different objects. Modifications of a stack generator have access only to the public free variables referenced by the generator. Thus, if StackGen defined a Print operation, in addition to Push and Pop, NewStackGen would have access to Print's free variables, but not to the procedure itself. In order for a print operation to work over instances of NewStackGen, it must be defined explicitly within the record returned. We consider the issue of delegation of operations across objects (rather than object generators) in the following section.

---

[7] We haven't provided mutation operators in the kernel language, but a semantics that supports references and stores is straightforward to incorporate[17].

Critics might argue that building modified versions of object generators is possible even in the absence of reification and environment-based reflection. Consider two possible alternatives. First, 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.,* `StackGen` and `NewStack`) reside in the same lexical context. This obviates the need to explicitly package and unpackage bindings via `reflect` and `reify`. The approach has the significant limitation, however, of requiring the original prototype environment be altered whenever a new modification needs to be reevaluated. Modularity is significantly reduced as a result.

Another non-reflective solution is possible in a language with record objects and a Pascal-style "`with`" operation. All public bindings found in a generator's closure are packaged as part of the record object returned. The `reflect` operation is subsumed by a "`with`" expression. This formulation also has some significant drawbacks however. If there exist several prototype objects defined in the same lexical context that share common operations, they must all duplicate these operations as part of their record representation. For example, a stack and a tree object defined in the same context may both share common print and exception handlers. To ensure that modified versions of these objects have access to these operations, instances of stacks and trees must both explicitly define bindings for these operations. If new exception routines are added to `StackGen`, for example, all object generators derived from `StackGen` must in turn be altered to reflect the availability of this new procedure.

Delegating Operations to Objects. In the previous section, we defined modified versions of object generators. This was possible because closure-based reification permits abstractions to share free variable bindings even if they do not exist in the same scope. The natural extension of this approach would be to permit an instance of one object to delegate operations to instances of other objects. Given this functionality, instances of an object may have different behaviors based on how they choose to delegate operations.

To illustrate, consider the familiar problem of specifying geometric objects. A box object consists of `x` and `y` coordinates, a `length`, a `width`, and `depth`, a procedure to move boxes from one coordinate to another, and a print routine to print box objects on a grid of some dimension. The outline of a simple box generator is as follows:

```
letrec MakeBox = (λ (x† y† length† width† depth†)
                       [ move =  (λ (dx dy)
                                      (MakeBox (+ x dx) (+ y dy)
                                               length width depth)
                         print = (λ () print box using grid coordinates) ]
       grid† = shape of grid on which boxes are printed
in MakeBox
```

By making the initial shape of 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: `MyBox = (MakeBox` *arguments*`)`.

Suppose we also 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` method that maps 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 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:

```
letrec MakeColorBox =
   (λ (box color†)
        (• box
            [ print = (reflect (reify box.print)
                          in (λ () new print routine that references grid))])
 in MakeColorBox
```

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

```
MyColorBox = (MakeColorBox MyBox initial_color)
```

Free references to shape information (*e.g.,* `x`-`y` coordinates, `length`, etc.) in the `print` method found in `MyColorBox` are resolved relative to their definition in `MyBox`. The new `print` method also has access to the public variables (*e.g.,* `grid`) that occur free in `MyBox`'s print method. Thus, the color `print` method is a modified implementation of the `print` method defined for a simple box. The record returned by `MakeColorBox` also contains a binding for `move`; both boxes and color boxes share this method. `MyBox`'s `print` method is shadowed by `MakeColorBox`'s definition.

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.

Unlike the implementation of stacks in which all modified implementations define their own methods, boxes and color boxes share individual methods. `MakeBox` could also be treated as an object generator:

```
(reflect (reify MakeBox) in
   letrec NewBoxGen = (λ (x† y† length† width† depth†)
                          [ move = (λ (new_x new_y) ...),
                            print = (λ () ...) ])
    in NewBoxGen)
```

`NewBoxGen` is bound to an abstraction that acts as another box generator. The record returned by applying `NewBoxGen` contains procedures that are closed over the public free variables defined by `MakeBox`. Free references to `grid` that occur in `print`, for example, get resolved based on `grid`'s definition in `MakeBox`.

The advantages of using reification and reflection in this example are similar to those in our earlier definition of a stack. We could have chosen to avoid reifying over closures by explicitly packaging all shape information into the object returned by applying `MakeBox`. If the methods defined by `MakeBox` refer to other public free variables in their body (*e.g.,* coordinate boundaries, exception handlers, transcendental operators, etc.), however, their bindings would have to be exported as well. Such a solution comprises modularity since it forces an object relevant only to the implementation of boxes to become visible in the interface specification of the abstraction. Given that a box can be manipulated only via the `print` and `move` methods it provides, packaging coordinate bindings, grid tables, exception routines, etc. along with the other bindings needed to build specialized versions of a `box` into its record representation would be an unfaithful characterization of its specification. Using reification to examine internal representations does not require modification of this specification. Only objects that view box as a prototype need access to operations associated with its implementation.

5.2 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[11], Reddy[26], and Kamin[19] 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 non-operational definition of the "self" pseudo-variable found in Smalltalk-style languages.

Building modified versions of recursive structures is also possible using environment-based reflection. 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. $\mathcal{L}$ is distinguished from these other proposals insofar as it provides a concrete self-contained *linguistic framework* within which class-based inheritance strategies implemented in terms of recursive record structures and late-binding can be expressed.

The notion of "*self*" is implemented using record composition, reflection and reification. Classes are record generators and a class hierarchy is built by composing new instances of records generated from a set of super-classes; these records are composed with the bindings found in the current evaluation environment. 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 [11, 19]. A `circle` is a sub-class 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. The code for `points` is given in figure 5.

The reflected image of the `self` record argument is used to define the evaluation environment of `point`'s methods; it is the definition of *self* that gives the late-binding semantics

```
point = !(λ (obj a† b†)
           letrec self = (• (• methods (reify)) obj)
                  methods = (reflect self in
                               [ DistfromOrig = (sqrt (+ (sqr a) (sqr b)))
                                 ClosertoOrig =
                                   (λ (p)
                                       (< DistfromOrig
                                          (reflect p in DistfromOrig)))])
          in  (• methods (reify)))
```

Fig. 5. A `point` generator.

of object-based languages such as Smalltalk[13]. `Self` is defined to be a record containing the bindings found in the caller (`obj`), `point` 's methods, and bindings for `a` and `b` . The mutual recursion that exists between `self` and `methods` is handled naturally by the non-strict semantics of the language; resolving the recursion is tantamount to finding a fixpoint for these two definitions. Late-binding (*i.e.,* the definition of `self`), and method sharing are realized by closure-based reification. For example, the first reference to `DistfromOrig` in `ClosertoOrig` refers to `DistfromOrig` 's binding-value in `self` ; if `obj` does not define such a binding, the value of `DistfromOrig` in `methods` is used instead. The object returned by the point generator contains bindings for `a` and `b` as well as the methods defined by `methods` .

We point out that lazy evaluation of records fields is not fundamental to the correctness of this solution. `self` and `methods` could be introduced as abstractions closed over a proper recursive environment or alternatively, the semantics of letrec could be based on a "lenient" evaluation strategy that would cause the concurrent evaluation of all its bindings; dataflow synchronization ensures that expressions have access to proper binding values. The implications of this approach in the context of reflection is given in [18].

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.

The object yielded by creating a new circle is a record containing the method definitions and variables of both points and circles. The meaning of `DistfromOrig` in a `circle` instance should refer to its meaning as specified by the `circle` (*not* the `point`) generator. This means that in resolving the binding value of `DistfromOrig`, the `ClosertoOrig` method found in the point generator defined when a new circle is created should use the definition of `DistfromOrig` relevant to circles. The code for the `circle` generator is given in figure 6.

Circles inherit properties of points. To create a circle, we first define a new point instance. 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` .

```
circle = !(λ (obj a† b† r†)
             letrec self = (• (• methods (reify)) obj)
                    super = (point self a b)
                    methods = (reflect self in
                                   [Distfromorig =
                                      (max (- (reflect super in
                                                 Distfromorig) r) 0))]
             in  (• super (• methods (reify))))
```

Fig. 6. A `circle` generator.

The object returned by this definition contains the binding for `DistfromOrig`, the instance variables `a`, `b` and `r` and the binding for `ClosertoOrig` found in the point instance associated with this new circle.

If `circle` has no sub-classes, we can create a new instance by evaluating:

`(circle [  ]` *value-of-a value-of-b value-of-r*`)`

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 record representation of C_1 and C_2; this image would contain the method definitions for objects instantiated from C_1 and C_2.

Lexically-scoped languages that do not support reflective environments would be hard-pressed to support this functionality given the clumsiness of achieving late-binding using lexical scoping. Of course, it is possible to express object-based programming in languages like T[2] or Common Lisp[5, 29] that are statically scoped. Support for objects in these systems however often involves extensions to the language kernel (*e.g.,* dynamically-scoped instance variables in Common Lisp) or significant alterations to the language kernel (*e.g.,* as in T). 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 language kernel. Reflection permits distinct binding disciplines to be supported within a unified framework; thus, late-binding protocols essential in building object-based systems can be expressed in a lexical binding kernel without the need to define new binding primitives or alter existing ones.

6 Inheritance as Syntax

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 record composition and reification thus:

```
self = (• (• methods (reify)) obj)
```

where (reify) captures the local environnment (*e.g.,* instance variables) of the object being defined, methods is a record containing the method definitions of this object, 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:

super = (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; $\mathcal{L}$ programmers need only have the 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.

To specify an object generator that defines public bindings, we use a make_object macro that takes the form:

```
(make_object O
             (arguments a1 a2 ... aj)
             (methods m1 m2 ... mk)
             (local_definitions d1 d2 ... dl))
```

Each of the arguments is a variable name that may be suffixed with a "† to indicate that it is a public variable; each of the local definitions and methods are pairs of the form, (*name* = *expression*). This macro expands to the following $\mathcal{L}$ expression:

```
letrec O = (λ (a1 a2 ... aj)
               let d1, d2, ..., dl
               in  [ m1, m2, ..., mk])
in O
```

We can also define a "delegate" macro such that:

(delegate *prototype modification*) $\equiv$ (reflect (reify *prototype*) in *modification*)

Given a prototype operation P, a modification M can be specified by evaluating (delegate P M) ; M's definition is based on the free public bindings used to define P. The expression:

```
(delegate MakeBox
          (make_object NewBoxGen
                       (arguments x† y†)
                       (methods (move = (λ (new_x new_y) ...)),
                                (print = (λ () ...)))))
```

expands to the modified NewPointGen object generator shown earlier.

Class based inheritance strategies expressed using reflection operations are also easily transformed into a more abstract, succinct form. For example, a make_class macro is applied thus:

```
(make_class
      class_name
      (instance_vars i1 i2 ... im)
      (super_class super_class_name)
      (super_class_args super_class_arguments)
      (local_definitions  d1 d2 ... dn)
      (methods m1 m2 ... mo)
```

The terms following `instance_vars` range over variables, whereas the terms following `local_definitions` and `methods` are pairs of the form (*name* = *expression*). We assume a sufficiently expressive macro language that would permit us to avoid specifying fields if they are irrelevant to the specification of the object. [20] describes one such system.

The `make_class` macro expands to:

```
class_name = !(λ (obj i1† i2† ... im†)
                          let d1,
                              d2,
                              ⋮
                              dn
                          in letrec self = (• (• methods (reify)) obj)
                                    super = (super_class_name self
                                                              super_class_arguments)
                                    methods = (reflect self in
                                                  [ m1, m2, ..., mo])
                              in (• super (• methods (reify))))
```

Message sending is simply expression evaluation relative to a specified record; thus:

(send *object message*) $\equiv$ (reflect *object* in *message*)

Figure 7 gives a definition of a circle and point class using these syntactic abbreviations.

```
(make-class point
     (instance_vars a b)
     (methods ((DistfromOrig = (sqrt (+ (sqr a) (sqr b))))
               (ClosertoOrig = (lambda (p) (< DistfromOrig
                                              (send p DistfromOrig))))))


(make-class circle
    (instance_vars a b r)
    (super_class point)
    (super_class_args a b)
    (methods (DistfromOrig = (max (- (send super DistfromOrig) r) 0))))
```

Fig. 7. Specification of a circle and point class using macros. The "`self`" pseudo-variable is implicitly subsumed in the definitions. All free names are resolved relative to the environment in effect at the time a class instance is created, not the environment extant at the time the class definition is evaluated.

7 Conclusions and Comparison to Related Work

The operational view of inheritance in Simula[12] and Smalltalk [13, 16] led to its emphasis as a programming method to support sharing and reusability of code and data. In Smalltalk, for example, the concept is manifested in the name-lookup rule — the meaning of free names occurring in a message is determined by the object (and all its superclasses) to which the message is sent. In class-based languages, name overloading becomes the key operational feature of inheritance and explicit linguistic mechanisms are provided to build class hierarchies. Providing operations to reflect and reify over environment structures obviates the need for linguistic mechanisms (such as class or method definitions) tailored explicitly for inheritance.

Lieberman [22], Ungar [31, 9], and Stein [30] have advocated a variant of class-based inheritance in which objects subsume the functionality of classes. Objects receive messages which can be forwarded at their discretion to other objects. An object is a *prototype* for a class and *delegation* replaces message-passing as the main protocol for realizing inheritance. Reifying over closures closely captures the behavior of delegation since a closure defines a local namespace which, in the presence of reflection, constitutes a prototype object.

There has been much interest in a type-theoretic description of inheritance[15, 24, 32]. Under this view, inheritance can be implemented given a suitable subtype relation (or similar constraint system) over objects or records. To paraphrase [7], a record type τ is considered to be a subtype of τ' if it has at least all the fields of τ such that the common fields of τ and τ' are related under the subtype rule. A suitably constructed subtype relation permits strongly-typed, statically-scoped languages to support name-overloading albeit in a manner quite different from its manifestation in dynamically-scoped, weakly-typed languages like Smalltalk. Inheritance is realized by constructing a type system that supports inclusion polymorphism on records. The pragmatic utility of such an approach stills appears to be an issue of debate given the subtlety and complexity of the type rules [10].

Our approach is orthogonal to these efforts; we don't rely on a strong type-system to build inheritance systems, although a type system for a reflective language similar to $\mathcal{L}$ does exist[17]. In our context, type information is used to determine the presence (or absence) of bindings in environment-yielding (reflective) operations. For example, in the code fragment:

```
(λ (x) (reflect x in y))
```

the binding-value of `y` is predicated on the meaning of `x`. $\mathcal{L}$'s type system, in effect, performs static analysis of the call points to this function and makes inferences of the form, "All instances of `x` define a binding for `y`", or "No instance of `x` defines a binding for `y`", or "Only certain instances of `x` define a binding for `y`." Based on the information gleaned from such an analysis, we expect to transform expressions using late-binding operations into equivalent lexical-binding ones that can be efficiently compiled.

Rather than using type inference and subtyping, America[3] and Snyder[28] have argued that subtyping should be separated from inheritance which, in their view, serves primarily as a vehicle for code-sharing. For example, in implementing a stack, we may wish to inherit code from an array but we would not want to consider a stack to be a subtype of an array [3]. Reflection permits us to support this kind of formulation as described earlier.

There has also been much work in generalizing reflection to work in the presence of fine-grained concurrency[23, 34]; the use of reflection in this context permits program control over implementation concerns such as monitoring, scheduling, migration, etc. Insofar as one goal of these projects is to permit flexible high-level mechanisms for managing semantic objects (such as environments), it shares much in common with the stated aims of this work. The introduction of concurrency, and the desire to support programming environment functionality within the base language, however, distinguishes the presentation and focus of these efforts from ours in obvious and important ways.

The reflective systems of Brown[33] and 3-Lisp[27] differ from our work in two significant respects. First, our reflective model does not consider reification of expressions or continuations. Reification over expressions requires, in effect, keeping a compiler resident at runtime. While useful for extending the functionality of interpreters and for providing an expressive platform for manipulating macros, expression-based reification is orthogonal to more general modularity concerns. Secondly, our treatment of environments in this paper often takes advantage of the non-strict semantics we give them. The process of constructing an environment is decomposed from its reification. This separation of concerns leads to expressivity gains not addressed in these other reflective models.

In summary, our contribution is best regarded as a unification of different kinds of inheritance and delegation schemes within a simple linguistic framework. It defines an operational characterization of object-based systems in which objects are viewed *not* as fundamental elements in the language's semantics, but as composite structures built from more primitive reflective operations. As a result, we argue that this approach offers the promise to be a practical and simple vehicle in which complex object-based systems can be constructed. Moreover, it is the first attempt to our knowledge that uses reflection as a language basis for modeling inheritance; we feel the implications of such a unification deserves further investigation.

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 1988 Conference on Lisp and Functional Programming*, pages 277–288, 1988.
3. Pierre America. Issues in the design of a parallel object-oriented language. In Pierre America and Jan Rutten, editors, *A Parallel Object-Oriented Language: Design and Semantic Foundations*, chapter 2. Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands, 1989. in PhD thesis.
4. H. Barendregt. *The Lambda Calculus.* North-Holland, 1981.
5. 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.
6. 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.

7. Luca Cardelli. A Semantics of Multiple Inheritance. In *International Symposium on Semantics of Data Types*. Springer-Verlag, 1984. Lecture Notes in Computer Science, Number 173.
8. Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
9. Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.
10. William Cook, Walter Hill, and Peter Canning. Inheritance is Not Subtyping. In *17^{th} ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.
11. William Cook and Jens Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *OOPSLA'89 Conference Proceedings*, pages 433–444, 1989. Published as SIGPLAN Notices 24(10), October, 1989.
12. O.J. Dahl, B. Myhruhaug, and K. Nygaard. The Simula67 Base Common Base Language. Technical report, Norwegien Computing Center, 1970.
13. Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Press, 1983.
14. Michael Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.
15. Justin Graver and Ralph Johnson. A Type System for Smalltalk. In *17^{th} ACM Symposium on Principles of Programming Languages*, pages 136–150, 1990.
16. 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.
17. Suresh Jagannathan. A Programming Language Supporting First-Class, Parallel Environments. Technical Report LCS-TR 434, Massachusetts Institute of Technology, December 1988.
18. Suresh Jagannathan. Environment-based reflection. Technical Report 91-001-3-0050-1, NEC Research Institute, January 1991.
19. Samuel Kamin. Inheritance in Smalltalk-80: A Denotational Definition. In 15^{th} *ACM Symposium on Principles of Programming Languages*, pages 80–87, 1988.
20. Eugene Kohlbecker and Mitch Wand. Macro-by-Example: Deriving Syntactic Transformations from their Specifications. In *14^{th} ACM Symposium on Principles of Programming Languages*, pages 77–85, 1987.
21. B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. The BEA Programming Language. In Bruce Shiver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.
22. 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.
23. Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In *Proceedings of European Conference on Object-Oriented Programming*, pages 231–250, 1991. Published as Springer-Verlag LNCS 512.
24. Jens Palsberg and Michal Schwartzbach. Object-Oriented Type Inference. In *OOPSLA'91 Conference Proceedings*, pages 146–161, 1991.
25. Gordon Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.

26. Uday Reddy. Objects as Closures: Abstract Semantics of Object-Oriented Languages. In *Proceedings of the Conference 1988 on Lisp and Functional Programming*, pages 289–297, 1988.
27. Brian Smith. *Reflection and Semantics in a Procedural Language.* PhD thesis, Massachusetts Institute of Technology, 1982.
28. Alan Snyder. Encapsulation and inheritance in object-oriented languages. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 38–45. ACM Press, 1986.
29. Guy Steele Jr. *Common Lisp: The Language, Second Edition.* Digital Press, 1990.
30. Lynn Stein. Delegation is Inheritance. In *OOPSLA'87 Conference Proceedings*, pages 138–146, 1987. Published as SIGPLAN Notices 22(12), December, 1987.
31. 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.
32. Mitchell Wand. Complete Type Inference for Simple Objects. In *Second IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987.
33. Mitchell Wand and Daniel Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In *Proceedings of the 1986 Conference on Lisp and Functional Programming*, pages 298–307, 1986.
34. Akinori Yonezawa and Takuo Watanabe. An Introduction to Object-Based Reflective Concurrent Computations. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 50–54, 1989.