Towards a Theory of Reflective Programming Languages (Extended Abstract)

Anurag Mendhekar Daniel P. Friedman

Department of Computer Science
Indiana University
Bloomington, IN 47406
U.S.A.
E-mail: {anurag,dfried}@cs.indiana.edu

Abstract

This paper attempts to develop a better theoretical understanding of reflective systems. We begin by a developing a reflective extension of the λ_v-calculus and define a simple operational semantics for it based on the infinite tower model described in [10]. We then develop an equational logic from this semantics. The resulting logic is shown to be weak because of reflective properties. We establish properties about this logic and show that it corresponds to the operational semantics.

1 Introduction

Reflection was introduced by Smith [10] as a framework for language extension. He modeled this framework as an infinite tower of interpreters—each interpreter being just a program interpreted by the interpreter above it—with the user's program running at the lowest level. To be a little more explicit, the user program runs at level 0 by an interpreter, which is a program at level 1. This program in turn is interpreted by an identical program at level 2 and so on. Carrying this chain to the limit, we have an infinite tower of interpreters. If we think of a program behaving as a function at level n, it actually is just a data structure being interpreted at level $n+1$. The reflective tower enables one to access this data structure by providing so-called *level shifting* operators. Thus a program running at level n can use the level-shifting operator to look at its execution as viewed by the interpreter at level $n+1$. This operation is called *reification*. The inverse operation is called *reflection*. What is crucial here is that a given program in this reflective framework can behave not only as a function, but also as a data structure that can be examined and manipulated to change its behavior. These properties lead to an easily extensible language since the structures used by the language implementation are accessible to the programmer. The programmer can now define programming constructs that would otherwise have been either impossible or extremely difficult to define. These properties have led to the adoption of reflection as a primary means for language extensibility. An example is the Common Lisp Object System (CLOS) [2, 7], which defines a reflective protocol for manipulating language constructs.

Our goal is to develop a programming logic for reflective languages. We begin with the call-by-value λ_v-calculus [9]. In section 2, we introduce our conventions for this language and then introduce reflective properties in section 3. We then examine the effects of this upon the induced programming logic in section 4. The resulting logic turns out to be extremely weak in order to maintain confluence

and correspondence with the operational semantics. Some standard theorems are proved about this logic in section 4.1. Relation to previous work is discussed in section 5.

2 The λ_v-calculus

We have the following basic syntactic categories in the language of the calculus.

$$\begin{array}{rcl} \langle var \rangle & \in & V, \text{ a denumerable set of variables} \\ \langle const \rangle & \in & B, \text{ a denumerable set of constants} \end{array}$$

Terms in the language are defined as follows.

$$\begin{array}{rcll} \langle value \rangle & \rightarrow & \langle const \rangle & \\ & \rightarrow & \langle var \rangle & \\ & \rightarrow & (\lambda \langle var \rangle . \langle term \rangle) & abstraction \\ \langle term \rangle & \rightarrow & \langle value \rangle & \\ & \rightarrow & (\langle term \rangle \ \langle term \rangle) & application \end{array}$$

In the rest of this paper we use the following conventions. Lower case letters towards the beginning of the alphabet denote constants. The lower case letters towards the end of the alphabet denote variables (e.g. v, w, x, y, z). The upper case letters M, N, P, Q, R, S and T denote terms. Upper case letters U, V and W denote $\langle value \rangle$ terms. $\langle value \rangle$ terms are referred to as *values*. The set of all possible terms is denoted by Λ.

We define the following notion of reduction (β):

$$\{((\lambda x.M)V), M\langle x := V \rangle) \mid V \text{ is a value}\}$$

where, $\langle := \rangle$ is the standard substitution operation. N is appropriately rewritten to avoid capturing bindings from M (α-conversion). Basic constants interact in the following way (δ):

$$\{((ab), \Delta(a, b)) \mid \Delta : B \times B \rightarrow B\}$$

where Δ is a function that defines the behavior of functional constants. We also have the usual notions of free and bound variables. Any term without free variables is said to be *closed*. A term is said to be in *normal form* if it contains no available applications on which β-reduction can be performed. This definition of the language lacks reflective capabilities. The following sections show how these might be introduced.

3 Adding reflection

At this point let us consider what it would mean to introduce reflection in this language. The view of reflective systems that we are taking is *base system + reflective properties*. Traditional reflective systems tend to reify operational entities and provide them as terms that can be manipulated by the program. In this sense, we can never hope to have a generalized theory about reflective systems since the theory will have to take into account the operational behavior of every base system. Traditional calculi, on the other hand, are based upon substitutional equivalence and this gives rise to natural rewrite semantics for the language. One can thus have two descriptions of the same base system and end up with two different reflective systems, each with its own theory.

In this paper we introduce a simplistic reflective behavior based on a rewrite system implementing our language and demonstrate its theory. The model we assume is simple: the computation begins

with a term; a suitable redex is chosen; if none is found, the computation terminates; otherwise the redex is overwritten with its reduct and the process repeats.

Let us now consider what is necessary for adding reflective capabilities to this system. A crucial fact that we noted before is that a term in the language can be applied as if it were a function, or that it could be operated upon as if were a data structure. While there can be no solid proof that this is an absolute requirement, it is common to reflective systems and it is indeed hard to conceive of a system that claims to provide reflective properties and does not have this operation. Thus we provide operations to convert a term to its representation (a data structure) and vice-versa.

Before we begin to think about how this is to be accomplished, let us look at what our representations should be.

3.1 Representations

Since we expect representations to be manipulated inside the language, it is obvious that they must be terms in Λ. Another requirement is that these terms must be in *normal form*. This is important because it avoids having multiple representations for the same term. It is also reasonable to expect that these terms be closed. A third requirement is that it should be possible to mechanically (i.e., within the framework of the language) evaluate a representation so that we can obtain a behavior identical to the term that it represents.

Thus the space of representations is a subset of Λ that meets the above criteria. Let us call this subset Λ_R. For example, one well known and widely used representation is the encoding of the Gödel number of a term as a Church numeral. For the sake of simplicity, we assume representations to be constants, and that these representations can be inductively specified [8]. Such a specification would permit recursive examination of terms.

3.2 The representation function

We can now define a representation function $f_R : \Lambda \rightarrow \Lambda_R$ that gives us the representation of any term in Λ, based on our chosen representation scheme. It is well known that such a function is effective, given a suitable encoding. In a reflective system, however, there are some issues to be dealt with.

If we are to make such an operator available in the language, we would need to fix the order of evaluation. i.e., the behavior of the rewrite system is completely specified and is known to the programmer. For simplicity let us assume a left to right order of evaluation and evaluation of a term in the function position stops once it evaluates to a value. Values are not further reduced.

The following is a well known result, but we present it here again for its relevance.

Theorem 1 *f_R is not expressible in Λ.*

This does not mean that f_R is not computable. It implies that there is no term ϕ in Λ such that $\phi M \rightarrow_\beta M'$ where $M' = f_R(M)$. This implies that the function f_R must be added as a primitive operation. Moreover, the reflective system can never know how it has been implemented. i.e., the operation stands for itself and cannot be expressed in terms of other (non-reflective) operations in the language.

A dual operation is required—one that gives us a term from its representation. One could conceivably define an interpreter for the representations (such as Kleene's E combinator [1]), but such a combinator is only defined for closed terms, i.e., $f_R \circ \mathsf{E}$ is not an identity. Such an operation must again be defined externally, and the reflective system cannot know of its implementation.

We thus introduce two more reductions in our language.

$$\begin{array}{lcl} \uparrow M & \rightarrow & f_R(M) \\ \downarrow M & \rightarrow & f_R^{-1}(M),\ M \text{ is a representation} \end{array}$$

Recall that the order of evaluation is now fixed, M may not be further reduced when applied to $\uparrow$. Muller [8] has introduced such operations in the context of LISP and has shown how confluence can be preserved even when we relax the restrictions on the order of evaluation.

3.3 Turing Completeness and Evaluation Contexts

The λ_v-calculus is a Turing Complete system. The corresponding rewriting evaluator for this calculus is also definable in λ_v-calculus. Consequently, the rewrite system can itself be simulated by the λ_v-calculus. Therefore, if we are to make this rewrite system reflective, we need only provide one other piece of information to the computation being performed. This is the *current evaluation context*[5].

Let us consider the rewrite system again. At any stage in the computation, the only information it needs to maintain is a term and a redex. This forms the evaluation context. Note that we are assuming the tree-reduction model, i.e., it is possible to distinguish different occurrences of syntactically identical terms within the term. Thus we define an *evaluation pair* as a pair (T, P), where T is a term in Λ, and P redex in T. A context is that part of T which does not include P. As in the standard notation, we denote a context as a term with a hole [ ]. Thus the context in the evaluation pair (T, P) is denoted with the term T with P replaced by [ ]. We use $C[\,]$ and its subscripted versions to denote contexts. $C[M]$ denotes that M is the redex in a context $C[\,]$, and so it denotes an evaluation pair. $C(M)$ denotes the first element of the evaluation pair corresponding to $C[M]$.

So the reflective rewrite system need only provide the operation that reifies the current evaluation context. Thus, we extend the function f_R to be defined on evaluation pairs.

3.4 Reifying contexts

The process of "stepping back" to reflect upon a computation is now fairly straightforward. We next give a program the ability to reflect upon its context. We introduce an operation that yields the reified context. We also introduce a dual operation that restores a reified context. As we shall see, these operations are adequate to model movement within the tower.

Brown introduces the concept of meta-continuations to describe the tower [11]. The meta-continuations keep track of the information maintained in each interpreter in the tower. Effectively, meta-continuations maintain a history of reifications in the tower. The meta-continuations, however, are not reifiable. In fact, in the model proposed, attempting to remove meta-continuations from the semantics is shown to introduce unsolvable terms. The reason for this is that meta-continuations are infinite objects (circular, actually), and represent the infinity of interpreters executing above the current level.

We, however, view the tower in a slightly different way—while Brown looks at how far the tower extends above, we look at the how far the tower extends below. This extent is always finite. This finite tower is the complete evaluation context. $\mathcal{I}_{\mathcal{R}}$, a reflective interpreter defined in [6] is also based on this finiteness of the execution context and explicitly builds a finite tower of interpreters. Viewing this finite portion of the tower as the evaluation context raises another question. Does this context have to be explicitly stratified as it is in the finite portion of the tower? An interpreter at a lower level is waiting for a result from the one above it. Computationally speaking, however, the interpreter at the higher level does not "know" this. The value from this interpreter is returned when the starting continuation is invoked. Thus as far as the interpreter at the current level is concerned, it does not know about the stratifications of the continuation. Thus at any point, the interpreter just has a choice between keeping the continuation or discarding it and both these choices are available in our language since it has the ability to install arbitrary evalution pairs. We thus choose to avoid demarcating boundaries of continuations.

It is now time to formalize these operations. We extend the syntax of terms to now include the

operators $\bigotimes$ (reflect) and $\bigodot$ (reify).

$$\begin{array}{rcll} \langle term \rangle & \rightarrow & (\bigotimes \langle term \rangle) & \textit{reflect form} \\ & \rightarrow & (\bigodot \langle term \rangle\ \langle term \rangle) & \textit{reify form} \end{array}$$

We next define applicative contexts. Intuitively, these are contexts that would be generated in the usual evaluation order of the rewrite system. We define these as follows:

$$\begin{array}{rcll} \langle C[\,] \rangle & \rightarrow & [\,] & \textit{(empty context)} \\ & \rightarrow & (\langle C[\,] \rangle \langle term \rangle) & \\ & \rightarrow & (\langle value \rangle \langle C[\,] \rangle) & \\ & \rightarrow & (\bigodot \langle C[\,] \rangle \langle term \rangle) & \\ & \rightarrow & (\bigodot \langle value \rangle \langle C[\,] \rangle) & \\ & \rightarrow & (\bigotimes \langle C[\,] \rangle) & \end{array}$$

Henceforth, we will use the previously introduced notation for contexts to apply correspondingly to applicative contexts. This definition of applicative contexts enforces a strict order of evaluation from left to right. We can finally describe the operational semantics of the rewrite system. The rewrite rules are as follows:

$$\begin{array}{rcll} C[(\lambda x.M)V] & \rightarrow & C[M\langle x := V \rangle] & \\ C[\bigodot VU] & \rightarrow & C[VV'] & V' = f_R(C[N]), U = f_R(N) \\ C_0[\bigotimes V] & \rightarrow & C_1[N] & f_R(C_1[N]) = V \end{array}$$

In order to maintain the internal consistency of the rewrite system we must restrict the context representations that $\bigotimes$ can install. We could otherwise install an evaluation pair for which no rewrite system rule is defined. We require that the context $C_1[\,]$ must be an applicative context.

We use $\rightarrow^*$ to denote successive applications of the rewrite rules. We define the function E on terms to be $E(M) = V$ when $M \rightarrow^* V$. We say that two terms M and N are operationally equivalent if they can be substituted for each other in an *arbitrary* context (i.e., not just an applicative context). More formally, $M \approx N$ iff for *any* program context C such that $C[MN]$ is closed, $E(C[M]) = E(C[N])$ if either is defined and both are undefined otherwise.

Notice that the $\bigodot$ operator is similar to the `call/cc` operator found in languages such as Scheme. The difference here is that the context contains more than just the control context in that it can be examined as data here. Since these two operations work on exact snapshots of the context, it is a trivial matter to define the operations $\uparrow$ and $\downarrow$ in terms of $\bigotimes$ and $\bigodot$. We therefore do not treat these operators separately in the rest of the paper.

3.5 Renaming variables

One problem, however, remains in this description of the rewrite system. With function application, there might be implicit α-conversion. The rewrite semantics must make it clear what rule it applies for renaming bound variables before doing the substitution. Approaches to doing substitution have been described by Curry *et al.* [4] and de Bruijn [3]. While machine implementations of the λ_v-calculus are implicitly in the style of de Bruijn renaming, we use the former for simplicity. Curry dictates a canonical sequence of variables and picks the first variable in this list that is not a free variable in the argument term. We will not explicate this rule further and assume that the substitution operation is suitably redefined.

3.6 Some Examples of Reflective Programming

Let us consider the simple example of defining the $\downarrow$ operator in terms of the other reflective operators. Let us assume that the encoding is now fixed and suitable functions that extract and combine

sub-terms from the representation of a context have been defined. For simplicity, let us assume a call-by-value order of evaluation. The way we define the $\downarrow$ operator is as follows. We first obtain the representation of current context. We then replace the redex in this context with the operand of $\downarrow$ and install this new context as the current context. Thus we have:

$$\downarrow =_{def} (\lambda r.\bigodot(\lambda c.\mathit{replace\text{-}redex}\ c\ r)a)$$

where a is a constant (representing an applicative context) that is ignored. Here *replace-redex* is a term which takes the representation of an evaluation pair and replaces the redex with the second argument. The second argument must be the representation of a term.

Another example is that of defining the control construct call/cc, found in languages such as Scheme and ML. Such an operator will obtain the context in which it is invoked, package it up in a procedure and when this procedure is applied, the captured context will be restored. Thus we can define call/cc as follows:

$$\mathsf{call/cc} =_{def} (\lambda p.(\bigodot(\lambda c.p(\lambda v.\bigotimes(\mathit{replace\text{-}redex}\ c\ (\downarrow v))))a))$$

Again, a is a similar constant that is ignored.

4 The Programming Logic

In this section we develop a calculus for the operational semantics described above. In the following subsections, we prove certain properties about the calculus and give an example of reasoning with this calculus.

Reflective operations in the language are defined in terms of the rewrite system executing the language. While this may not be a problem operationally, it is problematic in the resulting calculus. The meaning of terms is now dependent upon the context in which they are evaluated. This can be seen simply by observing the term that returns the reified context as its value. Equational logics work by locally reducing terms and using this reduction in contexts where these terms occur as sub-terms. This usually means that equational logics inherently have the freedom of making reductions in a relatively unrestricted order. Reflective operations, however, do not permit this freedom as can be seen from the example below.

$$([(\bigodot(\lambda x.x))]N) \rightarrow f_R((\bigodot(\lambda x.x))N)$$

If we choose to reduce N before we reduce the $\bigodot$ term, we get

$$f_R([\bigodot(\lambda x.x)]N')$$

We could try to repair this by specifying that N cannot be reduced if it is being applied to a $\bigodot$ form, as has been done in [8]. This automatically forces us to reduce the term in the function position as much as we can before we reduce the term in the argument position. Where do we stop the reduction? Since we are trying to provide a logic for the rewrite system, it makes sense to stop the reduction when the term is a value, or a reify(reflect) form with value arguments. Moreover, because of this, values in function positions should not be reduced further since this could give rise to two different answers if the argument term is a reify form. Also, the restriction of checking for the reify form before reducing an argument can very easily be defeated by wrapping the reify form inside a lambda expression. There seems to be only one solution: values in any position must not be reduced further and reduction should proceed strictly in the prescribed order of the rewrite system. Since λ-abstractions are values, this strong restriction gives us a very weak equational logic in which λ-abstractions only reduce to themselves.

The logic we define is similar in spirit to [5]. The idea is that the reification operation captures its context incrementally. Moreover, this capturing must correspond to the atomicity of the reification

$$\frac{(M,N) \in \beta \cup \mathit{reflect}_r \cup \mathit{reflect}_l \cup \mathit{reify}_r \cup \mathit{reify}_l}{M \rightarrow_r N} \quad (0)$$

$$\frac{M \rightarrow_r M' \quad M \text{ is not a value}}{MZ \rightarrow_r M'Z} \quad (1)$$

$$\frac{N \rightarrow_r N'}{VN \rightarrow_r VN'} \quad (2)$$

$$\frac{R \rightarrow_r R'}{(V(\bigotimes R)) \rightarrow_r (V(\bigotimes R'))} \quad (3)$$

$$\frac{N \rightarrow_r N' \quad N \text{ is not a value}}{(N(\bigotimes R)) \rightarrow_r (N'(\bigotimes R))} \quad (4)$$

$$\frac{R \rightarrow_r R' \quad R \text{ is not a value}}{((\bigotimes R)N) \rightarrow_r ((\bigotimes R')N)} \quad (5)$$

$$\frac{N \rightarrow_r N' \quad N \text{ is not a value}}{(N(\bigodot MR)) \rightarrow_r (N'(\bigodot MR))} \quad (6)$$

$$\frac{M \rightarrow_r M' \quad M \text{ is not a value}}{(V(\bigodot MR)) \rightarrow_r (V(\bigodot M'R))} \quad (7)$$

$$\frac{R \rightarrow_r R' \quad R \text{ is not a value}}{(V(\bigodot UR)) \rightarrow_r (U(\bigodot VR'))} \quad (8)$$

$$\frac{M \rightarrow_r M' \quad M \text{ is not a value}}{((\bigodot MR)N) \rightarrow_r ((\bigodot M'R)N)} \quad (9)$$

$$\frac{R \rightarrow_r R' \quad R \text{ is not a value}}{((\bigodot VR)N) \rightarrow_r ((\bigodot VR')N)} \quad (10)$$

Figure 1: Schema for the calculus

operation in the rewrite system. i.e., it must not force reductions in any part of its context. It must also reflect the order of evaluation. We define the following notions of reduction:

$$\{(((\bigodot\ U\ V)N), (\bigodot\ U\ app_r(V, f_R(N)))) \mid U, V, N \in \Lambda, U, V \textit{ are values}\} \cup$$
$$\{((\bigodot(\bigodot\ U\ V)N), (\bigodot\ U\ \textit{reify-rep}(f_R(\bigodot UV), f_R(N)) \mid U, V \in \Lambda, U, V \textit{ are values}\} \cup$$
$$\{((\bigotimes(\bigodot\ U\ V)), (\bigodot\ U\ \textit{reflect-rep}(f_R(\bigodot UV)) \mid U, V \in \Lambda, U, V \textit{ are values}\} \cup \qquad (\textit{reify}_r)$$

where app_r is a function whose first argument is a context representation, the second argument is a term representation and it incrementally extends the context representation with the term representation on the right in the obvious way. Its dual app_l extends the context representation to the left. The corresponding functions *reify-rep* and *reflect-rep* build representations corresponding to the reify and reflect forms. The corresponding reduction for the left context is:

$$\{((N(\bigodot\ M\ R)), (\bigodot\ M\ app_l(R, f_R(N)))) \mid M, N, R \text{ are values}\} \cup$$
$$\{((\bigodot\ U(\bigodot\ V\ W)), (\bigodot\ V\ \textit{reify-rep}(f_R(U), f_R(\bigodot\ V\ W))) \mid U, V, W \in \Lambda, U, V, W \textit{ are values}\} \qquad (\textit{reify}_l)$$

The operation $\bigotimes$ ignores its context. We therefore define:

$$\{((V(\bigotimes U)), (\bigotimes U)) \mid U, V \in \Lambda \text{ are values}\} \qquad (\textit{reflect}_l)$$
$$\{(((\bigotimes U)N), (\bigotimes U)) \mid U \in \Lambda \text{ is a value}\} \qquad (\textit{reflect}_r)$$

The idea is that these reductions would bubble up the $\bigotimes$ and $\bigodot$ operators to the top level where they would be reduced by special rules called *computation* rules. The rule for $\bigotimes$ will simply install the term whose representation is the first argument to the operation as the term to be operated upon. i.e.,

$$\{(((\bigotimes U), N) \mid U \text{ is a value}\} \qquad (\textit{reflect}_{top})$$

where N is the term corresponding to the evaluation pair represented by U. This rule needs some clarification. The $\bigotimes$ operation in the rewrite system installs not only the term being rewritten but also the sub-term to be rewritten next. Since the equational logic does not specifically have a "next redex", the restriction on the kinds of context representations a reflect operation can handle ensures that the only available redex in the term corresponds to the particular sub-term that is to be rewritten next.

Similar considerations also apply for the computation rule for $\bigodot$:

$$\{((\bigodot\ V f_R(C[N])), (C(V(f_R(C[N]))))) \mid V \text{ is a value}\} \qquad (\textit{reify}_{top})$$

The schema for the calculus is given in figure 1. We define $\rightarrow_R$ to be the reflexive, transitive closure of $\rightarrow_r$. We also define $=_R$ to be the corresponding equivalence relation. We then add the computation reductions $\textit{reflect}_{top}$ and $\textit{reify}_{top}$ to get the final notion of reduction:

$$\rhd^r = \textit{reify}_{top} \cup \textit{reflect}_{top} \cup \rightarrow_R$$

The reflexive, transitive closure of this relation is denoted by $\rhd^R$. We define $=^R$ to be the equivalence relation defined from $\rhd^R$. Apart from having enforced a strict order of evaluation, the thing to be noticed about this schema is that values cannot be further reduced. This is the price one must pay for the consistency of the logic in the presence of reflective operations. The resulting logic is

extremely weak in that functions, which would otherwise behave identically, cannot be proved equal. Although it sounds negative, this is actually consistent with our paradigm of reflective programming since there are instances where one would like to differentiate between values which would behave identically in other situations. The $\uparrow$ operation is a simple example.

4.1 Properties of the logic

In this section, we prove some properties about the logic. The presentation here is similar to the one in [5], which in turn follows [1] and [9].

Lemma 1 *For any term M, there is at most one term M', such that $M \rightarrow_r M'$.*

Proof: By a simple induction on the structure of terms.

From this we have:

Lemma 2 *$\rightarrow_R$ is Church-Rosser.*

Proof: By the previous lemma, and the fact that $\rightarrow_R$ is the transitive closure of $\rightarrow_r$.

The computation rules are also trivially Church-Rosser. Since no non-reflexive $\rightarrow_R$ reductions are defined on values, and the computations are defined only on terms that are composed of values, these two reductions commute. By the Hindley-Rosen theorem (see [1]), their union is Church-Rosser. Thus, the theory is consistent.

Since there is only one order of reduction, the standardization properties of the logic also follow trivially.

We should now establish the equivalence between the calculus and the operational semantics.

Lemma 3 *$M \rightarrow N$ implies $M \rhd^R N$.*

Proof: By cases on the rewrite system rules and induction on the structure of M.

This lemma entails the following result.

Lemma 4 *$M \rightarrow^* N$ implies $M \rhd^R N$.*

Lemma 5 *$M \rhd^R V$ then $M \rightarrow^* V$.*

Proof: Since $\rightarrow_R$ is transitively closed, two or more consecutive applications of $\rightarrow_R$ can be converted into a single application of $\rightarrow_R$. We can thus shorten the reduction from M to V such that it is only a sequence of $\rhd^r$ reductions with no sub-sequence of consecutive $\rightarrow_R$ applications being longer than one. The result then follows by induction on the length of this sequence.

These lemmas help us conclude:

Theorem 2 *(Simulation). $M \rightarrow^* V$ iff $M \rhd^R V$.*

Since the reductions in the logic are forced to follow the reductions in the rewrite system, we have the following theorem:

Theorem 3 *(Operational eqiuivalence) For M, N in Λ,*

1. *if $M =_R N$, $M \approx N$*
2. *if $C[M] =^R C[N]$, for any applicative context C, them $M \approx N$.*

4.2 Reasoning with this calculus

Let us consider the $\downarrow$ operator again. Based on its definition, we can conclude that $C[\downarrow f_R(M)] =_R C[M]$ (1). and from the operational equivalence theorem above, it follows that $\downarrow f_R(M) \approx M$. The proof of (1) above can be carried out by a simple induction on the structure of applicative contexts.

5 Related Work

This work is based on earlier work by Muller [8] and Felleisen et al. [5]. Muller examined $\uparrow$ and $\downarrow$ in the context of LISP. He developed a calculus for reasoning about these operators. Felleisen et al. developed an equational theory for control constructs such as call/cc. We have combined features from both these calculi. Plotkin [9] was the first to develop the λ_v-calculus.

6 Conclusions

The infinite tower model described by Smith [10] can really be described in much simpler operational terms. While the system we have described here does not support first-class environments, we believe that it captures the essence of reflective programming. We pointed out that there are limitations to the reflective capability of systems and that reflective systems depend crucially on a defining system that cannot be reflected upon.

Reflection is a very attractive solution for building extensible systems. This paper shows that reasoning about such systems is much harder. Language designers must take this into account when introducing reflection into languages. We developed a calculus for reasoning about programs in this system. The resulting calculus was shown to be consistent, although extremely weak. This weakness is a consequence of the reflective properties of the system. Equational logics derive their power from behavioral equivalence between terms. Since behavioral equivalence between terms is significantly reduced in reflective systems, the corresponding equational logic has turned out to be weak.

The weakness of the logic stems from the fact that the operational semantics distinguishes between terms that would behave similarly otherwise. If we were to reduce this distinction, we could strike a compromise between a more powerful logic and adequate operational reflection. Meta-Object protocols [7] as in CLOS are aimed at this sort of compromise. Although weak, this logic is only a starting point for developing theoretical models for reflective languages. The insight gained by developing this logic will help us design reflective programming languages that will be easier to reason about and implement.

References

[1] Barendregt, H.P., The Lambda Calculus: Its Syntax and Semantics, (North-Holland, Amsterdam, 1981).

[2] Bobrow, D.G., DiMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., Moon, D.A., A Common Lisp Object System Specification: 1. *Lisp and Symbolic Computation* 1 3/4 (1989).

[3] de Bruijn, Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, *Indag. Math.* 34 (1972).

[4] Curry, H.B., Feys, R., Combinatory Logic, Vol. II, Studies in Logic (North-Holland, Amsterdam, 1958).

[5] Felleisen M., Friedman D.P., Kohlbecker E., Duba B., A Syntactic Theory of Sequential Control, *Theoretical Computer Science* 52 (1987).

[6] Jefferson S., Friedman D.P., A Simple Reflective Interpreter, *Proceedings of the International Workshop on New Models for Software Architecture, Reflection and Meta-level Architecture*, (Tokyo, 1992).

[7] Kiczales, G., des Rivieres J., Bobrow, D.G., The Art of the Meta-Object Protocol, (MIT Press, 1991).

[8] Muller, R., M-LISP: A Representation-Independent Dialect of LISP with Reduction Semantics. *ACM Transactions on Programming Languages and Systems*, Vol. 14, 4, (1992).

[9] Plotkin G.D., Call-by-name, Call-by-value and the λ-calculus, *Theoretical Computer Science* 1 (1975).

[10] Smith, B.C., Reflection and Semantics in a Procedural Language, MIT-LCS-TR-272, MIT, (Cambridge, 1982).

[11] Wand M., Friedman D.P., The Mystery of the Tower Revealed: A Nonreflective Description of the Reflective Tower, *Lisp and Symbolic Computation* 1 (1988).