Coercion as a Metaphor for Computation

Suresh Jagannathan*
Department of Computer Science
Yale University

Abstract

The idea of *coercion* – taking objects of one type and transforming them into objects of another – is not a new one, and has been an important feature of language design since the advent of Fortran. This paper considers a generalization of coercion that permits structured transformations between *program* and *data structures*. The nature of these coercions goes significantly beyond what is found in most modern programming languages.

Our intention is to develop a programming model that permits the expression of a wide-range of superficially-diverse modularity constructs within a simple and unified framework. We base the design of this model on the observation that a variety of program structures found in modern programming languages are represented fundamentally in terms of an environment. Given suitable transformations that map the environment representation of a program structure into a data object, we can enable the programmer to gain explicit control over his naming environment.

We investigate the semantics of program/data coercion in the presence of a non-strict parallel evaluation semantics for environments. Parallelism and program/data coercion form an interesting symbiosis and it is the investigation of their interaction that forms the primary focus of this paper.

Key Words and Phrases: Namespace management, modularity, first-class environments, reflection, object-oriented programming, actors, non-strictness, interpreters.

1 Introduction

The idea of *coercion* – taking objects of one type and transforming them into objects of another – is not a new one, and has been an important feature of language design since the advent of Fortran. Most languages provide a limited form of coercion, *e.g.,* integers are usually allowed to be coerced into reals (and *vise versa*); some languages come equipped with more wide-ranging coercion mechanisms: APL and Algol 68 allow coercion between array and scalar types; PL/1 permits coercion between numeric and string types. This paper considers a generalization of coercion that permits structured transformations between *program* and *data structures.* The nature of these coercions goes significantly beyond what is found in most modern programming languages.

Our intention is to develop a programming model that permits the expression of a wide-range of superficially-diverse modularity constructs within a simple and unified framework. We base the design of this model on the following observation: a variety of program structures (*e.g.,* closures, packages [8], classes [12, 17], etc.) are represented fundamentally in terms of an environment structure[2]. Given suitable transformations that map the environment representation of a program structure into a data object, we can enable the programmer to gain explicit control over his naming environment. Inverse transformations that permit a data object to be transformed or "lifted" into an environment image allow the programmer to build his own customized environments. A data object so-transformed can be used in any context where an environment is expected; similarly, an environment so-transformed can be used wherever a data value can.

Making the representation of environments explicit within the language leads to a number of important expressivity gains. We describe some of these briefly below and expand on these points in the remainder of the paper.

1. We can customize the naming environment of any expression by evaluating the expression within the context of a coercion operation that transforms a user-specified data structure into an environment. New binding protocols can be introduced dynamically and the binding-environment of any expression can be customized appropriately.

2. Given the ability to compose data structures together, we can now compose environments in arbitrary ways. Data structure/environment composition allows us to capture the essence of object-oriented programming

*Funding for this work provided in part by NSF grants CCR-8601920, CCR-8657615 and ONR N00014-86-K-0310.

*Appeared in the Proceedings of the the 1990 IEEE International Conference Computer Languages.

[2]Two examples: (a) A closure is a binding-environment coupled with an expression that is constrained to evaluate within that environment. (b) The definitions found along a given chain in a class-based inheritance hierarchy may be used to implicitly affect the evaluation of expressions that refer to a class instance found at the lowest level of the hierarchy.

models that support inheritance. With an appropriate parallel evaluation semantics, transformations between program and data structure can be used to realize an actor-based programming model.

3. An expression can examine the environment within which it is evaluated. The ability of an expression to examine its own evaluation environment forms the basis of reflective programming models. A reflective procedure applied in the context of a parallel programming model can be used to implement process daemons, processes that can examine an evaluation environment in parallel with other expressions that access and manipulate this environment.

4. We can characterize program structures in terms of the coercion operators provided by the model. We can specify an interpreter, for example, without having to implement data structures and operations that explicitly manipulate an environment representation. A coercion-based interpreter of the sort discussed below is *self-describing* in the sense that the structures it manipulates can be given a well-defined representation within the language itself.

The paper is organized as follows: in the next section, we give a brief description of the programming model. The essence of the model is captured by two coercion operators that effect transformations between environment structures and data structures. The model also defines a specific data representation for environments; this representation takes the form of a non-strict record object. The presence of non-strictness makes parallelism an integral part of the model. Section 2.3 provides a formal semantics of a kernel language containing these operators. Section 4 examines a set of paradigms and applications that result from the interaction of non-strict data structures with program/data coercion. We begin by arguing that binding protocols typically "hard-wired" as part of the semantics of a language can in fact be customized as suits the convenience of the programmer. We then go on to consider the relationship between the environment structure produced by a meta-circular evaluator and the structure manipulated by an object-based program. Reflection is considered to be a consequence of first-class environments and non-strict data structures. Actor systems are examined in the context of a reflective programming environment; we show how the actor model can be simplified without loss of expressive power through the use of the coercion mechanisms developed here. Section 5 presents conclusions.

2 The Nature of the Model

We present a model that permits structured transformations between data objects and environments. An environment is a collection of bindings of names to values that is maintained by a programming language's interpreter. The names defined by a program structure are represented as elements of some environment. Environments can be coerced into data objects that broadly resemble conventional record structures. Users can manipulate such a record using the operations allowed upon it. In addition, a record object may in turn be coerced into an environment which can then be used (implicitly) to affect the evaluation of other expressions.

The semantics of the model is first given in terms of the simply typed λ-calculus augmented with record types and the transformation operations. We then examine the implication of injecting these operators into Scheme, a richer language that supports assignment.

2.1 Records

Records define a collection of mutually-recursive bindings of possibly heterogeneous type. An expression of the form

$$[\ \mathtt{id}_1 = \mathtt{e}_1,\ \mathtt{id}_2 = \mathtt{e}_2,\ \ldots,\ \mathtt{id}_n = \mathtt{e}_n\ ]$$

denotes an n-field record with field names $\mathtt{id}_1$, $\mathtt{id}_2$,..., $\mathtt{id}_n$; each of the e_i evaluate in an environment containing bindings for the id_i. Since our intention is to understand how to apply this model in a variety of different contexts, including some involving the use of parallelism, we ascribe a parallel-evaluation semantics to our record objects: each of the $\mathtt{e}_i$ evaluate in parallel subject only to the standard data flow dependency constraints. Record objects are also non-strict. Thus, a record object can be made available for inspection even if some of its elements are still under evaluation *provided* that the names it defines have been noted. In particular, a record has a well-defined meaning even if its component elements diverge. Non-strictness of this kind also implies parallelism since expressions can access a record structure even as other expressions proceed to compute the value of the record's fields.

The value of a record field can be retrieved using the "." operator: if r is a record, then evaluating $r.id$ returns the binding-value of id as defined in r; if r does not contain a field named id, an error results. The non-strict semantics of records means that we can evaluate a "." expression even if not all fields in its record argument yield values; thus,

$$[\ \mathtt{m} = \perp,\ \mathtt{n} = 1\ ]\ .\ \mathtt{n}$$

yields `1` (where $\perp$ denotes a diverging computation). The expression `M.x` is *blocked* until a value has been computed for the field named `x` in record `M`.

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. Then the "join" or composition of r_1 and r_2 (written ($r_1 \bullet r_2$) can be expressed as follows:

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

Like ".", $\bullet$ is also non-strict: it does not require its arguments to be values before it returns its result. Thus, the join of r_1 and r_2 can be computed even if both record expressions are still under evaluation.

2.2 The Coercion Operators: $\uparrow$ and $\downarrow$

In addition to providing record types, the model supports two operations (denoted $\uparrow$ and $\downarrow$) that implement our coercion semantics. The $\downarrow$ operator is used to coerce the environment representation of an expression into a record. If e_1 is an expression, then $(\downarrow\ e_1)$ yields a record with two fields, `val` and `env` bound to the value denoted by e_1 and to the record-image of e_1's environment representation, resp. (Informally, every expression has a representation in terms of an environment; the environment structure associated with an expression, in general, contains the set of bindings accessible to and introduced by the expression during the course of its evaluation; we give a more precise definition below.) Suppose that e_1's environment image contains a binding for x to v. The `env` field in the record-object yielded by evaluating $(\downarrow\ e_1)$ will contain a field named x bound to v.

The inverse of the $\downarrow$ operator is $\uparrow$. Given an expression e_1 that yields a record, we can evaluate an expression e_2 in the context of e_1's environment image by evaluating $(e_1 \uparrow e_2)$. The (non-strict) record-object yielded by evaluation of e_1 is coerced into an environment that contains a binding for each field-name found in the record. The binding-value of a free identifier found in the body of e_2 *not* defined within the environment image of e_1's record-object is resolved within e_2's lexical environment.

2.3 Formal Semantics

We give a formal semantics to a functional base language that supports the operators defined in the previous section. The term set Λ of our base language is defined inductively over a set of constants and variables:

$$
\begin{aligned}
\mathrm{E} ::= {} & 0 \mid 1 \mid 2 \ldots \\
& \mathrm{true} \mid \mathrm{false} \mid \\
& [\mathrm{id}_1 = \mathrm{E},\ \mathrm{id}_2 = \mathrm{E},\ \ldots,\ \mathrm{id}_n = \mathrm{E}] \mid \\
& \lambda \mathrm{id.E} \mid (\mathrm{E\ E}) \mid \mathrm{E} \rightarrow \mathrm{E;E} \mid \\
& \mathrm{E.id} \mid (\mathrm{E} \bullet \mathrm{E}) \mid (\mathrm{E} \uparrow \mathrm{E})\ (\downarrow \mathrm{E})
\end{aligned}
$$

where `id` ranges over variables. λ abstractions stand for call-by-name (non-strict) procedures.

We give meaning to expressions in our base language by associating each expression with an element of a sum domain $\mathbf{V}$ that satisfies the following isomorphism:

$$
\begin{aligned}
&\mathbf{V} \cong \mathbf{N} + \mathbf{B} + \mathbf{R} + (\mathbf{V} \rightarrow \mathbf{V}) \\
&\mathbf{R} = \mathbf{Env} \cong \mathbf{Id} \rightarrow \mathbf{V}
\end{aligned}
$$

where $\mathbf{N}$ is the flat domain of natural numbers and $\mathbf{B}$ is the flat domain of Booleans. The continuous function space operator is denoted by $\rightarrow$. $\mathbf{R}$ and $\mathbf{Env}$ denote the domain of functions that map identifiers to values. Details regarding the construction of $\mathbf{V}$ can be found in [7].

We use the following conventions throughout:

- d in $\mathbf{V}$, where d belongs to a summand $\mathbf{S}$ of $\mathbf{V}$ denotes the injection of d into $\mathbf{V}$.
- If $v = (d \text{ in } \mathbf{V})$ for some $d \in \mathbf{S}$, then $v\mid_S = d$. Otherwise $v\mid_S = \perp$.
- $=$ yields $\perp$ whenever either of argument does.

2.3.1 Operations over Environments

If A is a domain, then $\tilde{A}$, the domain extension of A, is defined by:

$$\tilde{A} = A + \{none\} + \{formal\}$$

The special values *none* and *formal* are used to distinguish unbound identifiers and the formals of an abstraction from ordinary binding-values: if x is mapped to *none* in some environment $\tilde{A}$, it implies that x is unbound in $\tilde{A}$; if x is mapped to *formal* by $\tilde{A}$, it indicates x is defined as a formal parameter within $\tilde{A}$.

We define a projection operation that maps elements from $\tilde{A}$ back to A with the function *proper*:

$$
\begin{array}{lll}
proper: & \tilde{A} & \rightarrow A \\
proper & \tilde{a} & = a\mid_A \\
proper & none & = \perp \\
proper & formal & = \perp
\end{array}
$$

If A is a set of identifiers and B is a domain of values, then the domain of environments binding elements of A into B is given by $A \rightarrow \tilde{B}$. We define singleton environments using the $\mapsto$ constructor:

$$[i \mapsto v]\rho = \lambda x. x = i \rightarrow v; \rho(x)$$

Let ρ_1 and ρ_2 be environments. Then, the *composition* of ρ_1 and ρ_2 (denoted $\circ$) is defined thus:

$$\rho_1 \circ \rho_2 = \lambda x.\ \rho_1(x) = none \rightarrow \rho_2(x); \rho_1(x)$$

2.3.2 The Meaning Function

The denotation function for Λ is given in figure 1. We use c to range over numerical and Boolean constants, x and *id* to range over identifiers, ρ to range over environments, and E to range over general Λ terms.

The meaning of our coercion operators is given in terms of an auxiliary function, $\mathcal{D}$, that captures our notion of *evaluation environment.* Every expression is associated with an environment image that can be captured and manipulated via the coercion operators. If $e \in \Lambda$, then $\mathcal{D}[\![e]\!]_\rho$ yields the environment image of e when evaluated in environment ρ. We give the definition of $\mathcal{D}$ in figure 2.

The environment image of a constant is simply its lexical environment; the environment image of an identifier is the environment image of its binding-value. A record has as its environment image the join of its lexical environment with the bindings it defines locally. The environment image of a λ-abstraction is the function's lexical environment augmented with a binding for its formal parameter to the distinguished value *formal*. The lexical environment of a function joined with the environment that associates a binding of the function's formal to the value of the actual defines the environment image of an application expression.

$$[\![\;]\!] : \Lambda \rightarrow \mathbf{Env} \rightarrow \mathbf{V}$$

$$\begin{array}{lcl}
[\![c]\!]_\rho & = & c \\
[\![x]\!]_\rho & = & proper(\rho(x)) \\
[\![[id_1 = E_1, id_2 = E_2, \ldots, id_n = E_n]]\!]_\rho & = & fix(\lambda f.[id_i \mapsto [\![E_i]\!]_{f \circ \rho}](\lambda x.none)), 1 \leq i \leq n \\
[\![\lambda x.E]\!]_\rho & = & \lambda v.[\![E]\!]_{\rho \circ \lambda id.id=x \rightarrow v;none} \\
[\![(E_1\ E_2)]\!]_\rho & = & [\![E_1]\!]_\rho \mid_{\mathbf{V}\rightarrow\mathbf{V}} ([\![E_2]\!]_\rho) \\
[\![E_1 \rightarrow E_2; E_3]\!]_\rho & = & [\![E_1]\!]_\rho \mid_{\mathbf{B}} \rightarrow [\![E_2]\!]_\rho; [\![E_3]\!]_\rho \\
[\![E_1 \bullet E_2]\!]_\rho & = & [\![E_1]\!]_\rho \mid_{\mathbf{Env}} \circ [\![E_2]\!]_\rho \mid_{\mathbf{Env}} \\
[\![E.id]\!]_\rho & = & [\![E]\!]_\rho \mid_{\mathbf{Env}} (id) \\
[\![E_1 \uparrow E_2]\!]_\rho & = & [\![E_2]\!]_{\mathcal{D}[\![E_1]\!]_\rho \mid_{\mathbf{R}}} \\
[\![\downarrow E]\!]_\rho & = & (\lambda v_1.\lambda v_2.[\![[\,\mathtt{val} = v_1,\ \mathtt{env} = v_2]]\!]_\rho)[\![E]\!]_\rho\ \mathcal{D}[\![E]\!]_\rho
\end{array}$$

Figure 1: The meaning function for Λ.

$$\mathcal{D}[\![\;]\!] : \Lambda \rightarrow Env \rightarrow Env$$

$$\begin{array}{lcl}
\mathcal{D}[\![c]\!]_\rho & = & \rho \\
\mathcal{D}[\![x]\!]_\rho & = & \mathcal{D}[\![[\![x]\!]_\rho]\!]_\rho \\
\mathcal{D}[\![[id_1 = E_1, id_2 = E_2, \ldots, id_n = E_n]]\!]_\rho & = & \rho \circ [\![[id_1 = E_1, id_2 = E_2, \ldots, id_n = E_n]]\!]_\rho \\
\mathcal{D}[\![\lambda x.E]\!]_\rho & = & \rho \circ \lambda id.id = x \rightarrow formal; none \\
\mathcal{D}[\![(E_1\ E_2)]\!]_\rho & = & \lambda id.\mathcal{D}[\![E_1]\!]_\rho(id) = formal \rightarrow [\![E_2]\!]_\rho; \mathcal{D}[\![E_1]\!]_\rho(v) \\
\mathcal{D}[\![E_1 \rightarrow E_2; E_3]\!]_\rho & = & [\![E_1]\!]_\rho \mid_{\mathbf{B}} \rightarrow \mathcal{D}[\![E_2]\!]_\rho; \mathcal{D}[\![E_3]\!]_\rho \\
\mathcal{D}[\![E_1 \bullet E_2]\!]_\rho & = & \mathcal{D}[\![[\![E_1 \bullet E_2]\!]_\rho]\!]_\rho \\
\mathcal{D}[\![E.id]\!]_\rho & = & \mathcal{D}[\![[\![E.id]\!]_\rho]\!]_\rho \\
\mathcal{D}[\![E_1 \uparrow E_2]\!]_\rho & = & \mathcal{D}[\![[\![E_1 \uparrow E_2]\!]_\rho]\!]_\rho \\
\mathcal{D}[\![\downarrow E]\!]_\rho & = & \mathcal{D}[\![[\![\downarrow E]\!]_\rho]\!]_\rho
\end{array}$$

Figure 2: Environment Representation of Λ terms

3 Manifestation of the Model

To make our discussion concrete, we define a language COE that supports the operators and structures described in the previous section. The syntax and semantics of COE most closely resembles that of Scheme [13]. In essence, COE is Scheme augmented with a record data type and the two coercion operators described above.

The semantics of the coercion operators in COE differs from their semantics in Λ because of Scheme's support for assignment. Consider the COE expression ($\downarrow$ e). The `env` field in the record-object yielded by evaluating this expression will contain a *reference* to the binding-values of all identifiers found in e's evaluation environment. Thus any changes made to `env` will be reflected in e's environment. Similarly, in an $\uparrow$ operation of the form ($\uparrow$ e_1 e_2) evaluated in COE, the binding-value of an identifier in the coerced environment image of e_1 will be a reference to the corresponding value in the record object that e_1 denotes. Here again, changes made to an environment binding by e_2 are reflected in the record structure denoted by e_1.

3.1 Some Simple Examples

The value yielded by the expression

```
(let ((f (↓  (let ((x 2
                    (y 5))
                 nil))))
   ((lambda (z) (+ z.x z.y)) f.env))
```

is `7`. The evaluation-environment of the $\downarrow$-operator's argument contains (among possibly other things) a binding for `x` to `2` and a binding for `y` to `5`; this environment is coerced into a record object which is then applied as the argument to the function, `(lambda (z) (+ z.x z.y))`.

As we mentioned above, side-effects on a coerced record are reflected in the corresponding environment. Thus, the value yielded by evaluating

```
(let ((x 1)
      (y (lambda (z) (+ z 1)))
      (z (↓ y)))
  (begin
     ((lambda (f) (set! f.x 10)) z.env)
     x))
```

is `10`; the assignment to `f.x` is reflected in the corresponding evaluation environment, *i.e.*, in the evaluation environment of the closure bound to `y`. It is precisely the ability to side-effect an environment via its data structure representation that justifies the use of the term *coercion* in describing our transformation operators. The component elements of the record object yielded by $\downarrow$ is shared with the binding-value of the corresponding element in its environment image. The $\downarrow$ operator simply manifests the environment image within the value domain of the language.

Since COE is based on a programming model that permits records to be explicitly denoted, we can define a record and use it as the evaluation environment of an expression via $\uparrow$. For example, the evaluation of

```
(↑ [ x = 2, y = 3 ]  (+ x y))
```

yields `5`. The coerced environment image of the record object yielded by evaluation of the expression `[ x = 2, y = 3]` is used as the binding environment in the evaluation of the expression `(+ x y)`. As we mentioned above, free names referenced during the evaluation of $\uparrow$'s second argument not defined in the environment-image of its first argument are resolved in the expression's lexical environment. Thus, evaluating

```
[ b = 3,
  f = (lambda (x) (↑ x (+ a b)))
  (f [a = 2])]
```

yields

```
[ b = 3,
  f =  closure of (lambda (x) (↑ x (+ a b))),
  5 ]
```

A lambda expression that is the second argument to $\uparrow$ has its free variables evaluated first in the environment-image of the $\uparrow$ operator's first argument and then in the context of its lexical environment. The environment captured by a lambda closure is superseded by a user-specified environment whenever a lambda expression is evaluated in context of an $\uparrow$ operation.

Thus, the value of the expression

```
(let ((x 1)
      (z 2))
  (let ((y (↑ [z = 20] (lambda () (+ x z)))))
     (let ((x 3)
           (z 4))
        (y))))
```

is `21`. The free variable `x` in the `lambda` expression is bound to its value in the function's lexical environment whereas the reference to free variable `z` is resolved relative to its binding-value in the environment image of the record expression.

4 Paradigms and Applications

Building and manipulating environments is a fundamental part of any computation; most languages, however, come equipped with only a small number of (fairly-limited) namespace devices to help the user manipulate his naming environment. The manner in which these devices are constructed rarely allow a computation direct access to the naming environments it builds. In the remainder of this section, we investigate the ramifications of allowing computations direct control over the naming environments they access.

4.1 Binding Protocols

Most languages that come equipped with a default binding protocol rarely provide facilities by which this protocol can be overridden in a semantically-clean way. Scheme is a good case in point. Scheme's primary environment-building structure, the *closure*, is built and maintained by the underlying interpreter: users can't write down an expression that defines the representation of a closure, nor can they examine a closure-object from within Scheme. Thus, it 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 can't alter the environment in any way not originally prescribed by the language design. This is an important limitation in the expressivity of the language; it often necessitates Scheme dialects to provide either *ad hoc* constructs to realize other binding disciplines (*e.g.,* the `f` luid-let [1] construct to achieve dynamic-binding) or significant extensions to the base language (*e.g.,* extensions for supporting late-binding and object-based programming [3, 9, 18]).

COE, like Scheme, also represents functions in terms of closures. The fact that the language provides operations to explicitly capture an environment, however, makes the lexical-binding rule logically unnecessary. We could, in other words, have ascribed a purely dynamic binding semantics to lambda expressions and used this binding protocol to define a fully lexically-binding variant. Consider an alternative semantics for lambdas, one in which lambda expressions are viewed as simple constants. Under such a semantics, the evaluation of a lambda expression simply yields the text representation of that expression. In the absence of any environment-capturing operations, the resulting language would be capable of only supporting dynamic binding. In COE, however, we can implement higher-order lexically-scoped functions on top of a dynamic binding protocol *without* having to extend or alter the underlying interpreter.

To see why, consider an extension to the base language that includes a dynamically-binding version of **lambda**, call it **lambda**$_d$. We wish to show that given **lambda**$_d$ we can specify the meaning of **lambda**. The basic idea is simple: an expression of the form:

```
(lambda (x1 x2 ... xn) Exp)
```

is rewritten as a record:

```
[ τ1 = (↓ nil),
  τ2 = (lambda_d (x1 x2 ... xn) Exp) ]
```

(τ_1 and τ_2 are assumed to be fresh identifiers.)

This record is, in effect, a closure – it consists of two parts, a representation of an environment and the text of the function: it can be passed freely, embedded within data structures, or returned as the result of an application. The expression `(↓ nil)` evaluates to the lexical environment of the **lambda** expression. In the absence of any coercion operations τ_2, when applied to arguments, will retrieve the binding-values of free names in its body based on the apply-time environment.

We can write an application expression that essentially behaves as though its first argument were statically scoped by using the ↑ operator. An application of the form, `(e1 e2 ... en)` is equivalent to the following expression:

```
(let ((τ3 e1)
      (arg1 e2)
      ⋮
      (argn-1 en))
  (↑ τ3.τ1.env (τ3.τ2 arg1 arg2 ... argn-1)))
```

`Arg1`, ... `argn-1` are fresh names introduced to avoid the unwanted capture of free names occurring in the actuals by ↑. The above expression evaluates the application relative to the function's lexical environment.

The particular transformation of functions and applications shown above is basically the same as one which would have been performed by a Scheme interpreter [27] with one important difference: in Scheme, there is no primitive representation of a naming environment nor are there any primitive operations that correspond to either the ↑ or ↓ operators. The Scheme meta-circular evaluator must maintain and update its environment image explicitly. In COE, environments have a well-defined representation and can be manipulated directly. Thus, the specification of lexical-binding and closure application can be given via simple rewrite rules; there is no need to define a complete evaluator in order to specify their semantics.

The same technique used to build a closure can be used to specify an arbitrary evaluation environment; for example, if we wish to evaluate a function in the context of a user-specified library we can do so by building a record of the same sort as above:

```
[ τ1 = L, τ2 = (lambda_d ...) ]
```

L is an expression that yields a record containing the bindings defined by the library. Applying this closure as we did above causes the **lambda**$_d$ expression to evaluate in the context of the bindings defined by this library object. In general, the object bound to τ_1 may be the result of a complex expression that builds and composes records; the result of evaluating this expression is used as the evaluation environment for the function bound to τ_2. The ability to evaluate an expression in the context of a user-specified environment is also a key requirement in an object-based programming methodology; we discuss this issue in detail in the sections following.

Note that a slightly different translation scheme would allow us to get the effect of dynamic binding via a lexical binding protocol. An expression of the form:

```
(lambda_d (x1 x2 ... xn) Exp)
```

can be rewritten thus:

```
(lambda (Env)
   (↑ Env (lambda (x1 x2 ... xn) Exp)))
```

We rewrite a dynamically-binding function into a higher-order lexically-scoped one that takes as its argument the record image of the dynamic environment and returns a function that evaluates in the context of this environment. Thus, an application expression that is to be evaluated under a dynamic binding protocol:

```
(e1 e2 ... en+1)_d
```

is equivalent to:

```
(let ((τ1 (↓ nil))
      (τ2 e1))
  ((τ2 τ1.env) e2 ... en+1))
```

This translation differs from a purely dynamic binding protocol in one respect: free names referenced in the body of the function not present in the dynamic environment are resolved relative to their binding-value in the lexical environment. It's easy to augment the translation scheme to handle this case, but we omit the translation here.

4.2 Modularity

The front-end (FE) of most interpreted languages is implemented by a `read-eval-print` loop – the FE acts as a virtual machine that repeatedly reads a new input expression, evaluates it on the basis of the internal environment structure maintained by the `eval` procedure, and prints the result. Users usually do not have access to the internal state of `eval` – programs to access and manipulate the environment image of an interpreter session must usually be provided as part of the evaluator package.

The COE front-end, on the other hand, implements a parallel *transparent* evaluator – expressions input by the user are added as a new element on top of the current environment; old bindings are superseded by new ones by layering the new binding expression on top of the old one; the non-strict evaluation semantics of records allows input expressions to be evaluated in parallel upto the ordinary serialization rules imposed by the name evaluation rule.

The outline FE (ignoring issues of printing and formatting) can be written as follows:

```
[FE =
  (lambda (user-env io-stream)
     (let ((next (↑ user-env
                    (read (first io-stream)))))
        (FE (• user-env next)
            (rest io-stream))))]
```

Expressions input by the user are represented as strings in a stream called `io-stream`. (A `stream` is a (potentially) infinite queue. It is represented in COE as an abstraction whose representation allows elements to be appended to the end in constant-time. We provide four operations on streams: `make-stream` (which returns a new empty stream), `first` (which returns the head element, blocking if the stream is empty), `rest` (which returns the rest of the stream, blocking if the stream contains zero or one element), and `attach` (which adds a new element to the end.).)

`Read` coerces its input (which is assumed to be a string representation of a single element record) into a COE record object. This object is then evaluated in `read`'s dynamic environment. In the above example, the record yielded by `read` is evaluated relative to the bindings found in `user-env`. Because records have a non-strict evaluation semantics, each expression input may be evaluated concurrently with every other. Thus, if `io-stream` is structured as follows:

```
( "[y = 3]"
  "[f = (lambda (x) (+ x y)]"
  "[a1 = (f 2)]"
  "[y = 4]"
  "[a2 = (f 2)]" )
```

the corresponding structure of `user-env` after these expressions have been read (and evaluated) would be:

```
[ y  = 4,
  f  =  closure of (lambda (x) (+ x y)),
  a1 = 5,
  a2 = 5 ]
```

Because `lambda`s are lexically-scoped functions, rebinding `y` to `4` does not change the apply-time behaviour of `f`; changing the behaviour of `f` requires side-effecting `y`. If, instead of the binding declaration `y = 4`, the user input, `(set! y 4)`, the second application of `f` would have used the new value of `y`. This property of the COE top-level front-end is in contrast to the behaviour of most Lisps but is consistent with other lexically-scoped interpreted languages such as ML [23].

Languages that use program objects such as packages or classes would be hard-pressed to support this kind of structure because they provide no operations to compose new environments dynamically. For example, given a Simula or Smalltalk class C, one cannot dynamically construct a new class C' that differs from C based on conditions known only at runtime. This is a fundamental requirement in the above example – each new expression input results in the construction of a new environment. Note that simply modeling environments as records would also not suffice. A record is a simple data object and its standard semantics does not permit its bindings *implicitly* to affect the evaluation of other expressions.

Language interpreters are arguably esoteric, but the modularity requirements they impose are found in a number of other paradigms as well. An object-oriented, inheritance-based programming style is a good example. There are several competing paradigms for general object-based programming: in Simula-67[12] instances are similar to records with function-valued components and message-passing is realized as field selection and application over these records; Smalltalk[17] (followed by CommonLoops[9] and Flavors[18]) treats message-passing as function call. Amber[10, 11] also models objects as records, but expresses inheritance as subtype relations among these records.

We pose an alternative view: in COE an object is considered a record, but inheritance is captured by simple record composition. The ↑ operator is used to coerce a record object into an evaluation environment. Message-send is realized by evaluating an expression in the context of the environment-image of a record object. In a coercion-based model, then, inheritance is viewed as essentially a namespace management problem; the inheritance hierarchy specifies a namespace that is composed from a collection of records that may define different bindings for the same name.

Thus, in thinking of objects as records, we see that an instance of a subclass in an inheritance hierarchy is simply a record that is the "fusion" of the environments defined by the associated instances of all its super-classes. The inheritance hierarchy determines how the record is to be constructed: nameclashes between a subclass instance and a superclass instance are always resolved in favour of the subclass. The coercion operation allow us to transform simple record objects into environments.

Let O_1 and O_2 be two classes such that O_1 is a subclass of O_2. Let O_1 define methods and instance variables m_1, m_2, $\ldots$, m_j and let O_2 define methods and instance variables n_1,n_2,$\ldots$,n_k and assume that there exists a non-empty intersection of method/instance variable names defined by these two objects. We represent O_1 as a record:

```
O1 = [ create =
          (lambda ()
             (• O2.(create)
              (• O1.(instance-vars)
               O1.methods)))

       instance-vars =
          (lambda ()
             [ O1's instance variables ]

       methods = [ O1's methods ] ]
```

O_2 is structured similarly. Evaluating (O_1.create) yields a record that contains a fresh copy of O_1's instance variables, O_1's methods and the instance variables and methods of O_2 (and all its superclasses). Note that because of the semantics of "•", instance variables and methods defined in O_2 (and its superclasses) also defined by O_1 are superseded with O_1's definition in all of O_1's instances. Thus, if O_1 and O_2 both define a method named M, bound to definitions D_1 and D_2 resp., the record returned as a result of evaluating (O_1.create) will contain a binding for M to D_1, not D_2.

Message-passing in this model is simply expression evaluation within the environment-image of the record object. If I_1 is an instance of O_1, then (↑ I_1 *exp*) evaluates *exp* using the instance variables and methods defined in I_1; if M occurs free in *exp*, and I_1 contains a binding for M, then the value of M in *exp* will be the binding value of M in I_1. Changes made to environment variables within *exp* that correspond to field-names in I_1 become visible to other expressions that subsequently access I_1.

The Smalltalk-style expression[3]

```
(send object method args)
```

is, therefore, represented in COE as

```
((↑ object method) args)
```

Because of the non-strict semantics of records, the system we have described here is essentially a *parallel* object-based system: many objects may send (and receive) messages to (and from) one another concurrently. We expand on this point below. Note also that modeling objects in terms of records also allows us to support multiple-inheritance; the order in which the multiple superclasses of an object are layered determines how method and instance variable name clashes between superclasses are to be resolved. This is essentially the same view taken in [10].

As a final point, note that the object returned as a result of interpreter-session is structurally the same as the objects built in an inheritance system. Both define a complex naming environment. In the inheritance example, the record/environment structure was created at the time of object instantiation and object creation allowed superclass methods to be superseded by subclass ones; in the front-end example, the environment structure is built recursively allowing new definitions to supersede old ones.

4.3 Reflection and Non-Strictness

Reflection [29, 26] refers to an activity in which an executing process can examine its evaluation environment, store and continuation. COE supports a restrictive form of reflection that allows an expression to examine its evaluation environment. Unlike standard reflective systems, however, the language's non-strict evaluation semantics allows an expression to examine its evaluation environment *concurrently* with the manipulation of the same environment by other executing expressions. We discuss the implications below.

A process daemon is a passive process that watches a program or data structure for interesting developments. It's difficult to support the construction of such processes in conventional languages because of the enforced separation between program and data structures. On the other hand, it's easy to implement processes of this kind in the framework of a model that permits computations access to their evaluation environment. Consider, as a simple example, the implementation of a daemon process that is to print a message whenever a user redefines the keyword "lambda" at the top-level (see figure 3).

When applied, the function waits for a new element to be added onto the environment; this information is conveyed through a data stream named `signal`. When a new input element has been added, the daemon checks whether the environment now contains a definition for "`lambda`" by evaluating the expression `my-env.lambda` where `my-env` is defined to be the join of a record containing a dummy definition of `lambda` with `user-env`. It prints an appropriate message whenever the result yields a definition for `lambda` that is not identical to the dummy one. We can install this daemon within an interpreter session that is managed by a front-end similar to the one described earlier. However, we need to change the definition of the front-end slightly so that (a) it builds the environment object via side-effect rather than recursion and (b) it builds a data stream indicating whenever a new element has been added to the current environment object:

```
[FE =
  (lambda (user-env io-stream signal)
     (let ((next (↑ user-env
                    (read (first io-stream)))))
       (begin
          (attach signal t)
          (set-record user-env
                      (• user-env next)
          (FE user-env
              (rest io-stream)
              signal))))))]
```

(`Set-record` mutates the object referenced by its first argument with the value yielded by its second.)

Suppose, given this implementation, the user inputs

```
"[check-lambda = (redefine user-env
```

[3] Of course, the system we've described assumes lexical scoping of all method definitions. To implement Smalltalk-style dynamic name resolution requires passing as an extra argument to every method a self object that is environment representation of the method's receiver. Thus, if free names found in method definitions in O_1 are to be resolved relative to their bindings in O_2, for example, such methods would need to take self as mandatory argument where self is the environment image of O_2. The free names occurring in these methods are then resolved in the context defined by O_2.

```
[ redefine =
    (lambda (user-env signal)
       (letrec ((id (gensym))
                (loop
                   (lambda (signal)
                      (let ((my-env (• [lambda = id] user-env)))
                         (begin
                            (first signal)
                            (if (not (equal? my-env.lambda id))
                                (write "Redefining lambda"))
                            (loop (rest signal)))))))
          (loop signal)))]
```

Figure 3: Parallel Process Daemons

```
signal)]"
```

to the interpreter built by the application of `FE`. `Check-lambda` can monitor the environment-object and `io-stream` of the evaluator responsible for its evaluation. New data values that augment `user-env` are visible to this application despite the fact that `check-lambda` exists within `user-env`. Many different monitors can be written that all examine `user-env` notified via `signal` whenever a new element is added. Note that even though `check-lambda` runs forever, the FE doesn't hand – because FE creates a non-strict record object, it is ready to accept new input even as previously input expressions continue to evaluate.

The ability to build a reflective daemon process comes fundamentally from the ability to treat program structures as data objects via coercion. Because the environment defined by a program structure can be treated as a simple non-strict data object, one can examine the internal structure of a program even if it is still in mid-evaluation. Viewing programs as *transparent* data objects encourages an interesting program methodology not supported by other programming models. For example, one can write a program essentially unencumbered by calls to i/o routines. Programmers are free to drape this program with routines that monitor its evaluation, format and display results as they see fit. Many different display routines can be written for the same program; the original core is left untouched. This style of programming, in essence, is no different from the reflective daemon described above: an i/o routine is a daemon that monitors the evaluation of a program structure.

4.4 Parallelism and Coercion

As a final example, we consider the realization of an actor-based programming model [4, 19] within our base language. In an actor-based paradigm, the fundamental computational entities are long-lived *concurrent objects* that communicate through message-passing. We can express the essence of the actor model in COE by organizing an actor program as a collection of concurrent processes that communicate with one another via specified data streams. The coercion operators allow us to examine the current evaluation environment of any given actor from the outside by treating the actor's environment as a data object, and by using the bindings defined within a given actor to affect the evaluation of other actors. This latter property is especially useful, as we will see, if we wish to dynamically compose different actor subsystems together.

The context of our example will be a translation of a static dataflow program (implemented in terms of a dataflow graph) into COE[4]. The nodes in such a graph are actors; they have long-lived local state and communicate with one another via a simple form of message-passing. The COE version of the dataflow graphs is to be faithful to the semantics of the dataflow actors with respect to the synchronization and firing rules obeyed by these actors.

The translation treats nodes in the dataflow graph as perpetually running process monitors and edges as streams. Each process watches the stream corresponding to the input edges for the node it represents and computes a result based on input values found in these map streams. These results are then written to the stream corresponding to the node's output edge. Monitors execute asynchronously (in the same way that actors in a real dataflow system do). In the particular translation given here, acknowledgment arcs between nodes are not used: data written onto streams are queued; the translation guarantees that the order in which output values are emitted by a node is preserved when writing onto the appropriate edge by explicitly serializing the writing of an output value with the reading of new inputs[5].

Consider the following program fragment written in Val[2] to compute the factorial of a number:

Function *Factorial* (n : **integer returns integer**)
    for i : **integer** := 0;
        p : **integer** := 1;
    do if $i = n$ **then** p
        else
            iter $i := i + 1$;
                $p := p * i$
            enditer
        endif
    endfor

[4] For our purposes, a static data flow language is one in which the structure of the base language graphs is fixed at compile time; there are no function application operators that can instantiate new copies of function graphs at runtime. Iteration is supported by allowing cyclic graphs to be constructed. Readers unfamiliar with the dataflow model of computation should consult [5] which gives a comprehensive introduction to the subject.

[5] In other words, this simulation assumes unlimited queuing on edges.

endfun

The `iter` construct creates a new "local" environment for `i` and `p` and evaluates the expressions to which they are bound in the context of this new environment.

A possible translation of this function into a static dataflow graph representation is shown in Figure 4.4. Edges entering into the sides of actors are signals – they generate boolean tokens. `True` and `False` gates pass their input only if the current value on their signal line is either **true** or **false**, resp.; they consume their input otherwise.

The corresponding representation in COE is given below:

```
[ fact =
    (lambda (n)
       (let ((graph REP))
         (begin
            (attach graph.i 0)
            (attach graph.p 0)
            (↓ graph)))) ]
```

where `REP` is a record:

```
[ i = (make-stream),
  p = (make-stream),
  answer = ⋆,
  edge1 = (make-stream),
  edge2 = (make-stream),
  edge3 = (make-stream),

  false-gate  =
    (lambda (input signal result)
       (begin
          (if (= (first signal) "false")
              (attach result
              (first input)))
          (false-gate (rest input)
                      (rest signal)))),

  =actor =
     ((lambda (i n)
        (begin
           (if (= (first i) n)
               (attach edge1 "true")
               (attach edge1 "false"))
           (=-actor (rest i) n)))
       i n),

  true-gate =
     ((lambda (p edge1)
         (begin
            (if (= (first edge1) "true")
                (set! answer (first p)))
                (true-gate
                   (rest p)
                   (rest edge1))))
       p edge1)

   false-gate-1 = (false-gate i edge1
                              edge2),
   false-gate-2 = (false-gate p edge1
                              edge3),

   +-actor =
      ((lambda (edge2)
          (begin
             (attach i (1+ (first edge3)))
             (+-actor (rest edge2))))
        edge3)

    *-actor =
       ((lambda (edge3 i)
           (begin
              (attach p (* (first edge3)
                           (first i)))
              (*-actor (rest edge3)
                       (rest i))))
        edge4 i)) ]
```

It is often convenient to allow the binding-value of names defined within a record to be supplied by expressions found outside the record. To support this facility, we provide an explicit synchronization mechanism (denoted $\star$) that acts as an unbound value; an expression that accesses a record field bound to $\star$ blocks until some other expression replaces the $\star$ with a non-unbound value. Insofar as it provides an explicit synchronization mechanism, unbound values in COE correspond roughly to I-structures [6] found in Id [24] or logical variables [22, 30] found in logic languages.

Each actor executes the same high-level process repeatedly: (1) wait for new input, (2) attach result to the output stream and (3) recurse. For example, the `+-actor` waits for a new value for `i` to be attached to `edge2` by `false-gate-1`. Once a value is written, the actor increments it and attaches the result to its output stream and waits again for new input. Despite the fact that all actors and gates can run asynchronously, the serialization introduced by the `begin` form guarantees that an output will not be produced until the corresponding inputs are received. Moreover, because every edge has only one producer, it is easy to see that merging of output values from different nodes cannot occur. The final result is given by the true gate which evaluates the `set!` expression that stores the result into `answer`. The true gate is activated only when all iterations have completed, *i.e.*, only when `i = p`.

We can probe the state of any given actor using the ↑ and ↓ operators. Suppose we wish to monitor the construction of various edges during the evaluation of `(fact 10)`. To do this, we simply evaluate:

```
(let ((fact-env (fact 10).env))
   (↑ fact-env  inspect edges))
```

The translation given here is limited in one respect relative to a more general actor model: the static dataflow graph has a fixed number of actors; it is not possible to dynamically change or compose the set of actors in a simple static dataflow program. COE however is not bound by this limitation and it is straightforward to extend the base dataflow model to permit dynamic composition and generation of actors.

For example, suppose we require the stream of integers produced by the `+-actor` to be channeled as an input to another dataflow graph; we'd like to do this dynamically and *non-invasively* without altering the `fact` program. Dynamic composition of actor systems is possible using coercion. The record object that contains the individual actors in our program is coerced into an environment; once the coercion is performed, we can evaluate another program in the context of the bindings found in this environment. Thus, a function that siphons the elements produced by the

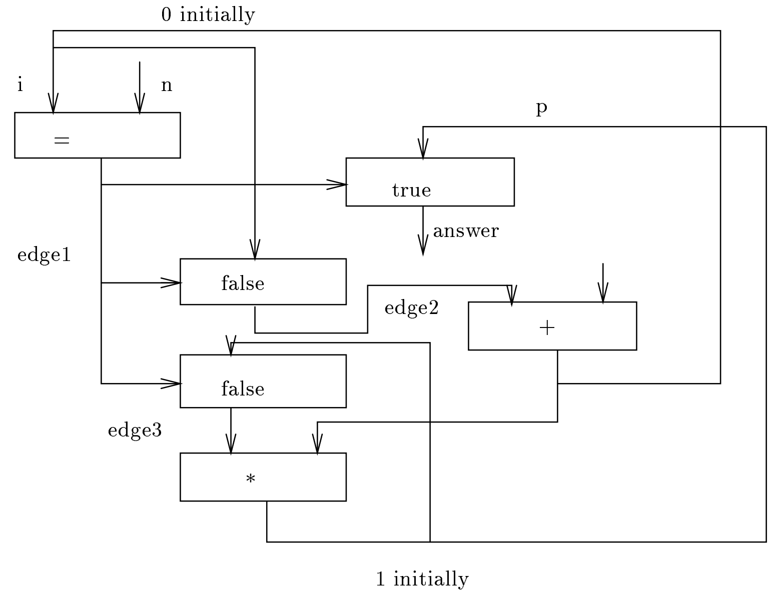

Figure 4: A Static Dataflow Graph for Factorial

`+-actor` (in the activation `(fact 10)`) to another (unrelated) dataflow program that multiplies these elements by 10 can be written as follows:

```
(let* ((fact-env (fact 10).env))
       (input-stream
           fact-env.edge3))
   (mult-10 input-stream))
```

The actor-based model described here is a simpler realization of the standard model formalized in [4]. Actors traditionally have been represented in terms of a set of components that represent both the computational as well as the communication behaviour of the object they manifest. The communication of an actor consists of the set of all other actors to which this actor responds. An actor's computational behaviour may include the monitoring of other actors, the creation of new "replacement" actors or the evaluation of a simple expression in response to input becoming available on the mail queue. In the COE framework, the specification of an actor is separated from the specification of its communication: an actor is an element of a record structure. Its state and environment may be examined by other actors via the coercion operators. The communication medium is represented via streams which are themselves represented as a recursive (potentially infinite) list of records (*i.e.,* actors). Abstracting the essence of actor behaviour (*i.e.,* the ability to model objects as computational entities with *visible* state) from communication (*i.e.,* stream generation) results in a simpler but no less expressive programming model.

5 Conclusions

In its support for non-strict evaluation of record objects COE's operational semantics most closely resembles various dataflow and graph reduction languages [21, 25, 28]. The coercion operators in COE distinguish it from these languages in obvious ways however; in terms of COE's support for direct manipulation of naming environments, it bears a strong resemblance to Symmetric Lisp [15, 16, 14], a non-strict language that provides support for first-class naming environments. Symmetric Lisp is based on a programming model that unifies program and data structures via a single environment generating mechanism. There is a strong similarity between COE's $\uparrow$ operator and Symmetric Lisp's `with` expression; both constructs in essence use the bindings defined within a user-specified record structure to influence the evaluation environment of other expressions. One can view COE as an extension of the Symmetric Lisp effort insofar as it permits arbitrary record/environment transformations.

The goal of providing a uniform basis for reasoning about a number of superficially-different modularity constructs motivated our investigation into the semantics of a model that supports explicit representation of naming environments. The symbiotic interaction of parallelism with first-class environments leads to a number of interesting paradigms; the structure of a process daemon or the organization of a parallel object-based system are two cases in point. In [20], we present other examples illustrating the utility of the model, *e.g.,*the construction of a guarded horn clause logic system, the implementation of a parallel blackboard structure, and the design of a monolingual parallel programming environment. We plan to continue investigation of the semantics and implementation of languages that are based on the model described here; an extensive implementation effort is currently under way.

References

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

[2] William Ackerman and Jack Dennis. VAL – A Value-Oriented Algorithmic Language: Preliminary Reference Manual. Technical Report 218, MIT, 1979.

[3] 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.

[4] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* PhD thesis, MIT Artificial Intelligence Laboratory, 1985. Published as AI-TR-844.

[5] Arvind and David Culler. *Dataflow Architectures*, volume 1, pages 225–253. Annual Reviews Inc., 1986.

[6] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction.* Springer-Verlag, 1986. Lecture Notes in Computer Science, Number 279.

[7] H. Barendregt. *The Lambda Calculus.* North-Holland, 1981.

[8] J.G.P. Barnes. An Overview of Ada. *Software Practice and Experience*, 10:851 – 887, 1980.

[9] Daniel Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops:Merging Lisp and Object-Oriented Programming. In *Object Oriented Programming Systems, Languages and Applications*, pages 17–30, September 1986.

[10] 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.

[11] Luca Cardelli. Amber. Technical Report 11271-840924-10TM, AT&T Bell Laboratories, 1984.

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

[13] 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.

[14] David Gelernter and Suresh Jagannathan. A Symmetric Language. Technical Report YALEU/DCS/RR-568, Yale University, May 1989.

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

[16] David Gelernter, Suresh Jagannathan, and Thomas London. Parallelism, Persistence and Meta-Cleanliness in the Symmetric Lisp Interpreter. In *SIGPLAN '87 Conf. on Interpreters and Interpretive Techniques*, 1987.

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

[18] R. Greenblatt, T. Knight, J. Holloway, D. Moon, and D. Weinreb. The LISP Machine. In *Interactive Programming Environments*, pages 326–352. McGraw-Hill, 1984.

[19] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artifical Intelligence*, 8(3):323–364, 1977.

[20] Suresh Jagannathan. *A Programming Language Supporting First-Class, Parallel Environments.* PhD thesis, Massachusetts Institute of Technology, December 1988. Published as LCS-Technical Report 434.

[21] Thomas Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Goteborg, Sweden, 1987.

[22] R. Kowalski. Algorithms = Logic + Control. *Communicatiosn of the ACM*, 22(7):424–436, July 1979.

[23] Robin Milner. The Standard ML Core Language. Technical Report CSR-157-84, Edinburgh University, 1984.

[24] Rishiyur Nikhil. ID Reference Manual (Version 88.0). Technical report, MIT, 1988. Computation Structures Group Technical Report.

[25] Simon L. Peyton Jones, Chris Clack, Jon Salkild, and Mark Hardie. GRIP – A High Performance Architecture for Parallel Graph Reduction. In *Proceedings of the 3rd. International Conference on Functional Programming and Computer Architecture, Portland, Oregon*, September 1987.

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

[27] Guy Steele Jr. and Gerry Sussman. The Art of the Interpreter, or the Modularity Complex. Technical Report AI-TM 453, MIT Artificial Intelligence Laboratory, 1978.

[28] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software - Practice and Experience*, 9:31–49, 1979.

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

[30] David H. Warren. Logic Programming and Compiler Writing. *Software Practice and Experience*, 10(2):97–127, February 1980.