An Exploration of Relationships between Reflective Theories

Anurag Mendhekar
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, CA 94304,U.S.A

Daniel P. Friedman*
Indiana University
Bloomington, IN 47405, U.S.A

Abstract

There have been individual notions of reflection in different branches of logic and computer science. Apart from an intuitive connection between these notions about a "meta" level, there is little understanding of relationships between them. In this paper, we present a preliminary report of an exploration of a deeper understanding of these relationships. The three notions we are going to explore are: logical, computational and monadic. Based on the logical notion, we setup a framework of desirable properties with which to evaluate the other forms of reflection. Computational reflection does not satisfy some of these properties, while monadic reflection does. Computational reflection is, however, of greater utility, since it allows finer control over program implementation. Based on some of this understanding, we show how this utility of computational reflection can be achieved in the framework of monadic reflection by separating base-code from meta-code, and restricting them in certain ways. This allows us to use the reasoning mechanisms provided by the call-by-value λ-calculus for reflective programming.

1 Introduction

There are notions related to reflection in many fields of logic and computer science. The intuition behind all these notions of reflection is the broad

*Research partially supported by NSF grant CCR-9302114

notion of access to a "meta-level". For example, the proof of Gödel's Incompleteness theorem vitally depends on the notion of "truth" predicates, which provide access at the object language level, to the deduction power of the meta-theory. Logicians have been attempting to make sense of "truth" and its meaning for a long time. There is a large body of work that deals with the study of truth. For the sake of uniformity in this paper, let us call this field of study *logical reflection.*

Computational reflection has now become an important tool in language design. In this form of reflection, access is provided to the implementation structure of a system in a principled and uniform manner. This has proved to be important in improving the quality of software—for better maintainability and performance.

More recently, and from a theoretical perspective, *monads* [9] have been proposed as a way of organizing the meta-level so that the semantics of diverse "computational effects" (such as side-effects and continuations) can be expressed in a uniform manner by appropriately adjusting a very small part of the meta-level organization. This form of reflection is known as *monad reflection.*

In this paper we study the relationship between these forms of reflection in a mathematical sense. Our motivation for studying this is simply that we seek a theoretical understanding of computational reflection. As reflection becomes more widely used, it is increasingly important that any language that incorporates reflection must be designed so that programs written in that language are easy to read, write and debug. This requires a reasoning mechanism, either formal or informal.

While the other two fields mentioned above have had deep logico-mathematical grounding, computational reflection is as yet poorly understood, despite much early work in the semantics of reflection [2, 17]. We hope to build a mathematical understanding of computational reflection by exploiting its connections with other forms of reflection. This paper is a preliminary report on this work.

We begin with logical reflection, and how truth predicates are understood. Using constructive principles, we relate truth predicates in logic to a language with reflective operators. Based on this analysis, we establish certain criteria that will help in reasoning about reflection. Using these criteria, we evaluate other notions of reflection in programming languages: static reflection, monadic reflection and computational reflection. We show that monadic reflection has many desirable properties, but in many cases, lacks the utility of computational reflection. Based on some of this understand-

ing, we show how this utility of computational reflection can be incorporated into monadic reflection.

2 Reflection in Logic: Truth Predicates

Truth predicates are well known. These are predicates introduced in a theory to reason about propositions in the theory itself. Truth predicates are defined over representations of propositions and they hold for all those representations for which the corresponding propositions are true. Tarski [16] was the first to formalize this definition of truth under his famous "Convention T", also known as the Tarski biconditionals:

$$\textit{T-introduction}\ \frac{A}{T(\ulcorner A \urcorner)} \qquad \textit{T-elimination}\ \frac{T(\ulcorner A \urcorner)}{A}$$

For simplicity, we assume a positive (no negation), implicational logic with a single predicate T. *T-introduction* incorporates the fact that if A holds, then the predicate T holds for the name of A. *T-elimination* affirms that the reverse also holds: if T holds for the name of some sentence A, then A holds.

While extremely simple, the T-biconditionals incorporate our basic intuitions about a meta-level: T is *about* the base language, and its behavior reflects that of the theory.

3 The Constructive Approach

One of the standard ways of relating logical concepts with computational ones is through the use of the *Curry-Howard isomorphism* [5]. This isomorphism allows formulae in a logic to be interpreted as types in a suitable computational domain. Proofs of the propositions correspond to programs of that type. We define reductions on proofs so that execution of the program yields a result that is considered to be a *witness* of the proposition being proved.

This construction for intuitionistic logic is well understood, and we repeat it here only briefly. We take typed λ-calculus terms as proofs and use standard reductions to extract witnesses. Examples of term construction are: $\wedge$-introduction corresponds to pairing, $\wedge$-elimination to projecting out components of a pair; $\vee$-introduction to disjoint unions, $\vee$-elimination to

dispatch based on the union; $\rightarrow$-introduction to procedures and *modus ponens* to application and so on. The rules of the logic can be considered to be typing rules of the corresponding operators.

How can we interpret truth predicates this way? We must first introduce operators into the λ-calculus that correspond to truth predicates. Let us define operators $\Uparrow$ (reify) and $\Downarrow$ (reflect) to correspond to (i.e., with typing rules) T-introduction and T-elimination. Let us call these operators *reflective* operators. These operators must be given a computational interpretation consistent with the constructive interpretation of truth predicates. What is the constructive interpretation of truth predicates? Let us look at T-introduction. If we are given a proof p for a sentence A, then $\Uparrow p$ must give a proof of the assertion that a proof for A exists. We can simply take this proof to be p. The proof of $T(\ulcorner A \urcorner)$ is witnessed by some representation of p. This is what $\Uparrow p$ must evaluate to. T-elimination can be interpreted similarly. Given a proof p of the existence of a proof of A, $\Downarrow p$ must give rise to a witness for A.

Let us take an example. It is often attractive to optimize functions at run time based on information available when all the data to a program is available. No widely used programming language actually supports this. What we would like to write is a function f that takes in the representation of a function g and returns a new representation of g that is optimized based on run-time information. The new g can now be incorporated in the program. Suppose that we have written f. How can we think about a program that uses f? Let us consider a function R, say of type $A \rightarrow A$. A run-time optimization of this can be written as follows: $\Downarrow(f(\Uparrow R))$. We take a representation of R using $\Uparrow$, process it through f, and incorporate the optimized function using $\Downarrow$. Let us try to look at this program as a proof. The way to read the following is that the proposition to the right of the ':' is a formula and the term to its left is a proof of that formula. The horizontal lines represent application of inference rules, and the rule used is written to the right of the line.

$$\frac{\dfrac{R : A \rightarrow A}{\Uparrow R : T(A \rightarrow A)}\ (T\text{-}intro) \qquad f : T(A \rightarrow A) \rightarrow T(A \rightarrow A)}{\dfrac{f(\Uparrow R) : T(A \rightarrow A)}{\Downarrow(f(\Uparrow R)) : A \rightarrow A}\ (T\text{-}elim)}\ (\rightarrow\text{-}elim)$$

This also can be read in another way: as an inference of the type of $\Downarrow(f(\Uparrow R))$. The formula to the right of the ':' is the type of the term to the left of ':'. The horizontal lines are applications of type-inference rules.

4 Equivalence of Proofs

Since we are interpreting proofs as programs, our ability to reason about these programs is determined by what programs we consider to be equivalent to one another. For λ-calculus terms, this equality is given by the calculus itself. With the introduction of reflective operators, our calculus must be augmented with rules about how programs containing reflective operators behave. In this section, we state some desirable properties of these operators, without actually giving a real operational semantics to these operators. In later sections, we instantiate the semantics of these operators with existing ones to evaluate them in the context of these properties.

Our first observation is that the rules T-introduction and T-elimination are symmetric. This suggests that the operators $\Uparrow$ and $\Downarrow$ ought to be inverses of each other. Indeed, going back and forth between a proof and its representation does not change the proof itself, or its representation. We propose the following equalities. If p proves A, $p = \Uparrow\Downarrow p$ and $p = \Downarrow\Uparrow p$. Let us call these properties *inverse-1* and *inverse-2*, respectively.

Our second observation is that if we decide that representations of two proofs are equal, the proofs must be equal. This gives rise to the following rule: If M and N are equal and give rise to a representation of p, $\Downarrow M = \Downarrow N$. Let us call this rule the *reflection property*. Similarly, we would like to have: If M and N are equal and give rise to a representation of p, $\Uparrow M = \Uparrow N$. Let us call this rule the *reification property*.

4.1 Discussion

Why are these properties desirable? If we are to think about other forms of reflection in terms similar to the well-understood truth predicate, then it is desirable that the observations we glean from truth predicates be guides for evaluating the others. Access to meta-level is the fundamental commonality in the forms of reflection that we have talked about above. Truth predicates incorporate this commonality without committing to specific operational semantics. This leads us to believe that reflective properties of truth predicates are properties about a fundamental intuition about reflection that we have. As we see later, this belief is borne out by the fact that those languages that violate even one of these properties have more complicated theories (i.e., they are, in some sense, harder to think about).

The four properties elucidated above incorporate certain basic notions of how we want to think about reflective operators. The inverse properties

assert that a term and its representation, while differentiated, correspond intimately with each other. This is important because when programming reflectively, we think of manipulating representations. The awareness that this representation corresponds to the actual term is important in helping us reason about programs.

The other two properties are necessary for referential transparency to hold, which is, in general, a useful property for reasoning mechanisms to have.

5 Reflection in Programming Languages

We have now set up a general framework in which to evaluate various notions of reflection in programming languages. We have deliberately stayed away from concrete operational semantics and mainly set up some general principles that we think are important for reasoning about reflective programs. We have been talking about "representations" in very abstract terms, but have made no commitments to what this representation should really look like.

In the next few sections, we instantiate our abstract concepts with very concrete ones in order to examine how they stand with respect to this framework.

5.1 Static Reflection: `quote` and `eval`

Let us start with the simple operators `quote` and `eval` present in a language such as Lisp. Lists are used as representations of programs and `quote` is a mechanism for reifying programs. An added restriction is that *quote* is not affected by variable bindings. i.e., `(lambda (x) '(x))` will always return the list `(x)`, regardless of what argument is actually passed to the procedure. `Eval` takes a quoted program, evaluates it, and returns the result.

Let us see how these operators behave with respect to our rules. For Inverse-1, `'(eval` *exp*) is clearly not equal to *exp*. This is because we use a list-representation for programs. For Inverse-2, `(eval '`*exp*)=*exp* holds, provided *exp* is a closed term. It is easy to see that the Reflection rule also holds for this operational semantics. The reification rule, however, fails. This can be seen by taking two equal terms: $((\lambda x.x)3)$ and 3, but $'((\lambda x.x)3)$ is not equal to $'3$.

Muller [10] develops a theory for these operators and demonstrates that adding these operators affects equational reasoning–some things that we

think ought to be true turn out not to be so. Clearly, from a reasoning point of view, this operational semantics is unsatisfactory.

5.2 Monad reflection

Let us now examine another kind of reflective operational semantics, based on monads. Monads are a category theoretic concept that can be used to express a wide variety of computational effects. We do not get into the category theoretic details of this, but instead refer the reader to Moggi [9].

$\Uparrow$ and $\Downarrow$ are defined in terms of two operators: η and $\_^*$ called inclusion and extension respectively, and these operators must satisfy the following properties:

$$\begin{aligned} \eta^* &= id \\ f^* \circ \eta &= f \\ (f^* \circ g)^* &= (f^* \circ g^*) \end{aligned}$$

Intuitively, η injects a value into a representation of the value and $\_^*$ transforms a function that takes a value into a function that takes the representation of that value. By defining η and $\_^*$ suitably in the base language, the user can control representations of the value.

Moggi gives a semantics for the operators $\Uparrow$ and $\Downarrow$ through a translation using η and $\_^*$. Filinski [4] shows that this can be done instead by using control operators `shift` and `reset`. While a theoretical definition of these operators is rather involved, it is easy to explain them in informal terms, using the concept of execution stacks. `reset` places a marker on the execution stack. The operator `shift` takes a procedure as an argument. It takes the continuation up to (but not including) the marker closest to the top of the execution stack, converts it to a procedure and passes it to its argument. The continuation is non-aborting. `shift`, however, is aborting and pops the execution stack up to (and including) the marker placed on the stack.
Filinski defines the operators $\Uparrow$ and $\Downarrow$ as follows:

$$\Uparrow exp =_{\text{def}} (\texttt{reset}(\eta\ exp))$$

$$\Downarrow exp =_{\text{def}} (\texttt{shift}(\lambda k.(k^* exp)))$$

Let us now examine what properties these operators satisfy.

1. Inverse-1:

$$\begin{aligned} \Uparrow\Downarrow M &= (\texttt{reset}(\eta(\texttt{shift}(\lambda k.(k^* M))))) \\ &= (\eta^* M) \\ &= M \end{aligned}$$

2. Inverse-2: This property is satisfied. The proof is tedious but straightforward, and we omit it here for brevity. Intuitively, however, this can be seen simply by observing that reifying the monad and reflecting it without modification is an identity operation. The proof depends upon the fact that `shift` and `reset` are being used in a restricted manner.

3. Reflection and reification Properties: It is easy to see that these properties are also satisfied by monadic reflection. η and $\_^*$ are defined so that if $M = N$, then $\eta M = \eta N$ and if $M = N$ then $M^* = N^*$. Moreover, `shift` and `reset` also behave similarly.

We see that from a perspective of reasoning mechanisms, monadic reflection has desirable properties.

5.3 Computational Reflection

In the original conception of Reflection [14], the idea is to make available operators that provide access to the complete state of the computation. A theory for this is presented by Mendhekar and Friedman [8]. We present a brief overview of this theory.

In this theory, it is assumed that the implementation is a rewrite system and maintains two pieces of state: The term it is rewriting and the sub-term that is the next redex. Reification returns some representation of these two pieces of state. Reflection takes such a representation and installs it as the state of the rewrite system. The representation is *chosen* so that it differentiates syntactically between terms. It is, in fact, a desirable property that reflective programming differentiates between two seemingly equivalent programs. This is what provides programmers with control over how their programs get implemented. If we only have representations that are based on equivalence classes of terms, reflective properties will be mostly useless.

A consequence of this representation decision is that the theory is not a conservative extension of the lambda calculus—things that we think ought to be true turn out not to be so. This follows from the fact that the reification property is not satisfied. Many obvious equivalences between programs are not provable in this theory. The other three properties are satisfied by this theory.

6 Violating the Reification Property

Reflective programming derives its appeal from allowing programmers to manipulate representations in order to achieve gains in performance and maintainability. One implication of this is that there is an ontological assumption that representations can be changed without changing the meaning of the program. This ontological assumption is in direct contradiction with the reification property.

In this sense, the violation of the reification property is vital to reflection in programming languages being useful.[1] On the other hand, this violation leads to a loss of reasoning power. How can we strike a balance between these two? How can we make reflection in programming languages easy to reason about *and* useful? We examine the answer to this question in the rest of this section.

Our first observation is that reflection is mostly used in a principled way: We usually define abstractions that use reflective operators and these abstractions are used everywhere within the program. This is referred to as *separation of concerns*. If we assume this to be the norm, then we would like that our reasoning about the program be supported in a similar manner. We would like to separately reason about reflective and non-reflective parts of the program and be assured that interaction between them is unproblematic.

As a trivial example, consider the case where we have a lambda-calculus based language with reflective operators, but in all our programs, we never use a reflective operator. We can surely use the λ-calculus to reason about this program. Another example is a restricted use of reflective capabilities when we only look at the representation procedures to determine how many variables they need. This might be necessary, for example, when we want to use reflection to build tracing into our procedures. A way to do it would be to take the procedure P in question, determine how many arguments it takes, and construct a wrapper procedure that prints these arguments before passing them on to P. This is a very limited use of reflection. Moreover, only part of the representation information is being used and so we can use (informal) λ-calculus reasoning mechanisms to reason about programs that use tracing procedures (ignoring the fact that tracing causes characters to be printed on the output stream).

Our second observation is that the framework of monadic reflection has

[1] This is not to say that monad reflection is not useful. That would depend on the nature of the application. Its applicability, however, is limited where explicit manipulations of representation are required, and we believe this to be the general case.

the properties we desire. Can it be augmented so that it still maintains its properties but allows a higher degree of reflection?

7 Reflective Monads

We examine a solution to the question raised above in this section. We use separation of concerns and partition a program into its reflective and non-reflective components. By "reflective" we mean those parts that explicitly manipulate representations of values. We cast these restrictions in the framework of monads, at the same time relaxing some of the assumptions that are used when dealing with monads. We call the resulting structure a *reflective monad.* At the outset, we mention that while monads have categorical underpinnings, in this section we are not concerned with them.

Let us clarify the framework in which we develop reflective monads. In the following, we assume that the base calculus is the typed call-by-value λ-calculus, denoted by λ_v-calculus. Apart from assuming that every term is type-correct, we do not explicitly mention types. Representations of terms of type A have the type $T(\ulcorner A \urcorner)$. Furthermore, we differentiate between values and their representations. Moreover, $\lambda \vdash M = N$ does not imply that the representations of M and N are the same. As we saw before, this is a crucial ontological assumption and it permits us to make finer differentiations between terms. We denote the representation of a term M as $\ulcorner M \urcorner$. Furthermore, we postulate an operation $\downarrow$ such that $\downarrow \ulcorner M \urcorner = M$.

We introduce the concept of a *reflective* monad that is similar to that of the monad, but has the following restrictions. The code for $\_^*$ is written in a language without reflective operators. It is, however, allowed to manipulate and construct representations, through the use of the $\downarrow$ operator, and other representation destructuring functions. Furthermore, this is the only code which is allowed to do this. We refer to this as the meta-code of the program. All other code is referred to as the base code. The base code is not allowed to construct term representations, except by using $\Uparrow$. These restrictions are such that they can be enforced syntactically.

The separation of concerns we desire is achieved through these restrictions. By disallowing the base code from manipulating term representations, we have encapsulated all reflective representation manipulations into the code for $\_^*$.[2] This separation of concerns allows us to use separate reason-

[2] A usability question arises. We have only one $\_^*$ function for a program. How can it be used to incorporate different kinds of meta-control at different points in the program?

ing mechanisms for base-code and meta-code. An important consequence of this is that the reasoning mechanism for base-code can use all four of our desirable properties.

We must also address the question of the nature of representations. The actual representation we choose depends upon the reflective capabilities we want the language to have. Some languages might decide to provide a full representation of the term and its context (as the theory described in Section 5.3). Others might simply choose to provide a simple interface. An example is **call/cc** in Scheme, where manipulating the continuation explicitly is not allowed, but a procedure-like interface is available. The assumption we make in the following is that the full representation of a reified context is rarely of any use. It is more important to have detailed representations of what goes into a context.[3]

7.1 The Reflective Monad

A reflective monad is a monad where η and $\_^*$ have the following behavior. ηM returns $\ulcorner M \urcorner$. Also, η is not defined on open terms. $\_^*$ is responsible for extending a given function to take a representation of its arguments. Furthermore, η and $\_^*$ must satisfy the monad properties described above.

Reflective monads differ from regular monads in subtle but important ways. The most important difference is that the reification property does not hold a priori. This is not true in regular monads, where η is assumed to be a function that satisfies the axioms of the λ-calculus.

Another difference is that regular monads allow values of the monad type to be constructed explicitly (i.e., without invoking $\Uparrow$). In fact, they sometimes require it. For example, with the non-deterministic monad that uses a list representation, the only way to construct a non-singleton set of multiple values is to explicitly construct it as a list.

7.2 Preserving the Reflection Property

If the meta-code were free to manipulate representations in an arbitrary manner, the reflection property would breakdown. This is because the meta-code could become sensitive to spurious differences in representation. We

We can solve this by appropriately tagging reified entities so that $\_^*$ can dispatch based on this tag.

[3]This restriction is not crucial: we can build a similar theory if we choose not to have this restriction, but it simplifies the presentation.

have, however, the second reflective monad axiom that $f^{*\ulcorner}M\urcorner = (fM)$. Therefore, we have the following theorem: if $M = N$, $f^{*\ulcorner}M\urcorner = f^{*\ulcorner}N\urcorner$. The implication of this is that even if the meta-code makes distinctions between intensional aspects of terms, what it finally does must be faithful to what the terms mean. This might disallow many valid reflective programs, but it makes reasoning for a large number of reflective programs much easier.

7.3 An example

Let us reconsider the run-time optimization example again. Suppose that we bind this run-time optimized procedure to a variable and use it in our code. We can extend the example in Section 3 in the obvious way to do this:[4]

```
(let ((g (reflect (f (reify R)))))
  (g args ...))
```

This, however, violates our restrictions since we are now using f in base code.

In the reflective monad, we write this code a little differently:

```
Base code:

(let ((g (reflect (reify R))))
  (g args ...))
```

The code for $\_^*$ is as follows:

```
(define star
  (lambda (k x)
    (if (run-time-optimizable? x)
        (k (down-arrow (f x)))
        (k (down-arrow x)))))
```

Here we are assuming the existence of a function that recognizes run-time optimizable functions. This could be done, for instance, by checking for membership in a list of predeclared optimizable functions. Also note the use of `down-arrow` to convert from representations to terms.

[4] We switch to a Scheme-like notation here. This notation is easier to work with for larger examples. We have used textual names for the operators: `reflect` for $\Downarrow$, `reify` for $\Uparrow$, `star` for $\_^*$, `eta` for η and `down-arrow` for $\downarrow$.

The way this example works is that when `reflect` passes on the reified value to its context (in this case the `let` binding, denoted in the meta-code by `k`), it processes it through `star`. This is when the meta-code gets a chance to manipulate the reified value. In our case, this is when the run-time optimization happens. The monad laws, of course, guarantee that this reified value is not changed semantically—only representationally.

Let us examine whether `star` satisfies the monad laws. The interesting case is the second law: $(h^{\ulcorner}M^{\urcorner}) = (hM)$. We can see that this holds, because our optimizing function `f` is an optimizer and does not change `R` semantically.

As an example of reasoning using the inverse property, in the base code, we can use inverse-1 to conclude that this program is semantically equivalent to `(R args ...)`. This is because we know that `star` satisfies our restrictions.

8 Theoretical Results About Reflective Monads

We have seen how reflective monads are useful in separating out reflective and non-reflective parts of a program. In this section, we develop theoretical results about reflective monads. We are interested in showing that the four desirable properties hold within the framework of reflective monads. We do this by developing a λ-theory for it and by showing that this theory is a conservative extension of the call-by-value λ-calculus.

Both inverse-1 and inverse-2 hold because reflective monads obey monad laws. The reflection property also holds because of monad laws. The only property that does not hold a priori is the reification property. This section takes into account the restrictions placed on reflective programs as given above and forces a rule in the theory that would make the reification property hold for the theory. We show that this is sound by proving that if the restrictions above are satisfied for all reflective programs, the theory is consistent. It is further shown that the theory is a conservative extension of the call-by-value λ-calculus. The implication of this is that conventional reasoning mechanisms need not be discarded to incorporate reflective capabilities in a language.

Let us define the syntax for our (base) language. The syntax is basically that of the λ_v-calculus but also allows $(\Downarrow term)$ and $(\Uparrow term)$ as terms, where $\Downarrow$ and $\Uparrow$ are as defined for monadic reflection. This set of terms is denoted Λ_R.

Let us now turn our attention to the semantics, which we present as an equational theory. We assume that the base language is call-by-value. We use β_v to denote call by value β-reduction. The semantics is described by a translation into the λ_v-calculus extended with the monad functions. We equip λ_v-calculus with the axioms for η and $\_^*$. Let us call the resulting theory $\lambda_{\eta,\_^*}$. (The η here stands for the η monad operator, and not the extensionality axiom of the λ_v-calculus.)

The translation is as follows. We first translate all the $\Downarrow$ and $\Uparrow$ terms, using the translation given in section 5.2. We use the standard CPS-translation to get a term in $\lambda_{\eta,\_^*}$. This whole translation is denoted by $[.]_{\text{cps}}$. Let us define a theory λ_R such that if M and N are terms in Λ_R, we say that $\lambda_R \vdash M = N$, iff $\lambda_{\eta,\_^*} \vdash [M]_{\text{cps}} = [N]_{\text{cps}}$.

Let us illustrate this translation on the base-code in example 7.3. By the first translation we get:

```
(let ((g (shift (lambda (k) (star k (reset (eta R)))))))
  (g args ...))
```

CPS converting this, assuming that this is the whole program, we get:

```
(star-cps
   (lambda (v) (let ((g v)) (g args-cps ... top)))
   (eta R)
   top)
```

Here the suffix `-cps` indicates results of CPS conversion. `top` stands for the top-level continuation. We also assume that the procedure bound to `R` is CPS'ed.

8.1 Re-introducing the Reification Property

Let us now consider the following rule, called the representation rule: $M = N \rightarrow (\eta M) = (\eta N)$. Is this a valid rule to add to $\lambda_{\eta,\_^*}$? On the face of it, it certainly isn't. Look, however, at all the restrictions we have placed on the base code. It cannot manipulate representations. Furthermore, the only place where such manipulation is allowed (i.e., the meta-code) is required to produce results that ultimately are independent of specific representation decisions. There are no base-code contexts that distinguish (ηM) and (ηN), when $M = N$.

Therefore, adding the rule should cause no problems. We must, however, prove this. From now on we write $\lambda_{\eta,\_^*}$ to mean the theory that includes

the above rule. The first theorem we prove establishes a correspondence between $\lambda_{\eta,-^*}$ and λ_v. This theorem is used to prove other properties such as consistency and conservative extension.

We begin by defining an erasure function $\mathcal{E}[\![.]\!]$ that replaces all occurrences of ηM with $\mathcal{E}[\![M]\!]$ and k^*M with $(k\mathcal{E}[\![M]\!])$. These correspond to using an identity monad.

$$\begin{array}{rcl}
\mathcal{E}[\![x]\!] & = & x \\
\mathcal{E}[\![(\lambda x.M)]\!] & = & (\lambda x.\mathcal{E}[\![M]\!]) \\
\mathcal{E}[\![\eta]\!] & = & \lambda x.x \\
\mathcal{E}[\![(M^*N)]\!] & = & (\mathcal{E}[\![M]\!]\mathcal{E}[\![N]\!]) \\
\mathcal{E}[\![(MN)]\!] & = & (\mathcal{E}[\![M]\!]\mathcal{E}[\![N]\!])
\end{array}$$

An important point to note is that erasure commutes with the substitution operation. We call this the *substitution-erasure* lemma. The proof is a straightforward induction on the structure of terms, and we omit it here for brevity.

Lemma 1 $\lambda_{\eta,-^*} \vdash M = N \rightarrow \lambda_v \vdash \mathcal{E}[\![M]\!] = \mathcal{E}[\![N]\!]$.

Proof: By induction on the derivation of $\lambda_{\eta,-^*} \vdash M = N$.
Basis: $M = N$ axiomatically. We have the beta axiom, and the monad axioms. In all cases, we see that the consequent holds. We appeal to the substitution-erasure lemma for the beta axiom. The other interesting case is when we use the axiom $(f^*\ulcorner M\urcorner) = (fM)$. By erasure, the left hand side gives us $(\mathcal{E}[\![f]\!]\mathcal{E}[\![M]\!])$, which is the erasure of the right hand side.
Induction: The interesting case is when the derivation ends with an application of the representation rule: if $M = N$ then $\eta M = \eta N$. By erasure, $\mathcal{E}[\![\eta M]\!] = \mathcal{E}[\![M]\!]$ and $\mathcal{E}[\![\eta N]\!] = \mathcal{E}[\![N]\!]$. By the induction hypothesis, $\mathcal{E}[\![\eta M]\!] = \mathcal{E}[\![M]\!] = \mathcal{E}[\![N]\!] = \mathcal{E}[\![\eta N]\!]$.

Theorem 1 (Correspondence). $\lambda_R \vdash M = N \rightarrow \lambda_v \vdash \mathcal{E}[\![[M]_{\text{cps}}]\!] = \mathcal{E}[\![[N]_{\text{cps}}]\!]$.

This follows immediately from the definition of derivability in λ_R and the above lemma.

Theorem 2 *λ_R is consistent.*

Proof: The proof of consistency is by contradiction. Take I and K as terms. From the definition of $\mathcal{E}[\![.]\!]$, $\mathcal{E}[\![[I]_{\text{cps}}]\!] = [I]_{\text{cps}}$ and $\mathcal{E}[\![[K]_{\text{cps}}]\!] = [K]_{\text{cps}}$. If λ_R were inconsistent, $\lambda_R \vdash I = K$, and hence $\lambda_v \vdash [I]_{\text{cps}} = [K]_{\text{cps}}$, which is a contradiction. Therefore, λ_R is consistent.

Theorem 3 *λ_R is a conservative extension of λ_v.*

Proof: One direction is given by the correspondence theorem. The other direction follows from the fact that the rules and axioms of λ_R contain those of λ_v.

From the above theorems, we have established that the addition of the representation rule to the theory is not problematic logically. This also means that the reification property holds for reflective monads. Furthermore, we have established that we can continue to use reasoning mechanisms of the λ_v-calculus, because the theory is a conservative extension of the λ_v-calculus.

9 Related Work

This paper draws from a large body of literature in constructive logic, λ-calculus and category theory. It was inspired by Lambek [6] and Scott [12]. Both explore interesting relationships between logic, computation and category theory, using the Curry-Howard isomorphism as a basis.

Constructive theories have been used as theories for program verification and construction for some time now. Martin-Löf's theory of types [7] is one of the most visible of such theories. A good introduction to the the Curry-Howard isomorphism is given by Howard [5].

Danvy and Filinski [1] have studied the control operators `shift` and `reset`. Plotkin [11] introduces the call-by-value λ-calculus. Moggi [9] introduces the notion of monads. Filinski [4] shows how monadic reflection can be achieved with control operators `shift` and `reset`. Other work on building calculi for control operators includes that by Felleisen [3] and Sitaram [13]. Talcott [15] presents an intensional theory of function and control abstractions. Wand and Friedman [17] and Danvy and Malmkjaer [2] have presented formal semantics of reflection.

10 Conclusions

This paper explores connections between computational, logical and monadic reflection. Using constructive principles, we present design guidelines for reflective programming languages. We evaluate some existing programming languages based on these design guidelines and show that only monadic reflection satisfies them.

Keeping in mind these design guidelines and the desirable intensional aspects of reflection, we propose a monadic framework for computational reflection in programming languages. The resulting framework maintains the desirable properties of monads, but allows finer distinctions to be made between representations.

One goal of reflection in programming languages is to enhance the power of a programmer to control how a program is implemented. Such a goal loses its appeal if a side-effect of achieving it is a loss of the ability of the programmer to reason about the program using conventional reasoning mechanisms. In this paper, we have also shown that this goal can be achieved without sacrificing reasoning power. The theory we have developed is a conservative extension of the λ_v-calculus.

References

[1] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the Seventeenth Annual ACM Symposium on the Principles of Programming languages*, pages 151–160, 1990.

[2] Olivier Danvy and Karoline Malmkjær. Intentions and extensions in a reflective tower. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 327–341. ACM, 1988.

[3] M. Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[4] Andrzej Filinski. Representing monads. In *Proceedings of the Twenty-First Annual ACM Symposium on the Principles of Programming languages, Portland, Oregon*, pages 446–457, 1994.

[5] W. Howard. The formulas-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490, 1980.

[6] J. Lambek. From the λ-calculus to cartesian closed categories. In J. R. Hindley and J. P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 375–402, 1980.

[7] P. Martin-Lof. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[8] Anurag Mendhekar and Daniel P. Friedman. Towards a Theory of Reflective Programming Languages. In *OOPSLA 1993 Workshop on Reflection and Meta-level Architectures. Washington D.C. (Available from* http://www.parc.xerox.com/eca*)*, 1993.

[9] Eugenio Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, pages 14–23, 1989.

[10] Robert Muller. M-LISP: A Representation-independent dialect of LISP with reduction semantics. *ACM Transactions on Programming Languages and Systems*, 14:589–616, 1992.

[11] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[12] D. S. Scott. Relating Theories of the Lambda Calculus. In J. R. Hindley and J. P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, 1980.

[13] Dorai Sitaram. *Models of Control and Their Implications for Programming Language Design.* PhD thesis, Department of Computer Science, Rice University, April 1994.

[14] Brian Cantwell Smith. Reflection and semantics in a procedural language (Ph. D. thesis). Technical Report TR-272, Laboratory for Computer Science, MIT, 1982.

[15] Carolyn Talcott. *The Essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation.* PhD thesis, Stanford University, 1985.

[16] Alfred Tarski. The semantic conception of truth. In L. Linsky, editor, *Semantics and the Philosophy of Language*, pages 13–47. University of Illinois Press, 1952.

[17] M. Wand and D. P. Friedman. The mystery of the tower reavealed: A nonreflective description of the reflective tower. *Lisp and Symbolic Computation*, 1:11–38, 1988.