

Reflecting the computation system of constructive type theory in itself*

Stuart F. Allen
Robert L. Constable
Douglas J. Howe

Cornell University
Ithaca, NY 14853

Abstract

The computation system of constructive type theory is open-ended so that theorems about computation will hold for a broad class of extensions to the system. We show that despite this openness it is possible to completely reflect the computation system into itself in a clear way by adding simple primitive concepts that anticipate the reflection. This work provides a method to modify the built-in evaluator and to treat the issues of intensionality and computational complexity in programming logics and provides a basis for reflecting the deductive apparatus of type theory.

In this abstract we use the term “reflection” to refer to grammatical constructions which allow a language to talk about itself. This capability is important in natural language, and in fact was used in the first sentence of this abstract (as well as in this sentence). Reflection is also an important mechanism in formal languages. In Lisp it is used to provide an extensible syntax. Formal logical calculi also use it to provide an extensible inference system [DS79], by allowing users to state new rules of inference and prove that they are sound. Reflection need not be explicitly provided because it can sometimes be achieved by a technique known as *gödelization*, used by Gödel to prove his incompleteness theorem by reflecting the relation “p is a proof of P” inside the pure language of arithmetic.

This mechanism not only provides a basis for reasoning about computation, but also a means of modifying evaluation, say to make it more efficient. For instance it is possible to provide other function evaluation procedures such as “call by value” in addition to the basic lazy evaluation. Reflection also provides a basis for reasoning about syntax. We can define basic operators on terms at the reflected level, e.g. substitution, renaming, pattern matching, unification, etc. These can be given just as at the metalevel, providing an internal account of the basic system operators. Moreover, because the term structure is so general, its reflection provides a way

*Supported in part by NSF grant CCR-8616552 and ONR grant N00014-88-K-0409.

to study syntax and metamathematics rigorously inside the system in such a way that the results are applicable to the reflected system. Moreover, because the theory is constructive, the metatheorems are also applicable.

We believe that reflection will be a good mechanism for treating at least two basic concepts that have proved troublesome in formal programming logics, namely the notion of the *structure* of algorithms, formulas, proofs, etc. and a notion of *computational complexity*. In some ways reflection seems like an “obvious way” to treat these concepts, but formal reflection mechanisms are subtle and sometimes opaque. So there is a bias away from them (see for example [Wey80, Sha84]), and they have not been used in this way. One of the results of our work is a clean reflection mechanism which will support an account of structural and computational complexity in programming logics. This is achieved by providing primitives that anticipate reflection (as opposed to taking a language as given, say number theory, and finding a gödelization).

One reason that we can make the reflection mechanism especially clear is that we describe it in constructive type theory, an exceptionally rich language without appeal to a vague concept such as semantic attachment [Wey80]. One reason it is so useful is that the theory it reflects is so rich, unlike the situation with a quantifier free theory based on Lisp syntax [BM81]. However, one of the characteristics of this theory is its *open-ended computation* system. The “openness” notion is that the theory should remain sound when new computational forms are added as primitives as long as they satisfy certain simple conditions. This feature of type theory is very important to its role in the foundation of constructive mathematics because in mathematics the notion of construction is never finally closed off. We maintain that this feature is also interesting in programming languages and logics. For example, building a programming logic on the principle guarantees that all rules and theorems will remain valid if a new operator is added, for example a new form of iteration, or a non-deterministic choice operator or a form to represent a more efficient access mechanism to a data structure or forms to reflect the theory itself. (For example, the underlying computational system of Nuprl has been extended several times, once to add induction for recursive types [Men88], again to allow generators for infinite objects [Men88], then a fixed point operator [Smi89]. Similarly Martin-Löf type theory has been extended twice since 1973 [ML73], to add well-founded trees [ML82] and lazy numbers. Because of the open-endedness principle, no previously established theorems became false in these extensions. Other programming languages, notably Lisp, have undergone similar evolution.)

In a sense, open-endedness is assumed by programming language designers who freely add new constructs in the evolution of a language, e.g. ?. Difficulties do not show up until one tries to axiomatize the language, then rules must often be amended to take account of new kinds of execution. Constructive type theory is designed from the start to accommodate these extensions without changing the rules. Our results show that it is possible to preserve this important design principle and nevertheless allow a reflection mechanism. It is not clear *a priori* that this could be done. However we show that the mechanism we provide completely reflects the openended evaluation procedure (Property 2 below).

One of the by-products of our analysis is a new method for explaining type theory itself. This can be done because the type theory notation for inductive definitions and the basic types and sets used to define the theory itself are isomorphic to standard informal accounts of these concepts. So the definition of the syntax and computation rules is presented first in this natural high level notation which then turns out to be exactly the notation being reflected into the internal definition of a formal language. This capability is not so strikingly useful for less rich theories.

We put off the more complex issue of how to reflect the deductive apparatus of type theory, but the practical value of such a reflection is well illustrated by Howe’s development of a reflected theory of term rewriting for a fragment of type theory [How88, How89]. The work described here is a necessary prerequisite to “deductive reflection”, and is sufficiently involved that it bears this separate treatment. In particular, this is a good basis for exploring the notion of complete reflection in programming languages.

1 Preliminaries

terms

Constructive type theory can be defined starting with a class of terms which will denote the types and their inhabitants. For example, there may be a term for the list induction combinator, following [] it has the form $list\ ind(l; b; h, t, v.g)$. In this term, l, b and g are *subterms* and h, t , and v are *bound variables* whose scope is g . All occurrences of these variables in g are *bound*. Typically there will be many term constructors such as pairing of term a and b , often denoted $\langle a, b \rangle$; injecting a term a into a union, often denoted $inl(a)$ or $inr(a)$. In Martin-Löf ’82 there are 30 such terms, in Nuprl there are ?. It simplifies matters to regularize term formation. Here we adopt the convention that all terms are written in the form $op(\bar{v}_1.t_1; \dots; \bar{v}_n.t_n)$,

where \bar{v}_i is a vector of *bound variables*, say v_{i1}, \dots, v_{in_i} and t_i is a *subterm*. We can also identify the *operator name*, op , as a component. If there are n subterms of the term, we say that the operator has *arity* n .

Terms are built inductively starting from variables and constants. We treat variables as a special category of term, but constants, such as 0 and nil can be regarded as special cases of the general form where there are no subterm; so 0 and nil are just operators of *arity zero*. The inductive character of terms justifying the method of building new terms or proving properties of them by *induction on terms*. For example, we might define substitution by *term-induction* of the following kind. The result of substituting a for variable x in term t is denoted $subst(a; x; t)$.

To define this operation, let $arity(t)$ denote the arity of the operator and let *bindings* and *subterms* be functions which list the bound variable for each subterm position and the subterm of that position, e.g. for $op(\bar{v}_1.t_1; \dots; \bar{v}_n.t_n)$

$$\begin{aligned} arity(op) &= n \\ bindings(i) &= \bar{v}_i \\ subterm(i) &= t_i \quad \text{for } 1 \leq i \leq n. \end{aligned}$$

If we also use a special form for term induction, substitution is defined as

$$subst(a; x; t) =$$

$$\text{term_ind}(t; \text{ if } t = x \text{ then } a \text{ else } t; \\ \text{ op, bvd, subst, v. (make_term(op, bvd,} \\ \lambda i. \text{ if } x \text{ in bvd}(i) \text{ then } \text{subst}(i) \\ \text{ else } v)))$$

evaluation

Computational meaning in type theory arises from rules for evaluating terms, e.g. for saying that $1 + 1$ evaluates to 2 . These are expressed as a relationship between terms, say s *evaluates to* t . A more involved example than $1 + 1$ is needed to see the key points. We consider the evaluation of the list induction form.

If lists are built from nil by the operation of adjoining an element a to a list l , producing a new list $a.l$, then the rules for computing with this term are:

$$\begin{aligned} \text{list_ind}(\text{nil}; bh, t, v.g) &\text{ reduces to } b \\ \text{list_ind}(a.l; b, h, t, v.g) &\text{ reduces to } g \end{aligned}$$

with a, l and $\text{list_ind}(l; b; h, t, v.g)$ substituted for h, t, v respectively in g .

In general the evaluation relation will be of the form $op(\bar{v}_1.t_1; \dots; \bar{v}_n.t_n)$ evaluates to t . For a fixed closed system, such as the rules in Martin-Löf '82 or Nuprl as of 1986 [CAB⁺86], there are a finite number of such rules, and they determine the *computation system*.

open-endedness

For reasons discussed in the introduction, we are interested in *open-ended computation systems*. This means that we must be able to provide for the introduction of new terms and their incorporation into the evaluation relation. It is not clear how this can be specified in such a way that the system is truly open-ended yet can be completely described to the point where the mechanism can be reflected into the theory itself. We consider the issue further before introducing our solution.

We know a lot about the general structure of terms. We can require without loss of essential generality that they all look like $op(\bar{v}_1.t_1; \dots; \bar{v}_n.t_n)$. So we know that terms have operators, *op*, *subterms* t_i and *bound variables* governing t_i, \bar{v}_i . But how can we specify the evaluation procedure on a new term? We must in some way extend the evaluation relation, but how can this be done in such a way that evaluation is an evaluation function in the usual sense?

The first key to solving this problem is to recognize that all new terms can be built from one open-ended type which is much simpler, a type of *operators* or *term constructors*. This type of operator is open-ended, unbounded and discrete (we can decide whether any two operators are equal or not). We can think of it as a way of indexing the type of terms which inherits its openness from the operator type.

The second key to solving the problem is to require that with each new operator there is a rule for evaluating it. We can think of this as a *fragment* of the overall evaluation relation which is built uniformly from the evaluation fragments. So as part of introducing a new operator is the obligation to say how it evaluates.

omissions

This paper is about reflection, so we do not pursue here a detailed justification of the mechanism which would include a discussion of how to build evaluation as a fixed point of a procedure for using evaluation fragments.

2 Defining reflection

anticipation

We now turn to the question of whether we can completely reflect the open-ended computation system described above. Our hope is that by using type theory (rather than gödelizing) we can offer an especially clear analysis. Just as we have reduced the notion of open-endedness to one specific type, the operators, we reduce reflection to the notion of *anticipating it in the type system*. Given the type system before adding these new types, the extension illustrates the power of open-endedness to provide useful new forms of computation.

design criteria

We can determine what new type and members are needed by describing precisely the computation system as set out in section 2. One can view this effort as an explanation of the basics of computation in the theory itself. As such this account should be a clarification and refinement of what has gone before in section 2.

There must be types for *operators*, *variables* and *terms*. These will be denoted $Op, Var, Term$. In general we form the internal notions by capitalizing the first letter of the informal type used in the metalanguage. In addition to these types we need the functions

$$\text{Arity}: Op \rightarrow N, \text{Bvnum}: o : Op \rightarrow i : [1, \text{Arity}(o)] \rightarrow N$$

(Here we employ the dependent function space constructor to give an exact typing of these functions.)

There is also a formal analogue of term induction, given by the new operator $Term_ind$. It has arity 2 and is written generally as $Term_ind(t; b; o, s, b, v.g)$ where intuitively 0 picks out the operator of t , s the subterm, b the bound variables and v the value of $Term_ind$ on subterms.

There is a relation to represent evaluation and a function, Val , to compute the value of a term when it exists. The relation s *Evaluates_to* t

should hold when s represents s' and t represents t' and s' evaluates to t' . The term $Val(s)$ is defined when s represents some s' and s' evaluates to t' . The result should be some *standard representation* of t' , such as the fully evaluated term representing t' .

We know then that to make sense of terms we will need to understand these relationships: represents an operator, represents a variable, represents the i -th number, and the knowledge of how to define a *standard* representative of a term.

implementing reflection

We now want to specify a particular way to meet the design criteria listed above. We have a particular sort of type theory in mind. It should be an open-ended extensional system with at least the primitives of Martin-Löf '82 or Nuprl. Depending on certain other details, there are various ways to proceed. For example, if there are recursive types, then *Term* can be defined naturally using them, otherwise it must be encoded somewhat, say using the W-types of Martin-Löf '82. If there are partial types, as in extensions of Nuprl [book,Smi89], then Val can be interpreted as a partial function. Otherwise we can use *Evaluates_to* to specify its domain exactly and make it a total function.

However *Term* is defined, there must be a way to inject *Op* and *Var* into it because they are base cases of the inductive definition. We also need a way to guarantee that there are terms which represent the new operators just defined. Implicitly by adding *Op* and seeking a complete reflection of the theory, we will need names for the names of operators, then names for the names for the names, etc. For example, 0 is a term, *zero* will be the operator name for it used in building *Terms*. So $in_op(zero)$ will be the injection of the operator *zero* into *Term*, and it will represent 0. But now we need a term to represent *zero* itself.

We adopt the convention that the elements of *Op* are taken to denote themselves, since there is no interesting structure to these elements. This is accomplished by having a special injection of *Op* into *Term* to denote the operator itself. We call these degenerate terms. They are built using in_degen . So $in_degen(zero)$ represents *zero*.

We need to be able to analyze variables, so they are built as lists of letters, letter list. In accordance with the prescribed method of introducing a new operator such as Val , we must tell how to evaluate it by giving its evaluation fragment. It is just this:

to evaluate $Val(t)$, t must evaluate to a term, say $op(\bar{v}_1.t_1; \dots; \bar{v}_n.t_n)$. We apply the evaluation fragment of op .

3 Properties and Completeness

There is a sense in which the syntax and computation rules are completely reflected by this mechanism. The following properties and theorems give some sense of this.

Property 1: $\forall s, t : Term.$ (*s Evaluates_to t*) is inhabited
iff there are terms s', t' such that
 s' is represented by s &
 t' is represented by t &
 s' evaluates to t' .

Property 2: For all terms s, s', t, t'
if $eval(s') = t'$ & s' is represented by s and
 t' is represented by t ,
then $eval(Val(s)) = t$.

Property 3: $\forall s, t : Term$ (*s Evaluates_to t*) \Rightarrow
 $Val(s) = t$ in Term

Property 1 is the definition of *Evaluates_to*. Property 2 characterizes the operational definition of *Val* given in section 3. It says that *Val* provides a complete reflection of the meta operation *eval*. Property 3 is an internal characterization of *Val*. It is the kind of fact which would be taken as an axiom if we were reflecting the proof mechanism.

There are other ways in which we can express completeness. To show that the syntax of terms is reflected in *Term*, we can introduce a relation on *Term*, *s Represented_by t* (or *t Represents s*). It turns out that the relationship can be defined.

References

- [BM81] R.S. Boyer and J.S. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science.*, pages 103–84. NY:Academic Press, 1981.
- [CAB⁺86] Robert L. Constable, S. Allen, H. Bromely, W. Cleveland, and et al. *Implementing Mathematics with the Nuprl Development System*. NJ:Prentice-Hall, 1986.
- [DS79] M. Davis and J.T Schwartz. Metamathematical extensibility for theorem verifiers and proof checkers. In *Comp. Math. with Applications, v. 5*, pages 217–230, 1979.
- [How88] D. Howe. Computational metetheory in nuprl. *CADE-9*, 230:404–415, 1988.

- [How89] D. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1989.
- [Men88] P. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.
- [ML73] Per. Martin-Lof. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73.*, pages 73–118. Amsterdam:North-Holland, 1973.
- [ML82] Per. Martin-Lof. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. Amsterdam:North Holland, 1982.
- [Sha84] N. Shanker. Towards mechanical metamathematics. Technical report, Institute for Computing Science, University of Texas at Austin, 1984. Tech. report 43.
- [Smi89] S.F. Smith. *Partial Objects in Type Theory*. PhD thesis, Cornell University, 1989.
- [Wey80] R. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.