An Introduction to Reflection-Oriented Programming

Jonathan M. Sobel *Daniel P. Friedman*[*]
Computer Science Department, Indiana University
215 Lindley Hall, Bloomington, IN 47405
{jsobel,dfried}@cs.indiana.edu

Abstract

Most accounts of reflection are in an interpreted framework and tend to assume the availability of particular pieces of the state of a program's interpretation, including the current source code expression. This paper presents a *computational* account of reflection, drawing a distinction between the meta-level manipulation of data or control and the mere availability of meta-circular implementation details. In particular, this account does *not* presume the existence of program source code at runtime.

The paper introduces a programming paradigm that relies on reflective language features and draws on the power of object-oriented programming. Several examples of the use of these features are provided, along with an explanation of a translation-based implementation. The examples include the measurement of computational expense, the introduction of first-class continuations, and the modification of the semantics of expressed values, all in the framework of reflection-oriented programming.

1 Introduction

Intuitively, reflective computational systems allow computations to observe and modify properties of their own behavior, especially properties that are typically observed only from some external, meta-level viewpoint. The concept of reflection is best understood by reaching back to the study of self-awareness in artificial intelligence: "Here I am walking down the street in the rain. Since I'm starting to get drenched, I should open my umbrella." This thought fragment reveals a self-awareness of behavior and state, one that leads to a change in that selfsame behavior and state. It would be desirable for computations to avail themselves of these reflective capabilities, examining themselves in order to make use of meta-level information in decisions about what to do next.

In this paper, a realization of this fundamental understanding of reflection is presented as a means to create a new programming paradigm in which already-compiled programs can be extended naturally by the addition of composable, reflective components. We call this new paradigm *reflection-oriented programming*. This research departs from previous work

[*]Research partially supported by National Science Foundation Grant #CCR–9302114.

by striving to distinguish between the notions of *computation* and *program interpretation.* It also borrows concepts from object-oriented systems, which can be seen as closely related to reflective systems in many ways. The main contributions of this paper are a concise framework for computational reflection, a series of examples of its use, and a translation-based implementation that leaves code open to further reflection even after it has been compiled.

In the next section, we review several other avenues of research about computational reflection and how our work relates to them. In section 3, we explain our approach to reflection, followed by a series of examples that demonstrate the power and utility of reflection-oriented programming. In section 4, we present one possible implementation strategy (the one we have used) for this system. In the final section, we discuss several potential research directions in reflection-oriented programming.

2 Background and Related Research

An obvious step toward providing the background for a detailed explanation of reflection-oriented programming would be to give a precise, formal definition of *computational reflection.* Such a definition is elusive, however, for a variety of reasons. One is that we still lack an entirely satisfactory or theoretically useful definition of *computation.*[1] Furthermore, the circularity inherent to the notion of reflection makes it hard to express a concrete, well-grounded definition.

Thus, descriptions of reflection in programming languages have avoided the hazy notion of computation and relied instead on the concrete concept of *program.* Instead of addressing computation about computation, they have been restricted to programs about programs. In such a framework, reflection is just a property of programs that happen to be about themselves. In this paper, the *process* of program execution serves as an adequate model for computation; a more complete treatment of the topic of computation is beyond the scope of this research.

2.1 Introspective Interpreters

Much of the research on reflection focuses on introspective interpreters. These are precisely the "programs about themselves" referred to above. A meta-circular interpreter is written in the language it interprets, with a few special forms added to the language to allow a program to access and modify the internal state of the interpreter running that program [8]. Specifically, the process of converting some component of the interpreter's state into a value that may be manipulated by the program is called *reification*; the process of converting a programmatically expressed value into a component of the interpreter's state is called *reflection.*[2] By reflecting code "up" to the interpreter, a program can cause that code to execute at the meta-level.

[1]Models like Turing machines and λ-calculi define *computability*, but they do not provide the sort of formal mathematical understanding of computation that we have for other concepts in programming languages, such as the *meaning* of a program or the *type* of a term.

[2]It can be confusing that the term *reflection* refers both to a specific activity and to a broad subject. When reflection is discussed in relation to reification, the more specific meaning is the one we intend.

Three notable examples of this genre of introspective computation are 3-Lisp [17, 18], Brown [7, 20], and Blond [3]. In 3-Lisp, Smith introduces the idea of a *reflective tower*, in which an infinite number of meta-circular interpreters run, each executing the interpreter immediately "below" it. Since a program may reflect any code up to the next higher level, including code that accesses the next level, it is possible for a program to run code at arbitrarily high meta-levels. The other introspective interpreters essentially do the same thing, with variations on what interpreter state is available and what effect reflective code can have on ensuing computation. In each case, the whole concept of reflection is knotted very tightly to interpretation—so tightly, in fact, that it is commonplace to assume the availability of original source code during the execution of reflective programs. (Debugging aids, such as the ability to trace the execution of the source program, are often cited as examples of what can be done with introspective interpreters.) There have been some attempts to escape from the world of the interpreter [1] and to formalize the sort of reflection performed by these introspective interpreters [8], but none of them overcome the extremely operational, implementation-dependent nature of this approach to reflection.

By focusing on processes, rather than programs, we can free ourselves from the introspective meta-circular interpreter. Our implementation of reflection-oriented programming does not presume an interpreted language or the existence of source code during program execution; it allows pre-compiled code to be extended with reflective behaviors.

2.2 Metaobject Protocols

One alternative to introspective interpreters has already grown out of object-oriented programming. In an object-oriented system, a program specifies the behavior of the objects of a class by specializing a set of methods with respect to that class (or its superclasses). The program invokes methods on objects without "knowing" precisely from which class they come, and the system chooses the most appropriate methods to run. A metaobject protocol [9, 11] extends an object-oriented system by giving every object a corresponding *metaobject*. Each metaobject is an instance of a *metaclass*. The methods specialized with respect to some metaclass specify the meta-level behavior of the object-metaobject pair, where meta-level behaviors include, for example, the workings of inheritance, instantiation, and method invocation. (Most real implementations do not really create a metaobject for every object; instead, they use a metaobject protocol for some subset of the objects in the system, such as the classes or functions.)

Metaobject protocols provide a powerful, flexible means of modifying the behavior of a language. Using such a tool, one might add multiple inheritance to a language that begins only with single inheritance, or even allow for the coexistence of several different kinds of inheritance and method dispatch. Some metaobject protocols even let a program have control over the way new objects are represented by the system.

By nature, a metaobject protocol focuses on the *objects* in a system. This correspondence between objects and metaobjects has caused many of the languages that use metaobject protocols to limit themselves to reflection about values. Since the role of most programs is to manipulate data (i.e., values, objects), this added power is quite useful, but a traditional metaobject protocol does not let a program say anything directly about control flow or other meta-level concepts that do not correspond directly to objects in the problem domain. In reflection-oriented programming, on the other hand, the correspondence is between reflective objects and computations. In fact, it is possible to understand reflection-oriented programming as a kind of "meta-computation protocol," which makes it a much more plausible framework for reflecting about computational properties.

2.3 Monads

Monads are category-theoretic constructs that first received the attention of the programming language community when Moggi [13, 14] used them to represent effect-laden computation in denotational semantics. Since then, they have seen growing acceptance as a practical tool, especially as a means of handling I/O and other side-effects in purely functional languages. Each monad is defined as a triple consisting of a type constructor and two operators. If the type constructor is T, the two operators (*unit* and *extension*) allow expressions over type α to be transformed into expressions over the type $T\alpha$. Values of type $T\alpha$ are usually taken to represent *computations* over values of type α. Thus, the use of monads allows a program to manipulate computations as values, which is a kind of reflection. Monadic reflection has been explored on its own [5, 14], but its relation to other forms of reflection is still largely unaddressed [12].

The study of monads strongly influences the concept of computation used here—especially the dissection of computations—and yet it is possible to define and use reflection-oriented programming without any reference to monads. On the other hand, the implementation presented here, which is not actually monadic, would look very familiar to anyone acquainted with monads. The deeper connections, if any, between this work and monads has yet to be explored. One exciting possibility is that reflection-oriented programming, which allows multiple reflective properties to be defined independently, could be formalized to address the issue of monad composition. (Currently, the only systems for combining monads are very narrow in scope and either non-extensible or can only be extended with much work and theoretical awareness on the part of the user [4, 10].)

3 Reflection-Oriented Programming

In reflection-oriented programming, we understand computation as process, rather than program. It is all right to think of a process as the execution of a program, as long as care is taken not to focus on some particular application of an interpreter to a source program. Computation is a program happening.

What does reflection about computations add, and what is lost, in comparison with other approaches? For one thing, many other reflective systems, especially introspective interpreters (but also some metaobject protocols), have enabled a program to refer to its own variables. But the concept of a program variable does not necessarily make any sense in the context of pure processes, once the original source code has been thrown away. One of the arguments to a reifying procedure in Brown, for example, is the program text that the reifying procedure was applied to. The purely computational view of reflection disallows such examinations of code. The text of a program is merely a means to a process. On the other hand, in purely computational reflection we gain the ability to refer to such concepts as computational expense, flow of meta-information, and the meaning of expressed values.

3.1 Reflection-Oriented Programming as a Paradigm

To better understand reflection-oriented programming as a paradigm, consider some programming task. Suppose that in addition to being able to write and edit a program to complete this task, it is also possible to examine and edit the source code of an interpreter for the programming language being used. In the course of programming, there might be several situations in which there exists a choice: make extra effort and go to great lengths to solve a problem in the user code, or modify and extend the interpreter. (This dilemma is also mentioned by Maes [11].)

The most familiar example of this situation might be the need for first-class continuations. When such a need arises, one can either convert one's entire program to continuation-passing style [15], or one can convert the interpreter to continuation-passing style and provide some means for the program both to reify the current continuation and to replace the current continuation with another one.

Measuring computational expense is another simple example. If a programmer needs some measure of the number of steps a program takes to complete a task, there are again two main options. The program can be written so that it updates a counter every time it takes a step (by threading an extra argument through the entire program—again necessitating continuation-passing style—or by modifying some global variable). Or the programmer can make similar modifications to the interpreter, along with provisions for user programs to examine and replace the current value of the interpreter's "uptime" counter.

In cases like these, it is probably simpler just to rewrite the program if the program is very small or very simple (or if the right automatic translation tools are available). If the program is at all long or complex, though, it could be much more convenient to modify the interpreter and provide an interface to the new features. As a starting point, let us associate the reflection-oriented paradigm with choosing to modify the interpreter and providing an appropriate interface to those modifications. The name *reflection-oriented programming* is justified on the grounds that, in this style of programming, the computation directly depends upon information being created and maintained at a meta-level, and on the grounds that the computation can access and affect that information.

Of course, in many practical settings, the programmer has neither access to the source code for an interpreter nor desire to use an interpreted language. Thus, we extend the reflection-oriented programming paradigm to be programming *as if* the interpreter had been modified. More precisely, we define reflection-oriented programming to be a programming style that uses any means available to extend the meta-level semantics of computation in order to avoid situations in which the local requirements of some program fragment lead to non-local rewriting of the program.

3.2 Reflection-Oriented Concepts

Since our approach to reflection-oriented programming is based on the concepts of object-oriented programming, we start by reviewing the way an object-oriented program works. In object-oriented programming, one writes a program so that its "shape" is not yet completely fixed. Instead, its shape is determined by the shape of the data with which it comes into contact. We can imagine throwing an object-oriented program at some data and watching the execution of the program wrap around that data, conforming to the contours of what it hits. Likewise, we wish to have a system in which a program can wrap its execution around the shape of the *computation* that it hits. Then we want to throw it at itself. To give computations some shape, we divide them into two categories:

1. *atomic* – The computation is completed in a single logical step.
2. *compound* – The computation requires the completion of subcomputations.

Some examples of atomic computations are numbers, references to variables, Boolean values, and the performing of primitive operations (after the computations corresponding to the arguments of those operations have already been completed). Some examples of compound computations are function applications, conditionals, and sequencing operations. In any compound computation, it is possible to identify some *initial* subcomputation. When the semantics of the source language provides a specification for ordering subcomputations, the initial subcomputation may be uniquely determined. Otherwise, it may be chosen arbitrarily from among the several subparts of the compound computation. We call the non-initial subcomputations the "rest" of the compound computation. If there is more than one non-initial subcomputation, then the rest of the subcomputations are, as a group, another compound computation.

Now computations have shape; how can reflective programs be written to wrap around that shape? We introduce the concept of *meta-information.* Meta-information consists of knowledge available—and maintained—at a meta-level. For example, if we are interested in knowing about the effort expended by a program as it runs, we would think of the current number of steps completed in a given dynamic context as meta-information. Our view of the relationship between meta-information and computations is shown in the following diagram:

$$\boxed{1} \Rightarrow \bigcirc \Rightarrow \boxed{2}$$

where the circle is some computation, and the numbered boxes represent the states of some sort of meta-information. The numbers themselves are merely labels. Given some prior state of meta-information, the computation occurs, and then we are left with some new state of meta-information. In our computational expense example, the first box would represent the number of steps executed prior to this particular computation (e.g., 237), and the second box would represent the number of steps executed altogether (e.g., 1090), including those steps within the computation.

In our system, we use a *reflective* to specify the behavior of some type of meta-information. A reflective, which is much like a class, describes the structure of its instances. The instances of reflectives, or *reflective objects*, are the carriers of meta-information. Thus, from a meta-level viewpoint, computations act as transformers of reflective objects. The *current* reflective object for a given reflective represents the meta-information as it exists at that instant. Like classes, reflectives can inherit structure and methods from other reflectives. When a computation is provided with meta-information—in the form of an instance of some reflective—the exact transformative effect on the meta-information is determined by methods specialized to that reflective or its ancestors.

So far, we have found that reflectives can be placed naturally into one of two groups: *simple* reflectives and *control* reflectives. Reflective concepts such as environments, stores, and computational expense can be modeled by simple reflectives. Reflection about the meanings of expressed values can be carried out by a special subset of simple reflectives called *value* reflectives. Control reflectives can model concepts pertaining to flow and control.

3.3 Notation

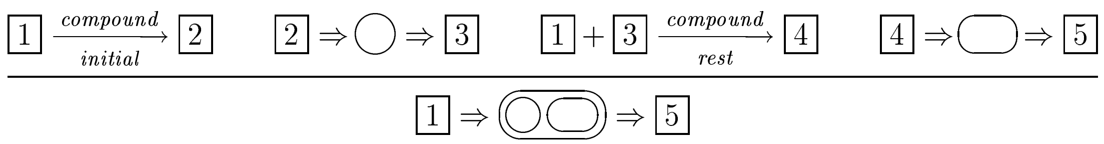
In the following sections, we presume an object-oriented language with generic functions. *<reflective>* is the convention for writing names of reflectives. Other types are simply capitalized, like *Integer*. Each parameter in a function definition can be followed by an optional specializer, which is a "::" followed by a reflective or some other type. *<reflective>* $\uplus$ $\{$*field*$, \ldots\}$ is a new reflective that extends *<reflective>* with new fields *field*$, \ldots$. Even if there are no new fields in the subreflective, we still write *<reflective>* $\uplus$ $\{\}$ to make it clear that the result is a subtype of *<reflective>*. If r is an instance of a reflective, and f is the name of a field in r, then $r[e/f]$ is an instance r' such that $r'.f = e$ and $r'.x = r.x$ for every other field x. *reify*(*<reflective>*) evaluates to the current instance of *<reflective>*. Evaluating *reflect*(*<reflective>*, r, e) causes the current instance of *<reflective>* to be r before evaluating and returning e. When *<reflective>* appears as text in an example, and not just as a meta-variable, it is intented to act as an empty base reflective type with no particular functionality. Application is left-associative, as usual.

3.4 Simple Reflectives

The behavior of a simple reflective's instances is governed by three methods. The first method, *atomic*, specifies what sort of transition the instances should make in atomic computations. The second method, *compound-initial*, specifies what sort of transition should take place upon entering the initial part of a compound computation. Finally, *compound-rest* specifies what meta-information gets passed on to the rest of a compound computation, given the reflective object produced by the initial subcomputation and the reflective object representing the state of meta-information prior to the whole compound computation. The following figures, which are in the form of proof rules, demonstrate how these methods fit together with the whole process of computation:

$$\frac{\bullet \rightarrowtail \mathbf{v} \qquad \boxed{1} + \mathbf{v} \xrightarrow{atomic} \boxed{2}}{\boxed{1} \Rightarrow \bullet \Rightarrow \boxed{2}}$$

The filled circle in this rule represents an atomic computation. We can read the rule as, "Given that an atomic computation produces $\mathbf{v}$ as its output, and given that the *atomic* method produces meta-information in state 2 when supplied with **v** and meta-information in state 1, we can conclude that the atomic computation causes a transition from state 1 to state 2 in the meta-information." For instance, if we define *atomic* to add one "tick" to the step counter of its argument when applied to instances of our computational expense reflective, then we can say that atomic computations take one computational step.



In the second rule, the oval with other computations inside represents a compound computation. The circle is the initial subcomputation, and the small oval is the rest of the compound computation. Each of these subparts may be atomic or compound. Continuing our example, we might make *compound-initial* add one tick to the step counter of its argument. The initial subcomputation starts with the meta-information produced by *compound-initial* and transforms it to a new state, possibly going through many intermediate states along the way, if the subcomputation is itself compound. Thus, the box labeled 3 represents the total number of steps taken by the initial subcomputation plus the one tick added by *compound-initial* plus however many steps had taken place prior to the whole compound computation. Both this sum and the original meta-information (labeled 1) are passed to *compound-rest*. In this example, we want *compound-rest* to return its second argument. Finally, the rest of this compound computation transforms the output of *compound-rest* into the ultimate result for the computation (labeled 5). The total number of steps for the compound computation is one more than the sum of its subcomputations. (If we wanted to model a system in which returning from subcomputations—popping the stack—represents an expense, we could make *compound-rest* add another tick to its second argument.)

The diagrams also make clear the separation between the meta-level behavior of information and base-level program behavior. The definitions of the three methods can be linked in *after* the program has already been compiled. Once the right "hooks" have been compiled into a program, it is possible to add new reflective behaviors at will, without modifying the original code. To make this idea more concrete, consider that the text of a program whose computational expense is to be measured is *exactly the same* as one in which no extra reflective information is being maintained. Furthermore, taking advantage of method selection, it is possible to define many different kinds of reflectives to be active at once.

To make this whole system truly reflective, we add operations *reify* and *reflect* to the language to allow meta-information to be reified or reflected on demand. For example, it would be possible for a debugger to track the expense of the program it runs, excluding the cost of the debugger itself. The debugger could reify the current expense-measuring reflective object upon entry into its own part of the computation. It could reflect the saved expense information back into the meta-level just before handing control back over to the user's program.

It is convenient to have default behaviors for *atomic*, *compound-initial*, and *compound-rest* that simply "thread" meta-information through the computations (like a store [16]). Thus, we build the following definitions into our system:

$$
\begin{array}{l}
\textit{<simple-reflective>} =_{\text{def}} \textit{<reflective>} \uplus \{\} \\
\textit{atomic}(v, \textit{outer}::\textit{<simple-reflective>}) =_{\text{def}} \\
\qquad \textit{outer} \\
\textit{compound-initial}(\textit{outer}::\textit{<simple-reflective>}) =_{\text{def}} \\
\qquad \textit{outer} \\
\textit{compound-rest}(\textit{outer}::\textit{<simple-reflective>}, \textit{inner}::\textit{<simple-reflective>}) =_{\text{def}} \\
\qquad \textit{inner}
\end{array}
$$

3.4.1 Example: Measuring Computational Expense

We have already discussed the measurement of computational expense in some detail in the preceding section. To implement such behavior, we need only these definitions:

$$
\begin{array}{l}
\textit{<runtime>} =_{\text{def}} \textit{<simple-reflective>} \uplus \{\textit{ticks}\} \\
\textit{atomic}(v, \textit{outer}::\textit{<runtime>}) =_{\text{def}} \\
\qquad \textit{outer}[\textit{add1}(\textit{outer}.\textit{ticks})/\textit{ticks}] \\
\textit{compound-initial}(\textit{outer}::\textit{<runtime>}) =_{\text{def}} \\
\qquad \textit{outer}[\textit{add1}(\textit{outer}.\textit{ticks})/\textit{ticks}]
\end{array}
$$

The inherited definition of *compound-rest* suffices. Now suppose we want to evaluate some expression E without counting its cost. We could use the *reify* and *reflect* operations like this:

$$
\begin{array}{l}
\textbf{let } r = \textit{reify}(\textit{<runtime>}) \\
\textbf{in let } \textit{result} = E \\
\quad \textbf{in } \textit{reflect}(\textit{<runtime>}, r, \textit{result})
\end{array}
$$

Here we capture the expense meta-information before evaluating E and save it as r. After evaluating E, we restore the old meta-information. The result from evaluating E is returned. Of course, any time we want to know the cumulative expense for the program, we can examine $\textit{reify}(\textit{<runtime>}).\textit{ticks}$.

3.4.2 Example: Using Environments in an Interpreter

The use of reflectives makes it possible to write a simple interpreter for an untyped call-by-value λ-calculus without passing an environment argument down in recursive calls to the interpreter. First we define an environment reflective.

$$
\begin{array}{l}
\textit{<env>} =_{\text{def}} \textit{<simple-reflective>} \uplus \{\textit{bindings}\} \\
\textit{compound-rest}(\textit{outer}::\textit{<env>}, \textit{inner}::\textit{<env>}) =_{\text{def}} \\
\qquad \textit{outer}
\end{array}
$$

The only way that an environment acts differently from the default store-like behavior of simple reflectives is that the rest of a compound computation is executed using the bindings that were in effect outside the compound computation (i.e., prior to the initial part of the compound computation).

Now suppose we have parsed λ-calculus terms into objects of types *Varref*, *Lambda*, and *App*. We can write an interpreter for these terms as follows:

```
$interp(exp{::}Varref) =_{\mathrm{def}}$
    $\mathbf{let}\ e = reify(<env>)$
    $\mathbf{in}\ lookup(exp.\mathrm{name}, e.bindings)$
$interp(exp{::}Lambda) =_{\mathrm{def}}$
    $\mathbf{let}\ e_1 = reify(<env>)$
    $\mathbf{in\ let}\ b = e_1.bindings$
        $\mathbf{in}\ \lambda a.\mathbf{let}\ e_2 = reify(<env>)$
            $\mathbf{in}\ reflect(<env>,$
                $e_2[extend(b, exp.\mathrm{formal}, a)/bindings],$
                $interp(exp.body))$
$interp(exp{::}App) =_{\mathrm{def}}$
    $interp(exp.\mathrm{rator})(interp(exp.\mathrm{rand}))$
```

We must use *reify* a second time inside the *Lambda* clause because some other reflective may have been derived from <*env*>; that is, the current instance of <*env*> may actually have more fields than just *bindings*, by virtue of inheritance. We must be careful not to throw away additional meta-information when we reflect a new instance back into the meta-level. (This is a typical issue in systems that support subtype polymorphism; it is not unique to reflection-oriented programming.)

The environment and expense reflectives can coexist, so that it is possible to measure the expense of evaluating λ-terms using *this* interpreter. More generally, many reflectives can exist simultaneously, so that an arbitrary amount of meta-information can be maintained by the system and accessed by programs.

3.5 Value Reflectives

It is useful to build in a special subreflective of <*simple-reflective*> for modeling the concept of expressed values, or values produced as the result of computations. By inheriting from <*value-reflective*>, it is possible to modify or extend the semantics of expressed values. By default, standard concrete values are returned from computations and passed on to continuations, but the creation of a new value reflective allows for the introduction of abstract values. For example, instead of producing the number 3, some computation may produce the type *Integer* as its value. Value reflectives are probably the most complex to manage, because changing the semantics of values necessarily has wide, systemic ramifications. Just as in any abstract interpreter [2], some or all of the language's primitive operations often need to be redefined. This, too, is possible with value reflectives. (For example, addition might have to be redefined so that *Integer* + *Integer* produces the value *Integer*, where *Integer* is a value, not a meta-level description of some number.)

An abridged version of the definitions of <*value-reflective*> and its methods appears below. In reality, the method definitions must be more complicated in implementation-specific ways in order to deal correctly with language primitives, etc., depending on how deeply reflection-oriented programming is embedded into the language.

```
$\textit{<value-reflective>} =_{\text{def}} \textit{<simple-reflective>} \uplus \{\textsf{base-value}\}$
$\textit{atomic}(v, \textit{outer}::\textit{<value-reflective>}) =_{\text{def}}$
    $\textit{outer}[v/\textsf{base-value}]$
```

As one would expect, the value coming from the outside is ignored in the default definition. The one field in the instances of this reflective acts as a "carrier" for what would be the standard (meta-level) expressed values in a meta-circular interpreter.

3.5.1 Example: Bounding the Possible Values of Integers

A very simple example of the power of value reflectives is the restriction of the domain of integers in a computation to be between -1000 and 1000.

```
$\textit{bound-to-1000} =_{\text{def}} \lambda i.\textit{min}(1000, \textit{max}(-1000, i))$
$\textit{<bounded-int>} =_{\text{def}} \textit{<value-reflective>} \uplus \{\}$
$\textit{atomic}(v::\textit{Integer}, \textit{outer}::\textit{<bounded-int>}) =_{\text{def}}$
    $\textit{outer}[\textit{bound-to-1000}(v)/\textsf{base-value}]$
$\textit{compound-rest}(\textit{outer}::\textit{<bounded-int>}, \textit{inner}::\textit{<bounded-int>}) =_{\text{def}}$
    let $t = \textit{inner}.\textsf{base-value}$
    in if $\textit{instance?}(t, \textit{Integer})$
      then $\textit{inner}[\textit{bound-to-1000}(t)/\textsf{base-value}]$
      else $\textit{inner}$
```

No new fields are added in the definition of *<bounded-int>*. This reflective merely modifies the behavior of integer values by overriding the default methods. (The new definition of *atomic* in our example applies only to integers; the inherited *atomic* still works for other types.) With these definitions in place, the computation corresponding to $600 + 600$ would produce 1000 as its value. Furthermore, in complex calculations such as

$$10^5 + 10^6 - 4572$$

all the intermediate values are subject to the bounding.

What happens if a program defines several disjoint subreflectives of *<value-reflective>*? For example, what if we define one value reflective to restrict the absolute values of integers to be less than 1000 and another (not as a subreflective of the first) to restrict integer values to be positive? One reasonable answer might be that such orthogonal definitions are simply disallowed. Instead, we have chosen to allow the existence of several independent value reflectives. At each step of the computation, the values in the various base-value fields are collected and passed, in turn, to the continuation. Of course, in the worst case, this can lead to an exponential increase in the number of steps a program takes! (An implementation can make some attempt to control this exponential explosion by recognizing duplicate values and passing only one of them to the continuation.)

3.6 Control Reflectives

Unlike value reflectives, which are really just simple reflectives that get special treatment, control reflectives are a completely different species. The methods associated with simple reflectives specify only how meta-information should flow through computations on a per-reflective basis. The methods associated with control reflectives, on the other hand, specify both how control itself should be managed in computations and how meta-information about control should be propagated. There are only two methods used to define the behavior of a control reflective: *atomic-control* and *compound-control*. Each atomic computation is encapsulated as a function that executes that computation when it is applied to an appropriate instance of a control reflective. Likewise, each compound computation is encapsulated as two functions, one for the initial computation and one for the rest of the subcomputations. Since it is possible for control to split into more than one thread during a computation—in the presence of multiple value reflectives, for example—each of these functions returns a *sequence* of control reflective objects, as can be seen in the default definition of *compound-control* below.

$$\begin{aligned}
&\textit{atomic-control}(\textit{outer}::\textit{<control-reflective>}, \textit{atom}::\textit{Function}) =_{\text{def}}\\
&\qquad \textit{atom}(\textit{outer})\\
&\textit{compound-control}(\textit{outer}::\textit{<control-reflective>}, \textit{initial}::\textit{Function}, \textit{rest}::\textit{Function}) =_{\text{def}}\\
&\qquad \textit{map/concat}(\textit{rest}, \textit{initial}(\textit{outer}))
\end{aligned}$$

The default definition of *atomic-control* applies the function representing the atomic computation to the current instance of *<control-reflective>*. The default definition of *compound-control* passes its reflective object to the function representing the initial subcomputation and then maps the function representing the rest of the subcomputations over the sequence of objects returned by the initial function. The resulting sequences are concatenated to form the final result.

3.6.1 Example: Implementing First-Class Continuations

An immediate use for control reflectives is the introduction of first-class continuations. First, we define a control reflective *<cont>* to model continuations; it has one field, used to hold the current continuation at any stage of the computation.

$$\begin{aligned}
&\textit{<cont>} =_{\text{def}} \textit{<control-reflective>} \uplus \{\textit{cc}\}\\
&\textit{atomic-control}(\textit{outer}::\textit{<cont>}, \textit{atom}::\textit{Function}) =_{\text{def}}\\
&\qquad \textit{map/concat}(\lambda x.\textit{head}(x.\textit{cc})(x[\textit{tail}(x.\textit{cc})/\textit{cc}]), \textit{atom}(\textit{outer}))\\
&\textit{compound-control}(\textit{outer}::\textit{<cont>}, \textit{initial}::\textit{Function}, \textit{rest}::\textit{Function}) =_{\text{def}}\\
&\qquad \textit{initial}(\textit{outer}[\textit{pair}(\textit{rest}, \textit{outer}.\textit{cc})/\textit{cc}])
\end{aligned}$$

The *compound-control* method passes to *initial* an instance of *<cont>* whose current continuation field has been supplemented with *rest*. In this way, the control flow "walks" down the leftmost (initial-most) branches of the computation tree until reaching the first atomic computation. At that point, the atomic computation is run, and the various resulting instances of *<cont>* are passed on to the continuation. Before the first "piece" of the continuation is run, the current continuation field is reset not to include that piece.

Given these definitions, Scheme's *call-with-current-continuation* (or *call/cc*) can be defined as an ordinary function, using *reify* and *reflect*. We use *reify* to capture the continuation in the context where *call/cc* is invoked. Then, inside the function passed to the receiver function, we use *reflect* to replace the current continuation with the saved one.

$$
\begin{array}{l}
\textit{call/cc} =_{\text{def}} \lambda f.\textbf{let}\ x = \textit{reify}(\textit{<cont>}) \\
\qquad\qquad\qquad \textbf{in if}\ \textit{instance?}(x, \textit{List}) \\
\qquad\qquad\qquad\quad \textbf{then}\ \textit{head}(x) \\
\qquad\qquad\qquad\quad \textbf{else}\ f(\lambda v.\textbf{let}\ \textit{new} = \textit{reify}(\textit{<cont>}) \\
\qquad\qquad\qquad\qquad\qquad \textbf{in}\ \textit{reflect}(\textit{<cont>}, \textit{new}[x.cc/cc], \textit{list}(v)))
\end{array}
$$

Some cleverness is required to return the value of v to the right context; we use $\textit{list}(v)$ to guarantee that the type of x is different from *<cont>* the second time x is bound. It might seem that we could simply replace the entire reflective object *new* with the saved one (i.e., *old*), rather than just updating it with the old continuation, but that would throw away any additional information that the current instance of *<cont>* might be carrying. Remember that the object in question might actually be an indirect instance of *<cont>*, i.e., a direct instance of some subreflective of *<cont>*.

4 An Implementation

Our implementation is translation-based. In essence, the translation lays out a program so that it *syntactically* resembles the computation/subcomputation model used by the reflectives and their methods. The next section describes the main portion of the translation; section 4.2 discusses the representation and propagation of meta-information; and section 4.3 extends the translation to support the definition of new reflectives, as well as *reify* and *reflect*.

4.1 Program Translation

The goal of the translation is to recognize and demarcate each unit of syntax that corresponds semantically to a computation. After being translated, a program should have runtime "hooks" into each atomic and compound computation. Furthermore, in each syntactic construct that evaluates as a compound computation, the subexpression that evaluates as the initial subcomputation should be clearly set apart. Given these requirements, it is not surprising that the resulting translation bears a strong resemblance to continuation-passing style [15], A-normal form [6], and monadic style [13, 19]. (See section 5.)

The translation $\mathcal{R}$ given below places references to two functions in its output: α is applied to the program fragments that correspond to atomic computations, and γ is applied to the program fragments that correspond to compound computations. The first argument to γ corresponds to the initial subcomputation, and the second argument corresponds to the rest of the subcomputations, packaged as a function that expects to be applied to the *base-value* generated by the initial subcomputation. To make the rest of the translation more readable, we begin by defining a helpful piece of "syntactic sugar":

$$\textbf{bind}\ v = E_v\ \textbf{in}\ E \quad \Rightarrow \quad \gamma(E_v, \lambda v.E)$$

We use **bind** instead of γ in the rest of the translation for two reasons: it is cumbersome to read the nested applications and λ-expressions that would otherwise appear, and the new syntax makes it clearer which subexpression corresponds to the initial subcomputation.

$$\begin{array}{lcl}
\mathcal{R}[c] & \Rightarrow & \alpha(c) \\
\mathcal{R}[x] & \Rightarrow & \alpha(x) \\
\mathcal{R}[\lambda x.E] & \Rightarrow & \alpha(\lambda x.\mathcal{R}[E]) \\
\mathcal{R}[\textbf{let}\ x = E_1\ \textbf{in}\ E_2] & \Rightarrow & \mathcal{R}[(\lambda x.E_2)(E_1)] \\
\mathcal{R}[\textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ E_3] & \Rightarrow & \textbf{bind}\ t = \mathcal{R}[E_1] \\
 & & \textbf{in}\ \textbf{if}\ t\ \textbf{then}\ \mathcal{R}[E_2]\ \textbf{else}\ \mathcal{R}[E_3] \\
\mathcal{R}[E_0(E_1, \ldots, E_k)] & \Rightarrow & \textbf{bind}\ e_0 = \mathcal{R}[E_0] \\
 & & \textbf{in}\ \ \textbf{bind}\ e_1 = \mathcal{R}[E_1] \\
 & & \quad\ \ \textbf{in} \\
 & & \qquad\quad \ddots \\
 & & \qquad\qquad \textbf{bind}\ e_k = \mathcal{R}[E_k] \\
 & & \qquad\qquad \textbf{in}\ e_0(e_1, \ldots, e_k) \\
\mathcal{R}[E_0[E_1/f]] & \Rightarrow & \textbf{bind}\ e_0 = \mathcal{R}[E_0] \\
 & & \textbf{in}\ \ \textbf{bind}\ e_1 = \mathcal{R}[E_1] \\
 & & \quad\ \ \textbf{in}\ \alpha(e_0[e_1/f]) \\
\mathcal{R}[E.f] & \Rightarrow & \textbf{bind}\ e = \mathcal{R}[E] \\
 & & \textbf{in}\ \alpha(e.f)
\end{array}$$

The translations for *reify* and *reflect* are absent because they depend on knowledge of the representation of meta-information to be able to shift reflective objects between the meta-level and the value domain.

4.2 Representation

Our goal now is to implement α and γ based on some particular representation of meta-information. Let us begin by restricting our interest to simple reflectives and returning to the idea of computations as transformers of meta-information. The translation above treats atomic computations as values, evidenced by the fact that the translation does not go "inside" those expressions that correspond to atomic computations. Suppose, for the moment, that there is only one reflective defined. To ensure that some sort of values are available to be returned by computations, let that one reflective be *<value-reflective>*.

We know that $\alpha(v)$—where v is a value produced by an atomic computation—should be able to accept a reflective object (an instance of *<value-reflective>*) and transform it according to the definition of *atomic*. Thus, we conclude that the type T of $\alpha(v)$ is

$$T = S \rightarrow S$$

where S is the type of simple reflectives. The type of γ is induced by the type of α, since γ

is only applied to translated code. In fact, the translation guarantees that

$$\begin{aligned}\alpha &: \mathbf{v} \to T \\ \gamma &: T \to (\mathbf{v} \to T) \to T\end{aligned}$$

always holds as we vary T, where $\mathbf{v}$ is the type of values. Roughly following the diagrams in section 3.4, we define α and γ as follows:

$$\begin{aligned}&\alpha(v) =_{\text{def}} \lambda s.\mathit{atomic}(v, s) \\ &\gamma(i, r) =_{\text{def}} \lambda s.\textbf{let } s_2 = i(\mathit{compound\text{-}initial}(s)) \\ &\qquad\qquad \textbf{in let } s_3 = \mathit{compound\text{-}rest}(s, s_2) \\ &\qquad\qquad\quad \textbf{in } r(s_3.\mathit{base\text{-}value})(s_3)\end{aligned}$$

These definitions only support a single simple (value) reflective, though. In order to handle many simple reflectives simultaneously, we must complicate the definitions a little.

Still restricting our world so that there are only simple reflectives and no sibling value reflectives (where one is not a subtype of the other), suppose we have a function *find-value-obj* that picks the value reflective object out of a set of simple reflective objects. Then we can support a system in which

$$T = 2^S \to 2^S$$

by extending α and γ as follows:

$$\begin{aligned}&\alpha(v) =_{\text{def}} \lambda l.\mathit{map}(\lambda s.\mathit{atomic}(v, s), l) \\ &\gamma(i, r) =_{\text{def}} \lambda l.\textbf{let } l_2 = i(\mathit{map}(\mathit{compound\text{-}initial}, l)) \\ &\qquad\qquad \textbf{in let } l_3 = \mathit{map}(\mathit{compound\text{-}rest}, l, l_2) \\ &\qquad\qquad\quad \textbf{in } r(\mathit{find\text{-}value\text{-}obj}(l_3).\mathit{base\text{-}value})(l_3)\end{aligned}$$

We still have not dealt with control reflectives in any of these definitions. In the end, we would like to support multiple threads of control, each with its own idea of what the current meta-information is. To accomplish this objective, we define control reflectives so that each instance carries around a complete set of simple reflective objects (possibly including several value reflective objects).

$$\mathit{<control\text{-}reflective>} =_{\text{def}} \mathit{<reflective>} \uplus \{\mathit{objs}\}$$

Before we leap into full support of multiple threads, let us relax our restrictions only slightly, so that there can be one control reflective in addition to the one value reflective and arbitrarily many other simple reflectives. We fix that control reflective as the base one: *<control-reflective>*. Then we upgrade our system so that

$$T = C \to C$$

where C is the type of control reflectives. The new definitions of α and γ are very similar to the previous ones, except that the list of simple reflectives is now carried in the *objs* field of a control reflective. Also, *find-value-obj* has been extended to get the value reflective object out of a control reflective object.

$$
\begin{aligned}
\alpha(v) &=_{\text{def}} \lambda c.c[\mathit{map}(\lambda s.\mathit{atomic}(v,s), c.\mathit{objs})/\mathit{objs}] \\
\gamma(i,r) &=_{\text{def}} \lambda c.\textbf{let } c_2 = i(c[\mathit{map}(\mathit{compound\text{-}initial}, c.\mathit{objs})/\mathit{objs}]) \\
&\qquad \textbf{in let } c_3 = c_2[\mathit{map}(\mathit{compound\text{-}rest}, c.\mathit{objs}, c_2.\mathit{objs})/\mathit{objs}] \\
&\qquad\quad \textbf{in } r(\mathit{find\text{-}value\text{-}obj}(c_3).\mathit{base\text{-}value})(c_3)
\end{aligned}
$$

Finally, we are ready to support multiple threads of control. In this model, multiple value reflectives could cause a process to branch in different directions, leading to differing sets of reflective objects. We only leave one restriction in place now: no user-defined control reflectives. Even with this restriction, we have reached our final definition of T:

$$T = C \to 2^C$$

In the following definitions of α and γ, we must use *map/concat* in two places to bring the type down from 2^{2^C} to 2^C. We also replace *find-value-obj* with the more general *value-objs*, which collects *all* the value reflective objects from among the set of objects carried by the control reflective object.

$$
\begin{aligned}
\alpha(v) &=_{\text{def}} \lambda c.\mathit{list}(c[\mathit{map}(\lambda s.\mathit{atomic}(v,s), c.\mathit{objs})/\mathit{objs}]) \\
\gamma(i,r) &=_{\text{def}} \lambda c.\mathit{map/concat}(\lambda c_2.\textbf{let } c_3 = c_2[\mathit{map}(\mathit{compound\text{-}rest}, c.\mathit{objs}, c_2.\mathit{objs})/\mathit{objs}] \\
&\qquad\qquad \textbf{in } \mathit{map/concat}(\lambda v.r(v.\mathit{base\text{-}value})(c_3), \mathit{value\text{-}objs}(c_3)), \\
&\qquad\quad i(c[\mathit{map}(\mathit{compound\text{-}initial}, c.\mathit{objs})/\mathit{objs}]))
\end{aligned}
$$

In the final revision of α and γ, we allow for user-defined control reflectives. Remember that we must leave it up to *atomic-control* and *compound-control* to decide when to perform the computations, so we wrap most of the activity of α and γ inside λ-expressions, which are passed to the control functions.

$$
\begin{aligned}
\alpha(v) &=_{\text{def}} \lambda c.\mathit{atomic\text{-}control}(c, \lambda c_1.\mathit{list}(c_1[\mathit{map}(\lambda s.\mathit{atomic}(v,s), c_1.\mathit{objs})/\mathit{objs}])) \\
\gamma(i,r) &=_{\text{def}} \lambda c.\mathit{compound\text{-}control}(c, \\
&\qquad \lambda c_1.i(c_1[\mathit{map}(\mathit{compound\text{-}initial}, c_1.\mathit{objs})/\mathit{objs}]), \\
&\qquad \lambda c_2.\textbf{let } c_3 = c_2[\mathit{map}(\mathit{compound\text{-}rest}, c.\mathit{objs}, c_2.\mathit{objs})/\mathit{objs}] \\
&\qquad\quad \textbf{in } \mathit{map/concat}(\lambda v.r(v.\mathit{base\text{-}value})(c_3), \mathit{value\text{-}objs}(c_3)))
\end{aligned}
$$

Looking back at the default definitions of *atomic-control* and *compound-control* in section 3.6, we can see that we have merely *uninstantiated* those definitions in order to produce the final versions of α and γ.

One issue we have avoided so far is precisely how to choose the necessary set of reflectives for completely representing the user's choice of meta-information. It turns out that it is *not* necessary to instantiate all of the reflectives that are defined. Since any new reflective inherits the fields of its parent, and since the definitions of the methods specialized to the new reflective are intended to supersede those of the parent, it is sufficient to use only the *leaves* of the inheritance hierarchy. For example, once *<bounded-int>* has been defined, there is no longer any need to carry around an instance of *<value-reflective>* to represent expressed values.

4.3 Interface

Now we are ready to return to the translation and extend it to support user-defined reflectives, *reify*, and *reflect*. The obvious and most convenient way to implement reflectives is as classes. Reflective objects are then merely instances of those classes. Thus, the translation of definitions of reflectives is an empty translation, if the notation for class definitions is the same as that of reflective definitions.

In order to implement *reify* and *reflect*, we introduce two new functions: *reflective-ref* and *reflective-subst*. The *reflective-ref* function takes a control reflective object c and a reflective r and finds an instance of r in c. The *reflective-subst* function takes some reflective object o, a reflective r, and a control reflective object c and returns a control reflective object c' that is like c, except that o has been substituted for the instance of r in c. (The reflective object o should be of type r.)

```
$\mathit{reflective\text{-}ref}(c{::}{<}\mathit{control\text{-}reflective}{>}, r) =_{\mathrm{def}}$
    if $\mathit{instance?}(c, r)$
    then $c$
    else $\mathit{reflective\text{-}ref}(c.\mathit{objs}, r)$
$\mathit{reflective\text{-}ref}(\mathit{refls}{::}\mathit{List}, r) =_{\mathrm{def}}$
    if $\mathit{instance?}(\mathit{head}(\mathit{refls}), r)$
    then $\mathit{head}(\mathit{refls})$
    else $\mathit{reflective\text{-}ref}(\mathit{tail}(\mathit{refls}), r)$
$\mathit{reflective\text{-}subst}(o, r, c{::}{<}\mathit{control\text{-}reflective}{>}) =_{\mathrm{def}}$
    if $\mathit{instance?}(c, r)$
    then $o[c.\mathit{objs}/\mathit{objs}]$
    else $c[\mathit{reflective\text{-}subst}(o, r, c.\mathit{objs})/\mathit{objs}]$
$\mathit{reflective\text{-}subst}(o, r, \mathit{refls}{::}\mathit{List}) =_{\mathrm{def}}$
    if $\mathit{instance?}(\mathit{head}(\mathit{refls}), r)$
    then $\mathit{pair}(o, \mathit{tail}(\mathit{refls}))$
    else $\mathit{pair}(\mathit{head}(\mathit{refls}), \mathit{reflective\text{-}subst}(o, r, \mathit{tail}(\mathit{refls})))$
```

(The definitions above have been simplified a great deal. More robust definitions have to check for error cases and handle the appearance of more than one satisfactory target object in the list of reflective objects.) The translations of *reify* and *reflect* simply use these functions to do their work.

$$\begin{array}{lll}
\mathcal{R}[\mathit{reify}(E_r)] & \Rightarrow & \textbf{bind}\ r = \mathcal{R}[E_r] \\
 & & \textbf{in}\ \lambda c.\alpha(\mathit{reflective\text{-}ref}(c, r))(c) \\
\mathcal{R}[\mathit{reflect}(E_r, E_o, E)] & \Rightarrow & \textbf{bind}\ r = \mathcal{R}[E_r] \\
 & & \textbf{in}\ \ \textbf{bind}\ o = \mathcal{R}[E_o] \\
 & & \quad\ \textbf{in}\ \lambda c.(\mathcal{R}[E])(\mathit{reflective\text{-}subst}(o, r, c))
\end{array}$$

Each of these translations takes advantage of the knowledge that α and γ are both functions expecting to be applied to instances of control reflectives.

5 Conclusion

This paper has introduced reflection-oriented programming, both as a programming paradigm and as a system for conveniently directing the propagation of meta-level information through a process. The main contributions of this work have been to lay out a concise framework for computational reflection, to provide examples of its use, and to demonstrate a translation-based implementation. Using such an implementation enables reflection in compiled programs, even when some parts of the programs are compiled *before* it is known what meta-level concepts are to be modeled. Throughout the paper, a distinction has been made between reflection about programs and reflection about computations.

This work leaves several issues unaddressed and open to further research. The most outstanding of these is a more precise formalization of reflection-oriented programming, so that it may be more carefully compared to monadic reflection and other formal accounts of reflection. It is clear that under some restrictions, the functions α and γ, along with a variant of T, form a Kleisli triple [14]. Those restrictions should be made more precise, and the properties of reflectives that violate the restrictions should be investigated carefully.

The power of the combination of value reflectives and control reflectives has not been fully explored. It should be possible to have a program perform a significant amount of "static" analysis on itself reflectively, where staticness is enforced by the manipulation of control reflectives. One extension of this idea is to create dynamically self-optimizing programs.

Multiple inheritance and local (fluid) introduction of reflectives are not addressed in this paper, although we expect many of the discussions about these topics in object-oriented programming to carry over cleanly into the realm of reflection-oriented programming. Supporting multiple inheritance would enable the creation of libraries of reflectives from which a user could inherit in order to mix together several reflective properties. Supporting the fluid introduction of reflectives would allow such things as the measurement of computational expense for some segment of a computation without defining a new reflective globally.

Our implementation is somewhat naïve in its uniform translation of all programs, regardless of the particular reflectives defined in them. In order to make reflection usable and efficient, a great deal of attention needs to be focused on improving and optimizing our implementation. For example, in a closed system, where it can be assumed that no new reflectives can be introduced, it should be possible to eliminate many calls to α and γ. (One approach would be to follow the methodology used by Filinski [5] for writing direct-style programs that use monads.) A related issue is the partial restriction of the extensibility of reflection-oriented programs. Sometimes, it might be desirable to leave a program only partially open to reflection, especially if total openness would create security risks.

Reflection-oriented programming may have significant ramifications for characterizing programming language semantics more concisely and more modularly. In particular, presentations of interpreters and compilers could be greatly simplified by leaning more heavily on reflection, especially when combined with well-crafted use of object-orientation.

Acknowledgments

We wish to thank Brian Smith for helpful comments on the nature of reflection and Erik Hilsdale and Jon Rossie for reading and critiquing our work. We also wish to thank anonymous referees whose critiques led to a revision of our notation and presentation strategies.

References

[1] Alan Bawden. Reification without evaluation. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 342–351, Snowbird, Utah, July 1988. ACM Press.

[2] Patrick Cousot and Rhadia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[3] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 327–341, Snowbird, Utah, July 1988. ACM Press.

[4] David A. Espinosa. *Semantic Lego.* PhD thesis, Columbia University, New York, 1995.

[5] Andrzej Filinski. Representing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, New York, January 1994. ACM Press.

[6] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247. ACM Press, 1993.

[7] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 348–355, Austin, Texas, August 1984. ACM Press.

[8] Stanley Jefferson and Daniel P. Friedman. A simple reflective interpreter. *Lisp and Symbolic Computation*, 9(2/3):181–202, May/June 1996.

[9] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[10] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, January 1995. ACM Press.

[11] Pattie Maes. Concepts and experiments in computational reflection. *Proceedings of OOPSLA '87, ACM SIGPLAN Notices*, 22(12):147–155, December 1987.

[12] Anurag Mendhekar and Daniel P. Friedman. An exploration of relationships between reflective theories. In *Proceedings of Reflection '96*, San Francisco, April 1996. To appear.

[13] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, April 1989.

[14] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[15] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[16] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[17] Brian C. Smith. Reflection and semantics in a procedural language. Technical Report MIT-LCS-TR-272, Massachusetts Institute of Technology, Cambridge, Mass., January 1982.

[18] Brian C. Smith. Reflection and semantics in LISP. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM Press, January 1984.

[19] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, December 1992.

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