

The Semantics of Reflected Proof*

Stuart F. Allen
Robert L. Constable
Douglas J. Howe
William E. Aitken

Cornell University
Ithaca, NY 14853

Abstract

We begin to lay the foundations for reasoning about reflected proofs in theories where proofs have computational significance, and we demonstrate several applications of the basic concepts in computer systems which implement these theories. The first key result is the definition of a single type of proof which can mention itself (reflected proof) using a new technique which finds a fixed point of a mapping between object language and metalanguage. The second main result is an argument that the proof type is valid. This single type contrasts with hierarchies of types used in other approaches to accomplish the same classification.

We believe that the mechanism of reflection is fundamental in building proof development systems, and we illustrate its power with applications to automating reasoning, extending type theories and describing modes of computation.

1 Introduction

The propositions-as-types principle [13, 22] is a way of assigning mathematical meaning to constructive proofs. The idea can be extended to classical proofs (giving a semantics of evidence [8]). This principle has led to useful applications, some under the banner “proofs-as-programs” [4, 9, 22] (which might better be called *computational content* of proofs as programs in this setting). The principle has been the central organizing idea in several logical theories, notably in logics used to support programming systems and proof development systems.

Formal proofs of interesting ideas, especially those arising from using the implemented systems, tend to be large objects. So in practice, only computational content is *extracted*, and some mechanism is used to shorten and abbreviate proofs. One of the most effective is the use of *tactics* LCF[15], Nuprl [9], λ Prolog [24], HOL [14]). Proofs then are *tactic-trees*. These are proofs represented as trees in which certain steps are justified by writing programs which compute subtrees.

In all heretofore existing systems, the tactic-trees use two languages, one for the proof object, one for the tactic program written in, say, ML, Prolog, λ Prolog, SML, Lisp or CAML. We can say *justification = rule + tactic*. But if proofs can sometimes be construed as programs, then perhaps only one language is needed, and the tactics can be represented by proofs as well. The question we consider is whether there can be a single notion of tactic-tree proof, in which the tactics are the programs coming from the *tactic-trees as program principle*, i.e.

$$\textit{justification} = \textit{rules} + \textit{tactic-from}(\textit{proof}).$$

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

We show that there is such a class of proofs. This requires a new kind of argument which uses the reflection of the proof concept in a formal theory to define the notion of proof in that theory. There is an interesting closure principle revealed here based on a fixed point construction over a class of proof-like trees which interleave objects from the object language and the metalanguage.

The ensuing concept of reflected-proof is noteworthy because it is encapsulated in a single recursive type, in contrast to other approaches which result in an infinite tower of types or languages [21, 28] or would result in an infinite tower if extended to provide the same closure [29]. Moreover, the issues raised in other uses of reflection are clearly present here, for instance the notions of *stance* and *connection* in the *procedural reflection* of Smith [26] where he says “when you take a step backward to reflect, you need a place to stand, with just the right combination of connection and detachment. But they are easier to pick out in this setting because they occur in a single type rather than in Smith’s “infinite tower of procedural self-models.”

Our notion of a reflected proof turns out to be more useful than the unreflected kind. We discuss seven applications of the concept in three different areas: theorem proving, type definition, and functional program evaluation. First, we show how to shorten proofs and speed-up proof generation and checking by proving that the tactics are correct. Second, we show how to define a kind of polymorphism in theorems involving universe types. Third, we show how to define evaluation mechanisms provably equivalent to the built-in evaluator yet more efficient on certain terms. Fourth, as a case of the third application, we show how to detect expressions that can be destructively updated, thereby allowing a procedural evaluation for them. Fifth, we show one approach to expressing computational complexity. Sixth, we show how to experiment with new systems ideas in a mathematically controlled way by running code from incomplete proofs of metatheorems. Seventh, we show how to use proof transformations to build new logics on top of the built-in logic of a theory.

This abstract first defines the reflected proof type and shows that it is valid in the sense that incomplete proofs map true sequents to true sequents. We spend most of the abstract on this point which is subtle. In section 3 there is a brief discussion of some of the applications. The full paper can treat these in more detail.

2 Semantics of Proofs

context

The context of our work is that we are standing outside of a formal language L (traditionally called the *object language*). The syntax of L is given by an inductively defined class of terms. This is a type in the observer’s language or *metalanguage*, ML . We are especially interested in those languages L with a computable evaluation relation on terms [13, 9, 15, 17, 25].

We are concerned with languages L that define types. From outside of L we can see a type T of L as the class of terms which have that type, and we require that t and $val(t)$ have the same types, i.e. $t \in T$ implies $val(t) \in T$. To be specific, these results will apply to the *type systems* defined in Allen [1].

representation and sequents

A basic concept for all that follows is the notion that a type of the language L *represents* a mathematical class S defined on the terms of L .

Definition 1 *Given a relation t reps s between terms t and members of a class of objects S , we say that a type T of the language L represents class S (via reps) iff*

- (i) for any term t and any $s, s' \in S$, if t reps s' then s is s' .
- (ii) for any $s \in S$ there is a term t such that t reps s .
- (iii) for any terms t, t' . $t = t'$ in T iff there is an $s \in S$ such that t reps s and t' reps s .

The first two conditions are general requirements for calling something a representation relation. The third is specific to the conventional use of a type as a system of notations for objects of some class.

This definition is interpreted constructively which means that there are computable maps, rep : from objects of S to terms such that $rep(t)$ reps t and $unrep$, a partial function from objects of S to terms such that $unrep(rep(t)) = t$.

In the case of the class of terms itself, there is a *natural representation* relation that can be designed into the language. Details of this are given elsewhere [1, 8, 18]. In short, let the class of terms be the least class containing variables and operation names, and such that if t_i are terms and op an *operator name*, then $op(\bar{v}_1.t_1; \dots, \bar{v}_n.t_n)$ is a term where the \bar{v}_i are lists of variables. We say that t_i are the *subterm*; the \bar{v}_i are *binding variables* with scope t_i and they bind the variables occurring in t_i . Operator names are included among the terms in anticipation of the need for a term to represent each operator name.

Let us suppose that there are types in L that represent variables and operator names, call them Var and Op . (In this abstract we ignore any finer structure here, in the full paper we discuss *operator families* and their use in representing variables as well.) Then the *natural representation* of a term $op(\bar{v}_1.t_1; \dots, \bar{v}_n.t_n)$ is a pair $\langle OP, (\langle \bar{V}_1, T_1 \rangle, \dots, \langle \bar{V}_n, T_n \rangle) \rangle$ where OP represents op , \bar{V}_i represents \bar{v}_i and T_i represents t_i . Using this we can show that if L admits recursive types, then there is a type, call it $Term$, which represents term.

$$Term \stackrel{\text{def}}{=} \text{rec } t. Op | Var | (Op \times ((Var \text{ list} \times t) \text{ list})).$$

The class of sequents we shall consider consists of all objects of the form $H \gg T$, where H is a list of hypotheses of the form $v_i : T_i$ where v_i is a variable and T_i is a term, and T is a term. There is a natural representation of this class that is an extension of the representation for terms, namely,

$$\text{Sequent} \stackrel{\text{def}}{=} (Var \times Term) \text{ list} \times Term.$$

representing proofs

The informal concept of proof that we start with is called a *primitive proof*, it is essentially a finite tree of sequents justified by rules or taken as premises. Here is an example:

$$\begin{array}{ll} \vdash \forall A, B : Prop. (A \vee B \Rightarrow B \vee A) & \text{by intro} \\ A, B : Prop \vdash (A \vee B \Rightarrow B \vee A) & \text{by intro} \\ A, B : Prop, A \vee B \vdash B \vee A & \text{by elim } A \vee B \end{array}$$

$$\begin{array}{l} A, B : Prop, A \vdash B \vee A \\ \text{by intro right} \end{array}$$

$$\begin{array}{l} A, B : Prop, B \vdash B \vee A \\ \text{by intro left.} \end{array}$$

A primitive rule is basically a function from a sequent s_o to a list of sequents s_1, \dots, s_n such that if s_1, \dots, s_n are true, then s_o is true. We say that a primitive rule justifies s from l provided the rule applied to s yields the sequents at the roots of the subproofs in the list l . A premise of a proof is an improved sequent at a leaf.

It is easy to see that in a language L with recursive types, the class of primitive proofs can be represented by a type, namely:

$$\begin{aligned} \text{PrimProofTree} &\stackrel{\text{def}}{=} \text{rec } p. \text{Sequent} \times (\{1\} | \text{PrimRule}) \times p \text{ list} \\ \text{IsPrimProof}(P) &\stackrel{\text{def}}{=} \text{rec-ind}(P; h, \langle s, j, l \rangle. \text{map-list}(h; l) \times \text{case}; j \text{ of} \\ &\quad \text{inl}(1) \rightarrow l = \text{nil} \in \text{Sequent list;} \\ &\quad \text{inr}(p) \rightarrow \text{PrimJustifies}(s; p; \text{roots}(l))) \\ \text{PrimProof} &\stackrel{\text{def}}{=} \{ p : \text{PrimProofTree} \mid \text{IsPrimProof}(p) \}, \end{aligned}$$

where PrimRule is a type containing representatives of the primitive rules, and PrimJustifies($g; j; s$) is a predicate that represents the primitive justification relation, that is, it is inhabited exactly when the sequent represented by g is justified from the list of sequents represented by s using the primitive rule represented by j .

tactic trees

One way to think of proofs containing tactics is to allow a third kind of justification. It could be an expression in the metalanguage which builds a primitive proof.¹ We would say that a tactic, tac , justifies a sequent s provided that tac evaluates to a proof whose goal is s and whose premises are the roots of the subproofs. However, this notion of proof uses two languages, L and ML. One way to achieve a unification is to use the computational nature of L and express tactics in L itself, exploiting the internal representation of terms of proofs. The tactic will be a term that represents a proof. This represented proof must have the goal as its root and the sungoal as its premise.

reflected proofs

In building a `tactic_tree`, there are different ways to check that a tactic produces the right thing. One way is to run it, another is to type check it, and a third is to prove that it has the right type. A proof (or type checking which is a special case) would save the expense of building the proof tree; we discuss this point in section III. The proof might be critical to understanding the tactic, and indeed we may want to extract the tactic from a formal proof. So we ask whether such proofs of tactic correctness could be part of this proof type. That is, can we extend the justification to include

$$\begin{aligned} \text{justification} = & \text{leaf} + \text{prim_rule} + \{ t : \text{term} \mid \text{unreps}(t) \in \text{proof} \} \\ & + \text{reflection_rule} \end{aligned}$$

The `reflection_rule` should require that we build a representation of the class of proofs whose goal is a representation of the sequent we are trying to prove, say s . That is, we want to require

$$\exists p : \text{Proof}. \text{goal}(p) = \text{reps}(s)$$

where Proof is intended to be an L_type which represents the class of proof which we are trying to define from outside of L. Proof is an internal version of the very type we are now attempting to design. We already know a great deal about this type, and we could write down a formal definition of most of it except for the clause we are examining now.

¹It is significant that we use an expression here because we want to be able to analyze the object intensionally.

Before writing the clause, let us note that one understanding of this reflection proceeds in levels. We first reflect those proofs which do not use any reflection in them, say $proof_0$. Then having grasped that type, and defined it internally, we understand a new level of proof, say $proof_1$. We can now reflect these proofs to obtain $proof_2$ and so forth. The reflection rule will be indexed by an integer i which provides this level. We will give a non-hierarchical definition below.

Notice that the reflected proof may be incomplete, so it will have subgoals. These subgoals are the goals of the subproofs in the list l that prove them. So the requirement on the subgoals of a proof step to prove s by reflection is that the first subgoal has this special form:

$$\begin{aligned} \exists p : Proof . goal(p) = rep(s) \\ \& premises(p) = \text{“roots of the remaining subproofs of } l\text{”} \& \text{“all uses of reflection in } p \text{ are of level } j < i.\text{”} \end{aligned}$$

The name of the reflection rule will have an integer to denote the level and a term t to represent the subgoals generated by the rule application.

So we say

$$\begin{aligned} \text{justification} = \quad & \text{leaf} + \text{primitive_rule} + \{ t : term \mid unrep(t) \in proof \} \\ & + (N \times term) \end{aligned}$$

where if j is a reflection rule, that is $\langle i, t \rangle$, then

the goal of the first subproof of l , call it P_0 , has form $\exists p : Proof . root(p) = rep(s) \& premises(p) = goals(rest(l)) \& reflection_level(p) < i$.

We can see this more clearly if we depict the relationship between s , j and l in this fourth clause of j .

$$\begin{array}{ccc} s & < i, t > & \\ & s_1 \quad \dots \quad s_n & \\ \exists p : PF . root(p) = rep(s) & P_1 & P_n \\ \& \text{premise (p) =} & \\ (s_1, \dots, s_n) & & \\ \& reflection_level(p) < i & \\ P_0 & & \end{array}$$

and where t represents the list (s_1, \dots, s_n) of goals of the subproofs after the first one.

Proof is a type in L which we can express except for the clause that mentions Proof itself. To deal with this self-reference we will take a fixed point. To facilitate that, we replace the entire first subgoal of the reflection rule, the one mentioning Proof, by a piece of syntax, the expression PF.

$$\begin{aligned}
\text{ProofTree} &\stackrel{\text{def}}{=} \text{rec } p. \text{Sequent} \times \\
&\quad (\{1\} | \text{PrimRule} | \{t : \text{Term} \mid \text{unreps}(t) \in p\} | (\text{int} \times \text{Term})) \times p \text{ list} \\
\text{IsProof}(P) &\stackrel{\text{def}}{=} \text{rec-ind}(P; h, \langle s, j, l \rangle. \text{map-list}(h; l) \times \text{case}; j \text{ of} \\
&\quad \text{in1}(1) \rightarrow l = \text{nil} \in \text{Sequent list}; \\
&\quad \text{in2}(p) \rightarrow \text{PrimJustifies}(s; p; \text{roots}(l)); \\
&\quad \text{in3}(r) \rightarrow s = \text{root}(\text{unreps}(r)) \in \text{Sequent} \times \\
&\quad \quad l = \text{frontier}(\text{unreps}(r)) \in \text{Sequent list}; \\
&\quad \text{in4}(\langle i, t \rangle) \rightarrow \\
&\quad \quad \text{cons}(\langle \text{nil}, \text{PFof}(s; t; i; PF) \rangle; \text{unreps}(t)) = l \in \text{Sequent list}) \\
\text{Proof} &\stackrel{\text{def}}{=} \{p : \text{PrimProofTree} \mid \text{IsPrimProof}(p)\}.
\end{aligned}$$

Let us be more precise about proofs and proof representation. In the reflection rule used in proofs as described earlier, a term *Proof* was used in the reflection subgoal. The intention was that it should be a term which is a type representing this very class of proofs. Here we shall construct such a term directly as a syntactic fixed point.

Parameterize the notion of proof by a term to be used where *Proof* was used above; call this family of classes *proof_R*. Our aim is to find a term *R* which is a type representing the class *proof_R* (thereafter using *proof_R* as our proof class). Now, consider the definition of the term *Proof* given above. It can be parameterized by a term analogously to the parameterization of the class of proofs, namely, replace *PF* by another term *r*; let us call this family of terms *Proof_r*: *Proof_r* represents the class *proof_R* whenever *r* represents the term *R*.

Consider the family of terms *Proof_r*; it should be clear that we can find a term *q* such that whenever *a* represents term *r*, then the application *q(a)* represents the term *Proof_r*. There is a term *AP* such that *AP(a)(b)* represents the application *c(d)* whenever *a* and *b* represent the terms *c* and *d*. Also, the standard representation function on terms can be represented by a term *Rep* such that *Rep(a)* represents the standard representation of *b* if *a* represents *b*.

Now for a familiar construction. Let *B* be the term $\lambda u. q(AP(u)(Rep(u)))$. Let *b* be the standard representation of term *B*. Then *B(b)* has the same value as *q(AP(u)(Rep(u)))*, which represents *Proof_{B(b)}*, hence, so does *B(b)*. Letting the term *Proof* be *B(b)*, and letting the class *proof* be *proof_{Proof}*,

Theorem 1 *Proof represents the class proof.*

Another approach to getting a proof type which contains a representation of itself is to exploit the library. The library can contain specifications for how to expand defined terms to primitive terms. Truth of a sequent is then parameterized by libraries to mean that the sequent is simply true after all terms of the sequent have been expanded using that library. Representation is parameterized by libraries so that a term represents whatever the term it expands to represents. Because representation will be relevant to some of the rules, the class of proofs will become parameterized by libraries. Let *proof_Λ* be this family of classes.

Now, what we shall do is let the term *PF* stand as a fixed definable non-primitive in the definition of the term *Proof* above.

Theorem 2 *With respect to any library Λ, Proof represents proof_Λ.*

Theorem 3 *With respect to a library Λ where PF expands to the standard representation of the term $Proof$, PF represents $proof_\Lambda$.*

The following definition and theorems are to be considered under both the above interpretations of *proof* and *truth*; viz., *proof* as defined for theorem 1 with simple truth, and *proof* and *truth* implicitly parameterized by libraries as in theorem 2.

Definition 2 *A proof is valid iff the goal sequent is true whenever the premise sequents are.*

Theorem 4 *All proofs are valid.*

The proof of this theorem is by induction on the level of reflection and then by induction on the structure of the inductively defined class *proof*.

Theorem 5 *Without the level restriction, the class *proof* is not valid.*

Here we can adapt Löb's theorem [5] to prove a contradiction.

Theorem 6 *Every proof can be reduced to a primitive proof.*

This metatheorem could prove quite useful because it allows us to keep all proofs at level 0 if we wish.

Remark: One might want to take Theorem 7 as a reducibility axiom for proofs. But the fact is that adding it as an axiom makes it false!

We could use this reflection mechanism to increase the power of the logic, e.g. by adding rules asserting the consistency of the logic [12]. We have chosen to provide a logically conservative extension with considerable practical benefits which we discuss next.

3 Applications

theorem proving

In systems like Nuprl [9, 2, 19], tactics are used to build significant parts of a proof tree. Sometimes these tactics are costly to run because they are building a large piece of data; this often happens in circumstances in which it is easy to discover before hand whether or not the tactic will succeed. For example, there is a type checking tactic, called `get_type`, which will prove all sequents (recall section 2) of the form $H \gg T \text{ in } Type$, for example $A : Type, B : Type \gg A \times B \text{ in } Type$, as long as the type T is of restricted form, say first order. In the reflected system we can express this as a metatheorem, assuming that the relation `First-order` and `Well-formedness` are defined.

Theorem 7 $\forall s : \text{sequent} . \quad \text{First-order}(S) \Rightarrow \text{Well-formedness}(s) \Rightarrow$
 $\exists p : \text{Proof} . \text{goal}(p) = s \ \& \ \text{complete}(p).$

Now in the course of any argument in which the well-formedness goal $H \gg T \text{ in } Type$ appears, we can prove it by reflection. The cost of checking that a type is first order and recognizing a well-formedness goal will often be vastly simpler than proving the theorem; checking will be linear in size of T whereas building the proof is quadratic. The reflection mechanism will enable users of systems like Nuprl to provide efficient type checking for those subclasses of terms for which a good algorithm is discovered, e.g. for ordinary logic.

The Nuprl evaluator calls all functions by name. However, when we know that all the functions used in a term are strict, we can, instead, use call by value, and thus evaluate the term more efficiently. There are syntactic subclasses of the terms which have members that use only strict functions. The simply typed λ -calculus and Gödel's system **T** are two such subclasses. Now, in a system in which the type `Term` reflects the terms, and in which the evaluator is reflected by the operator `Val`, it will be possible to define the functions

$\text{IsSimplyTyped} : \text{Term} \rightarrow U_1,$

and

$\text{ByValueEvaluate} : \text{Term} \rightarrow \text{Term}.$

Then we can prove the theorem

$\forall t : \text{Term}. \text{IsSimplyTyped}(t) \Rightarrow \text{ByValueEvaluate}(t) = \text{Val}(t) \in \text{Term},$

and use this theorem to justify the use of a call by value evaluator when it is appropriate to do so

type theory

In a system like Nuprl there are theorems stated for particular universes but hold in all of them, e.g. the axiom of choice

$$\forall A : U_i . \forall P : A \rightarrow A \rightarrow U_i . \quad \forall x : A . \exists y : A . P(x)(y) \Rightarrow \\ \exists f : A \rightarrow A . \forall x : A . P(x)(f(x))$$

is true at all universe levels U_i . Using reflection we can build a function lift $\text{Proof} \rightarrow \text{Proof}$ which would take a theorem true at level i and provide a proof at level $i + 1$.

There are other applications of this sort which we discuss in the full paper.

mathematically documented systems extensions

The reflection mechanism offers a way to explore extensions and alterations to a system. We have seen above the possibility of defining new evaluators. In the full paper we will discuss how to detect expressions permitting destructive update. We have also seen how to incorporate type checking algorithms. For each of these concepts, it is possible to explore their effect on the system before completing the metamathematics needed to completely justify them. The evaluator for strict functions is a particular example.

References

- [1] S. F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proc. of Second Symp. on Logic in Comp. Sci.*, pages 215–224. IEEE, June 1987.
- [2] D. Basin. *Building Problem Solving Environments in Constructive Type Theory*. PhD thesis, Cornell University, 1990.
- [3] J. Batali. Computational introspection. Technical Report AIM-TR-701, MIT, 1983.
- [4] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Trans. Program. Lang. and Syst.*, 7(1):53–71, 1985.
- [5] G. Boolos and R. Jeffrey. *Computability and Logic, Third Edition*. Cambridge University Press, 1988.
- [6] 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. Academic Press, New York, 1981.

- [7] A. Bundy. A Broader Interpretation of Logic in Logic Programming. In *Proc. 5th Symp. on Logic Programming*, page ??, 1988.
- [8] R. L. Constable. Assigning meaning to proofs: a semantic basis for problem solving environments. *Constructive Methods of Computing Science*, NATO ASI Series, Vol. F55, pages 63–91, 1989.
- [9] R. L. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.
- [10] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.
- [11] M. Davis and J. T. Schwartz. Metamathematical extensibility for theorem verifiers and proof checkers. *Comp. Math. with Applications*, 5:217–230, 1979.
- [12] S. Feferman. Formal theories for transfinite iterations of generalized inductive definitions and some subsystems of analysis. In *Proc. Conf. Intuitionism and Proof Theory*, pages 303–326, North-Holland, Buffalo, NY, 1970.
- [13] J. Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge Tracts in Computer Science, Vol. 7. Cambridge University Press, 1989.
- [14] M. Gordon. HOL: A machine oriented formalization of higher order logic. Technical Report 68, Cambridge University, 1985.
- [15] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, Lecture Notes in Computer Science, Vol. 78. Springer-Verlag, NY, 1979.
- [16] T. G. Griffin. *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems*. PhD thesis, Cornell University, 1988.
- [17] S. Hayashi and H. Nakano. *PX: A Computational Logic*. MIT Press, Cambridge, MA, 1988.
- [18] D. Howe. Equality in lazy computation systems. In *Proc. of Fourth Symp. on Logic in Comp. Sci.*, pages 198–203. IEEE Computer Society, June 1989.
- [19] D. J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988.
- [20] D. J. Howe. Computational metatheory in Nuprl. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 310, pages 238–257. Springer-Verlag, New York, 1988.
- [21] T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Proc. of the First Symp. on Logic in Comp. Sci.*, pages 237–248. IEEE, 1986.
- [22] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. North-Holland, Amsterdam, 1982.
- [23] J. McCarthy. A basis for a mathematical theory of computation. *Comput. Program. and Formal Syst.*, pages 33–70, 1963. Amsterdam:North-Holland.

- [24] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symp. on Logic Programming*. ACM, 1987.
- [25] F. Pfenning. Elf: a language for logic definition and verified metaprogramming. In *Proc. of Fourth Symp. on Logic in Comp. Sci.*, pages 313–321, 1989.
- [26] B. Smith. Reflection and semantics in lisp. *Principles of Programming Languages*, pages 23–35, 1984.
- [27] C. Smorynski. *Self-Reference and Modal Logic*. Springer-Verlag, NY, 1985.
- [28] A. Troelstra. *Metamathematical Investigation of Intuitionistic Mathematics*, Lecture Notes in Mathematics, Vol. 344. Springer-Verlag, 1973.
- [29] R. Weyrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.