MAX-PLANCK-INSTITUT FÜR INFORMATIK

Metalogical Frameworks

David A. Basin
Robert L. Constable

MPI–I–92–205 January 1992

INFORMATIK

Im Stadtwald
D 66123 Saarbrücken
Germany

Authors' Addresses

David Basin, Max-Planck-Institut für Informatik Im Stadtwald, D-6600 Saarbrücken, Germany
basin@mpi-sb.mpg.de
Robert L. Constable, Department of Computer Science, Cornell University, Ithaca NY, USA.
rc@cs.cornell.edu

Publication Notes

A version of this paper will appear as a chapter in a book on logical frameworks, edited by Gerard Huet and Gordon Plotkin, to be published by Cambridge University Press in 1992.

Acknowledgements

We gratefully acknowledge discussions with Nax Mendler, Alan Smaill, and Sean Matthews. Frank Pfenning provided helpful comments on a draft of this paper. The work this report is based on took place at the University of Edinburgh where Basin was supported by the the North Atlantic Treaty Organization under a grant awarded in 1990 and SERC grant GR/E/44589. Constable was supported by an SERC visiting fellowship and a Guggenheim fellowship.

Abstract

In computer science we speak of *implementing* a logic; this is done in a programming language, such as Lisp, called here the *implementation language*. We also reason about the logic, as in understanding how to search for proofs; these arguments are expressed in the *metalanguage* and conducted in the *metalogic* of the *object language* being implemented. We also reason about the implementation itself, say to know it is correct; this is done in a *programming logic*. How do all these logics relate? This paper considers that question and more.

We show that by taking the view that the metalogic is primary, these other parts are related in standard ways. The metalogic should be suitably rich so that the object logic can be presented as an abstract data type, and it must be suitably computational (or constructive) so that an instance of that type is an implementation. The data type abstractly encodes all that is relevant for metareasoning, i.e., not only the term constructing functions but also the principles for reasoning about terms and computing with them.

Our work can also be seen as an approach to the task of finding a generic way to present logics and their implementations, which is for example the goal of the Edinburgh Logical Frameworks (ELF) effort. This approach extends well beyond proof-construction and includes computational metatheory as well.

1 Introduction

1.1 Role of Logical Frameworks and Formalized Metamathematics

At one time logic seemed to be a *finished* subject, and computers, like radio telescopes and linear accelerators, seemed to be the tools of *big science*. Now computers are ubiquitous and their software systems have brought logics to life. We see limitless expansion in both domains with computers destined to play a part in nearly every aspect of life and the logics of their systems needed to bring order and sense to this activity and provide a basis for realizing ever bolder dreams. This is a new role for logic and with it come new tasks and problems; this paper is about some of them, especially those that fall to computer scientists to address. Let us be more specific about this new role for logic and put these tasks in context, starting with the most familiar.

The first generation of software systems were brought under control with the development of automata theory, formal languages, and programming logics. Modern compilers are built with tools and techniques based on a deep understanding of parsing and compilation, and the methods of rigorous program development make it possible to achieve high levels of reliability for a range of specifiable programs.

Now the field seeks to do more and to carry the techniques forward into more imaginative systems which have formal logics as components. For instance there will be programming languages with type systems that are equivalent to logics, and the type checkers will be little theorem provers (maybe even big ones). There will be formal specification languages and program verifiers both based on a formal logic of some kind. There will be systems that support program synthesis and hardware synthesis. There will be more experimental systems such as programmer's assistants. In all of these systems formal logic is playing its new role. Moreover, there will be many different logics and many different implementations of each. So as in the case of compiler construction, there will be an industry for building and modifying the implemented logics.

Thus far, there are few tools designed specifically to support the activity of implementing logics. So systems are built from scratch using tools intended for other purposes. What is wanting is a framework for designing these logics and supplying the generic tools. This is one of the main goals of research in *logical frameworks*.

Another way to approach the topic of this paper is to consider the activity of generating *formal mathematics*, as pioneered by the project of Automath [11]. This has been taken up with considerable energy in computer science because it is seen as an integral part of some of the new kinds of systems mentioned above and because it is related to concerns in AI and programming. One of the subjects that is most promising to formalize is metamathematics, in part because the results of that work feed back directly to the task of generating formal theorems. In the context of formalizing metamathematics, we are confronted with many of the same questions facing the designer of a logical framework. This paper considers why this is so. Moreover we claim that the vantage point of formalized metamathematics is a good way to look at the task of logical frameworks, and we advocate its primacy, hence the title of the paper.

Scientists actually doing formal mathematics have come to understand the importance of metareasoning from experience. They quickly saw that nearly all informal mathematics is a mixture of object theory and metatheory. The work of Howe [27], comes to our minds in this connection. Also the style of automating reasoning via tactics involves programming in a *metalanguage*, and if one applies the methodology of programming systems based on the proofs-as-programs principle, such as Nuprl [15] and the Calculus of Constructions (CoC) [17], then many of these tactics should be extracted from metatheorems. A great deal of work has been done in this direction at Cornell [30, 26, 3]. Also even if the proofs-as-programs principle is not underlying the programming style, it has been widely noticed that substantial economies in proof generation are possible if tactics are replaced by the metatheorems that they implement.

The key component in an implemented logic is usually the *inference engine*, and our starting point is the observation that the systems will need ways to modify and progressively enhance it. Our response to the situation is to conjecture that the problems of understanding, specifying and modifying the inference engine are best dealt with in the framework of a *formalized metalogic*. It will be important to reason about the object logic (the one being implemented) in a very expressive metalogic.

1.2 Conceptual Issues

Logicians have been concerned for years with treating their subject more abstractly and generally. The elementary formal systems of Smullyan were an attempt to characterize the deductive machinery of *Principia Mathematica*, and lately Feferman has proposed more usable formalisms [22, 21] such as FS_0. There are abstract semantic characterizations of logic [5, 6] and the theorem of Lindström in abstract model theory characterizing first-order logics. There are other efforts along these lines such as [2].

The work of logicians deals with many of the issues central to the problem of logical frameworks such as what is a logic and what are the different ways to present them. Their concern for the most general setting for major results such as Gödel's theorems or Löb's theorem is directly germane to the task of formalizing metamathematics. But some of the issues faced in defining a logical framework and formalizing mathematics on a machine are new. For example, how should bound variables be represented? How can a sequent based proof economize on storage? What is the best way to realize computational content? What is the essential difference between sequent style proofs and natural deduction style, and does it apply to all logics?

Inevitably discoveries about logical frameworks will contribute to a conceptual understanding of logic. From the vantage point of this paper we see that the efforts to formalize metamathematics will contribute as well. This might not be so obvious *a priori*.

1.3 Results

Beyond advocating the primacy of metalogic this paper contributes some specific technical results. We propose a particular way to specify logics, namely using types that one might call *higher-order abstract data types* or ADT's for this paper. We show that an implementation of a logic can be seen then as an instance of the abstract data type. There are many interesting connections between these types and other approaches to data abstraction in type theory and programming languages.

We also look at *constructive metatheory in action* and illustrate by example why a framework for automating logic will want access to the results of formalized metamathematics. This practical need is remarkably congruent to the philosophical stance adopted by the first metamathematicians who used finitist or constructivist metatheories. This historical accident means that there is a deep literature of informal implicitly computational metatheory to draw on as well the explicitly computational results of modern computer scientists such as unification, resolution, term rewriting, and the like, and moreover that formalizing these results makes them clearer and more general, thus fulfilling the promise of formalized mathematics. Indeed given the deep literature in proof theory, there is much to be done in this vein to bring it to life and make it available to applied logicians. But also new kinds of metamathematical result are needed in reasoning about implemented logics, and we mention some of them as well.

We show that by taking a rich constructive metalogic as the basis for a logical framework, the context becomes simpler and the relationship between the concerns of logical frameworks and those of the formal metamathematics stands out. Finally, rather than choosing a specific metalogic which is adequate, such as Nuprl or CoC, we talk about the requirements for such a theory.

2 Data Abstraction in Type Theory

In this section, we provide a brief account of data abstraction within type theory. Our notion of data abstraction has similarities to two standard techniques used to define data within type theories; these are the techniques of defining data-types within minimal type theories such as CoC, and the definition of ADTs as Σ-types within predicative type theories.

Within type theories such as the CoC standard data-types (e.g., pairing, lists, etc.) are encoded by inductive definitions. These data-types are specified by:

- naming a set (type) that members will inhabit;
- giving function constants (and their types) for constructing members of the set;

$$
\begin{array}{l}
LIST \hat{=} A\!:\!Type \rightarrow \\
\quad List\!:\!Type \\
\quad \#\ Nil\!:\!List \\
\quad \#\ Cons : A \rightarrow List \rightarrow List \\
\quad \#\ Lind\!: P\!:\!(List \rightarrow Type) \rightarrow l\!:\!List \rightarrow P(Nil) \\
\qquad \rightarrow (h\!:\!A \rightarrow t\!:\!List \rightarrow P(t) \rightarrow P(Cons(h,t))) \rightarrow P(l) \\
\quad \#\ P\!:\!(List \rightarrow Type) \rightarrow l\!:\!List \rightarrow g\!:\!P(Nil) \rightarrow i\!:\!(h\!:\!A \rightarrow t\!:\!List \rightarrow P(t) \rightarrow P(Cons(h,t))) \rightarrow \\
\qquad\qquad Lind(P, Nil, g, i) = g \\
\qquad\qquad \wedge\ \forall h\!:\!A. \forall t\!:\!List. Lind(P, Cons(h,t), g, i) = i(h, t, Lind(P, t, g, i))
\end{array}
$$

$$
\begin{array}{l}
MSET \hat{=} A\!:\!Type \rightarrow \\
\quad MSet\!:\!Type \\
\quad \#\ Empty\!:\!MSet \\
\quad \#\ Add : A \rightarrow MSet \rightarrow MSet \\
\quad \#\ \forall a,b\!:\!A. \forall s\!:\!MSet.\ Add(a, Add(b,s)) = Add(b, Add(a,s)) \\
\quad \#\ Mind\!: P\!:\!(MSet \rightarrow Type) \rightarrow l\!:\!MSet \rightarrow P(Empty) \\
\qquad \rightarrow \{i\!:\!(a\!:\!A \rightarrow s\!:\!MSet \rightarrow P(s) \rightarrow P(Add(a,s)))\ | \\
\qquad \forall a,b\!:\!A. \forall s\!:\!MSet. \forall p\!:\!P(s). i(a, Add(b,s), i(b,s,p)) = i(b, Add(a,s), i(a,s,p))\} \\
\qquad \rightarrow P(l) \\
\quad \#\ P\!:\!(MSet \rightarrow Type) \rightarrow l\!:\!MSet \rightarrow g\!:\!P(Empty) \\
\qquad \rightarrow i\!:\!\{i\!:\!(a\!:\!A \rightarrow s\!:\!MSet \rightarrow P(s) \rightarrow P(Add(a,s)))\ | \\
\qquad\quad \forall a,b\!:\!A. \forall s\!:\!MSet. \forall p\!:\!P(s). i(a, Add(b,s), i(b,s,p)) = i(b, Add(a,s), i(a,s,p))\} \rightarrow \\
\qquad\qquad Mind(P, Empty, g, i) = g \\
\qquad\qquad \wedge\ \forall a\!:\!A. \forall s\!:\!MSet. Mind(P, Add(a,s), g, i) = i(a, s, Mind(P, s, g, i))
\end{array}
$$

Figure 1: List and Multi-set ADT

- providing rules for reasoning about type members and functions defined over them.

These declarations can be seen as capturing the rules for type formation, introduction and elimination (right/left rules), and computation. In theorem provers for CoC such as *LEGO*[12] these declarations are made within a *context*, and one reasons within the scope of these declarations.

When a type theory is enriched with Σ-types, a second approach is possible whereby the declarations of the ADT are packaged together with a (iterated) Σ-type; theorems are then proved within the context of these types. This has the conceptual advantage of unifying the data-type definition into a single type and the practical advantage of allowing parameterization of one ADT by another. It can also be seen as a natural generalization of signatures in ML, Extended ML [44], and other languages implementing data-abstraction using dependent types and is also related to the ideas of Bauer and his group at Munich on nonfree algebraic types [9]. This is the approach we shall take. One complication is that such modularization requires predicative quantification and hence a type hierarchy. For simplicity, we shall use the first universe of types (which we call $Type$) as the type of ADT parameters and set carriers.

Figure 1 contains two examples of ADTs: parameterized lists and finite multi-sets (sometimes called "bags"). Members of the LIST type are functions: when applied to a parameter type A, they return a tuple in which the first projection, the *carrier* of the ADT, is the type of lists over A whose members are built from the functions inhabiting the Nil and $Cons$ projections. The multi-set ADT is similar except for the defined carrier equality and the elimination and computation rules which are modified to respect this equality (more will be said on this below). Some notation should be explained. First $x\!:\!A\#B$ denotes a Σ-type built from the types A and B. Free occurrences of x become bound in B. Sometimes, to emphasize its logical significance, we abbreviate this to $\exists x\!:\!A.\, B$, or conjunction (when x is not free in B) $A \wedge B$, or even pairing $A \times B$. Similarly $x\!:\!A \rightarrow B$ represents the Π-type built from A and B and may be alternatively displayed as a $\forall x\!:\!A.\, B$ or $A \Rightarrow B$. As syntactic sugar, we will sometimes write multiple function applications in an "un-curried" style. E.g., writing $Cons(h,t)$ instead of $Cons(h)(t)$. To axiomatize equalities we make use of the equality (or I-type) of Martin-Löf type theory; equality is a ternary type-constructor ($t_1 = t_2\ in\ T$) that has a member precisely when t_1 and t_2 are equal members of T. We shall omit the third argument (the type

T) since it can usually be determined from the context. We also make use of the set type $\{x:A|B\}$; in this type, x is bound in B and its members are a in A when $B[a/x]$ is also inhabited. The proof rules for this type may be found in [15]. Finally, $\hat{=}$ is our notation for definitional equality.

The difference between these two ADTs is their specification of member equality. In an implementation of lists, members must be equal when they are built in identical ways from constructors. Moreover, when the induction combinator is known to be unique (which it is when the metalogic contains identity types), all implementing structures will be equivalent up to isomorphism. If we are working within a type theory such as Nuprl's which has a type constructor *list* with term constructors *nil* and "." (for cons) and an induction combinator *list_ind*, then we can trivially build a member of the list ADT.

$$\begin{array}{l}\lambda A.\, \langle A\ list, \langle nil, \langle \lambda a.\, \lambda t.\, (a.t), \\ \quad \langle \lambda P.\, \lambda l.\, \lambda g.\, \lambda i.\, list\_ind(l; g; h, l, v.i(h, l, v)), \\ \quad \langle \lambda P.\, \lambda l.\, \lambda g.\, \lambda i.\, \langle axiom, \lambda h.\, \lambda t.\, axiom\rangle\rangle\rangle\rangle\rangle\end{array}$$

Other implementations are possible; for example, lists as iterated pairs, fringes of trees, and so forth. For multi-sets, any implementation of the MSET ADT must equate not only those members built in the same way, but also those members which only differ in the order in which their elements were added. Specifying carrier types for ADTs that have axiomatized equalities requires a quotient type constructor such as Nuprl's, or extensions to the underlying type theory in the spirit of those suggested by Backhouse [4]. For example, using the quotient type we can implement multi-sets as lists where the quotienting equality is equivalence under permutation.

One complication of packaging ADT declarations as Σ-types is that we must be able to open them and access the components of their members. To this end, projection functions must be created and named, but these may be automatically generated in a straightforward way. In subsequent sections we shall assume that these projection functions have been defined and are named, when possible, by the binding variable associated with that component of the Σ-type. For example, let L be member of $LIST$, then for T a type, $L(T)$ is a tuple with $List(L(T))$ naming the first projection, $Nil(L(T))$ naming the second, and so forth. To simplify presentation in the remainder of the paper we will often make radical abbreviations in presenting projections from ADTs, and terms built from the constructor functions. For example, when L and T can be determined from context we will write terms like $List(L(T))$ and $Nil(L(T))$ simply as $List$ and Nil. Moreover we will use standard abbreviations in terms built from these constructors; for example we might write $Cons(L(T))(t_1, Cons(L(T))(t_2, Nil(L(T))))$ as $Cons(t_1, Cons(t_2, Nil))$, or even $[t_1, t_2]$.

The most significant distinction between our ADTs and traditional ADTs is that we axiomatize induction (elimination) rules and computation rules. The addition of these rules is not difficult within either impredicative or predicative type theories and may be automatically generated from the constructor declarations. In [40], Paulin-Mohring details how induction and computation principles can be automatically generated from declarations of constructor functions and their types and arities within the Calculus of Constructions; The INRIA implementation of CoC contains such a facility. In [4], Backhouse gives a similar account for predicative type theories (e.g., Martin-Löf's) and he also explains how these rules can incorporate quotiented equality relations like the kind needed for multi-sets. For brevity of presentation, we will not explicitly give the induction and computation rules associated with defined ADTs. *But these should be understood as being part of all ADTs defined.*

As an example, within the context of the list ADT we may define the functions "head" and "tails" as follows.

$$\begin{array}{rcl} hd(l) & \hat{=} & Lind(\lambda x.\, List)(l)(Nil)(\lambda h.\, \lambda t.\, \lambda v.\, h) \\ tl(l) & \hat{=} & Lind(\lambda x.\, List)(l)(Nil)(\lambda h.\, \lambda t.\, \lambda v.\, t) \end{array}$$

From these definitions, using the *Lind* induction principle it is straightforward to prove theorems such as

$$\vdash \forall A\!:\!Type.\, \forall L\!:\!LIST.\, \forall l\!:\!List.\, l \neq Nil \Rightarrow Cons(hd(l), tl(l)) = l.$$

There several subtleties that arise in the definition and application of functions over the ADT carriers. The first concerns the use of equality axioms in ADTs like the multi-set. Any defined functions must respect

$$\frac{}{\Gamma, A \vdash A}\,Axiom \qquad \frac{}{\Gamma, f \vdash G}\,FalseE$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,AndI \qquad \frac{\Gamma, A, B \vdash G}{\Gamma, A \wedge B \vdash G}\,AndE$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,OrIL \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,OrIR \qquad \frac{\Gamma, A \vdash G \quad \Gamma, B \vdash G}{\Gamma, A \vee B \vdash G}\,OrE$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\,ImpI \qquad \frac{\Gamma, A \Rightarrow B \vdash A \quad \Gamma, B \vdash G}{\Gamma, A \Rightarrow B \vdash G}\,ImpE$$

Figure 2: LJ Inference Rules

this equality and, as a result, many functions over lists, such as head and tails, have no analog for multi-sets since they do not respect the multi-set equality. This is sensible; since multi-sets are unordered, they have no "first" and "rest". Functions respecting this equality can still be defined. For example, we can define multi-set membership provided we are careful (e.g., using set types) that the definition does not provide a way of computing the "position" of the element in the multi-set. In future sections, we shall make use of defined functions on both lists and multi-sets such as member deletion (which we denote by "$-$") and membership ("$\in$"). We shall also denote the empty list (multi-set) by "$\{\}$" and *cons* (*add*) by "$+$".

Another subtlety concerns equality reasoning. Note that the computation rules allow us to reason about the conversion behavior of defined functions; we may compute with them, but only in an indirect sense. That is, the computation axioms do not augment the framework logic with reduction rules that can directly be computed with (in the sense of, say, the direct computation rules of Nuprl). Instead, one computes by using equality reasoning to establish that a defined function is equal to some normal form, e.g. $hd([1,2,3]) = 1$. This kind of computation via rewriting can be automated and made fairly efficient. In [27], Howe gives an example of how this kind of equality simplification can be automated by rewrite tactics similar to Paulson's conversions [41] and moreover how this can be accomplished (using derived inference rules) without direct construction of the corresponding equality proof. It would be interesting to study an alternative approach to computational equality in which direct computation rules are expressed in the ADT, perhaps in a type theory that contains a primitive for Howe's lazy congruence relation [28].

3 Specifying Provers With Abstract Types

Textbook presentations of logics typically begin with an account of logical syntax, i.e., syntactic entities such as terms, formulas, and proofs. Typical is Gallier [23], which presents the formulas of the propositional calculus as those in the inductive closure of a collection of propositional functions. As this is a freely generated set, he observes that we can define functions on formula by recursion and prove properties of formula by induction.

What this and most other presentations have in common is that they define logical syntax as sets built from functions having certain properties. Equally significant, they leave implementation details unspecified. It should not matter whether formulas are represented by ASCII strings or trees built from cons cells; all that matters is their extensional properties. This kind of abstraction and desire for representation independence between the specification of a logic and its implementation suggests that they stand in the same relation as that between an abstract data-type or signature (in the parlance of ML modules [38]) and its implementing structure. Note, in particular, that the induction and recursion requirements are perfectly captured by our induction and computation rules.

In this section, we illustrate this approach to specification by formalizing a theorem prover for a simple

$$
\begin{array}{l}
WFF \doteq \\
\qquad Wff\!:\!Type \\
\quad \#\ Var\!:\!Atom \rightarrow Wff \\
\quad \#\ False\!:\!Wff \\
\quad \#\ And : Wff \times Wff \rightarrow Wff \\
\quad \#\ Or : Wff \times Wff \rightarrow Wff \\
\quad \#\ Imp : Wff \times Wff \rightarrow Wff
\end{array}
$$

Figure 3: Propositional Formula ADT

logic: a sequent calculus presentation of an intuitionistic propositional calculus. This system, LJ, is taken directly from Dyckhoff [19] and is due to Gentzen [24]. Figure 2 presents the rules of this logic. We have chosen LJ for two reasons. First, as it is simple it admits an exceptionally clean exposition. So the reader won't feel cheated, in Appendix A we indicate how the term and proof types can be generalized in a straightforward way to the more complex predicate calculus. But we have a second reason, and that is a simple variant of this logic has certain structural properties that we will use to illustrate metatheoretic extensibility via proof-transformations in Section 5.

3.1 Well Formed Formula

Well formed formula (Wffs) of LJ are those terms freely generated from a collection of variables and propositional constructors. We shall assume that the term constructors are fixed, although this restriction is not strictly necessary. The Wff and proof data-types could be parameterized over sets of Wff and proof constructors and this can lead to a kind of theory-polymorphic metatheory. We do not explore this possibility though.

To specify variables, we will take as given in the framework logic a type *Atom* which is a collection of names that may be associated with variables.[1] The decision not to represent variables using variables in the framework logic is deliberate and will be discussed in Section 6 where we compare this to the alternative, which is used in frameworks like ELF. The formalization of this as an ADT is given in Figure 3. Recall that this ADT, like all our ADTs, also comes equipped with the appropriate induction and computation principles.

3.2 LJ Inference Rules and Proofs

We have considerable flexibility in how we present proofs as an ADT. In this section, we present a simple proof type for LJ and indicate alternatives that differ in their assumptions about the structure of proofs. Alternatives formalizations are significant as different types capture different models (implementations) of theorem proving and support different kinds of metatheoretic reasoning.

We begin with a simple encoding that captures a minimal amount of proof structure; essentially, the relationship between sequents that comprise proofs. It will have some similarities to ELF-style encodings since both explain the meaning of logical constants in the encoded logic through the declarations of constants in the framework logic. However unlike ELF encodings, where each proposition has its own type of its proof-objects we define a single type of proofs which we can equip with induction and computation principles.

To represent sequent calculus inference rules we must first define a data-type of sequents. In the system we formalize, a sequent is a pair where the first component represents hypotheses and the second the conclusion. Furthermore, hypotheses are unordered and repetition is allowed; this eliminates the need for structural rules.

[1]This may be a base type of strings in the framework logic (e.g., the type of strings *Atom* in Nuprl's type theory), or alternatively defined as a type of elements whose equality relation is decidable. We take this second approach in defining function names in the appendix.

Hence we model hypotheses using multi-sets of Wffs. The second component is simply a Wff.

$$\begin{aligned}&\mathit{SEQUENT} \hat{=} W: \mathit{WFF} \to M: \mathit{MSET} \to \\ &\quad \mathit{Sequent}: \mathit{Type} \ \# \ \ \mathit{MkSeq}: \mathit{MSet}(M(\mathit{Wff}(W))) \times \mathit{Wff}(W) \to \mathit{Sequent}\end{aligned}$$

Note that due to parameterization, any implementation of sequents depends on how both the Wff and multi-set ADT parameters are instantiated, and of course, the function inhabiting the sequent signature. Also note that by analysis of the component ADTs we may determine what principles may be employed to reason about and compute with sequents. For example, we may split sequents into their components, perform Wff induction on the conclusion, perform multi-set induction on the hypothesis set, and so forth. For s a member of the carrier of this type, we will denote its first and second projections by $Hyps(s)$ and $Concl(s)$ respectively.

The LJ proof rules (Figure 2) specify a relationship between tuples of sequents (the premises of the rule) and a sequent (the conclusion). Proof rules have limited applicability and we express constraints and side conditions using Π-types and set-types in the framework logic. The use of set types here is not strictly necessary (one could achieve similar specifications using additional axioms to capture these constraints) however they do provide an economy of expression as well as a way of tagging computationally uninteresting parts of the specification. For example, if LJ is the type of proofs, and for p in LJ, $Seq(p)$ represents the sequent proved, then we can formalize $AndI$ in the following way.

$$\begin{aligned}\mathit{AndI}:\ & p_1 : LJ \to p_2 : \{p : LJ | Hyps(Seq(p)) = Hyps(Seq(p_1))\} \\ & \to \{p : LJ | Hyps(Seq(p)) = Hyps(Seq(p_1)) \\ & \quad \wedge Concl(seq(p)) = And(Concl(Seq(p_1)), Concl(Seq(p_2)))\}\end{aligned}$$

This rule takes two members of the proof-type, p_1 and p_2 where the sequent proved in p_2 has the same hypotheses as the sequent proved in p_1, and returns a member of the proof-type with the same hypotheses and a conclusion that is the conjunction of the conclusions of p_1 and p_2. Other rules are presented in Figure 4.

Within the context of this proof type we may formulate the type of provable Wffs as follows.

$$Thm(w) \hat{=} \{p : LJ | Seq(p) = MkSeq(Nil, w)\}$$

Similar in spirit to the ELF, the inhabitants of the carrier LJ encode proofs. For example, if A is the propositional variable $Var(W)(\text{“A”})$, and similarly for B, then $ImpI(A, ImpI(B, axiom(MkSeq([A, B], A)))$ is a member of $Thm(Imp(A, ImpI(B, A)))$.

The LJ proof type specifies a bottom-up style of proof-construction where one works from theorems (under assumptions) to theorems (eventually without assumptions). Of course no orientation is specified by the informal presentation of these proof-rules in Figure 2, and we have the freedom to formalize alternative proof types. Different types will capture different models of theorem proving and different metatheories.

To illustrate, let us briefly sketch an alternative formulation, an ADT for a (slightly novel) kind of top-down proofs. The carrier of the ADT will have as members proofs constructed by refinement of goals. Refinement proceeds by naming a goal and applying a proof rule; the result is 0 or more subgoals. Goals will not be organized as a tree or stack, but rather as an unordered set (so identical unproven subgoals will be collapsed). Each goal in the set will be a *marked* sequent, which is a sequent paired with a status indicator of whether it has been refined (Ref) or is still unrefined ($UnRef$). A proof begins with an injection of a Wff into the proof type (with a function we call $Goal$) and ends when all sequents have been refined. Figure 5 contains a partial definition of this proof type. Here $MarkedSequents$ is the carrier of an ADT representing a set of sequents with status. This type has a constructor $Mark$ that given a member of the Sequent type (built with $MkSeq$) and a status indicator returns a marked sequent. The marked sequent ADT also has defined operators for member addition and deletion. The function $IsConjunct$ is defined by recursion on Wffs and is true when its argument is a conjunction. Similarly $LeftConjunct$ and $RightConjunct$ return the left and right conjuncts of a Wff that represents a conjunction.

Formalizing different models of theorem proving is interesting in its own right; it can bring into focus *how* the logic will be used in theorem proving. Assumptions can be made explicit such as whether proofs

$$
\begin{array}{l}
LJ\_PROOF \doteq W:WFF \to M:MSET \to S:SEQUENT \to \\
\quad LJ:Type \\
\# \ Seq : LJ \to Sequent \\
\# \ Axiom : s_1 : \{s:Sequent | Concl(s) \in Hyps(s)\} \to \{p:LJ | Seq(p) = s_1\} \\
\# \ FalseE : s_1 : \{s:Sequent | False \in Hyps(s)\} \to \{p:LJ | Seq(p) = s_1\} \\
\# \ AndI : p_1 : LJ \to p_2 : \{p:LJ | Hyps(Seq(p)) = Hyps(Seq(p_1))\} \\
\quad \to \{p:LJ | Hyps(Seq(p)) = Hyps(Seq(p_1)) \\
\qquad \wedge Concl(Seq(p)) = And(Concl(Seq(p_1)), Concl(Seq(p_2)))\} \\
\# \ AndE : w_1 : Wff \to w_2 : Wff \\
\quad \to p_1 : \{p:LJ | w_1 \in Hyps(Seq(p)) \wedge w_2 \in Hyps(Seq(p))\} \\
\quad \to \{p:LJ | Hyps(Seq(p)) = Hyps(Seq(p_1)) + And(w_1, w_2) - w_1 - w_2 \\
\qquad \wedge Concl(Seq(p)) = Concl(Seq(p_1))\} \\
\# \ OrIL : w_1 : Wff \to p_1 : LJ \\
\quad \to \{p:LJ | Hyps(Seq(p)) = Hyps(Seq(p_1)) \wedge Concl(Seq(p)) = OR(Concl(Seq(p_1)), w_1)\} \\
\# \ OrR : w_1 : Wff \to p_1 : LJ \\
\quad \to \{p:LJ | Hyps(Seq(p)) = Hyps(Seq(p_1)) \wedge Concl(Seq(p)) = OR(w_1, Concl(Seq(p_1)))\} \\
\# \ OrE : w_1 : Wff \to w_2 : Wff \to p_1 : \{p:LJ | w_1 \in Hyps(Seq(p))\} \\
\quad \to p_2 : \{p:LJ | w_2 \in Hyps(Seq(p)) \wedge \ Hyps(Seq(p_1)) - w_1 = Hyps(Seq(p)) - w_2 \\
\qquad \wedge Concl(Seq(p_1)) = Concl(Seq(p))\} \\
\quad \to \{p:LJ | Hyps(Seq(p)) = Hyps(Seq(p_1)) - w_1 + Or(w_1, w_2) \\
\qquad \wedge Concl(Seq(p)) = Concl(Seq(p_1))\} \\
\# \ ImpI : w_1 : Wff \to p_1 : \{p:LJ | w_1 \in Hyps(Seq(p))\} \\
\quad \to \{p:LJ | Hyps(Seq(p)) = Hyps(Seq(p_1)) - w_1 \wedge Concl(Seq(p)) = Imp(w_1, Concl(Seq(p_1)))\} \\
\# \ ImpE : w_1 : Wff \to w_2 : Wff \\
\quad \to p_1 : \{p:LJ | Imp(w_1, w_2) \in Hyps(Seq(p)) \wedge w_1 = Concl(Seq(p))\} \\
\quad \to p_2 : \{p:LJ | Hyps(Seq(p_1)) - Imp(w_1, w_2) + w_2 = Hyps(Seq(p))\} \\
\quad \to \{p:LJ | Hyps(Seq(p)) = Hyps(Seq(p_1))) \wedge Concl(Seq(p)) = Concl(Seq(p_2))\}
\end{array}
$$

Figure 4: LJ ADT

are constructed top-down, bottom-up, or in no particular order, whether they are constructed sequentially or in parallel, whether proof are represented by trees, DAGS, stacks, and so forth. Metatheoretically, as we alter the proof type, we change what concepts we can express and what properties of encoded proofs we can prove in our framework logic. For example, not all elements of the top-down proof type represent theorems, only those in which all sequents have been marked as refined. Hence, we can reason about incomplete proofs and routines (e.g., tactics) that extend and possibly complete these proofs. This is not possible with our LJ proof type. Our belief is that, in the broad sense of the word, the "metatheory" of a logic can go far beyond simply a theory of derivable rules and the more of a logic that we formalize the better we can reason about it.

4 Derived Rules

In the previous section we demonstrated that given our encoding of proof terms we can formulate a type of theorems in a straightforward way. All members of the LJ proof type are theorems (under assumption), and members of Top are theorems when all sequents are marked as refined. In both cases, we can construct proofs for the encoded logic by demonstrating that these theorem types are inhabited in the framework logic. This is similar to the conventional uses of type theory as a logical framework. In this section, we shall indicate how representing logics with ADTs provides a basis for metatheoretic reasoning that extends beyond simply capturing provability.

The most common method of augmenting an implementation of a logic with metatheoretic capabilities is to extend it with a metalanguage for writing *tactics*, which are programs that construct proofs. Many theorem proving systems, for example LCF, HOL, and Nuprl, take this approach. Each is equipped with a programming language (in all three cases ML), and an interface between this language and the proof engine. This interface typically consists of types for object-logic syntax (e.g., terms, sequents, proofs) and functions that manipulate and construct members of these types. This approach to metareasoning can be directly

$$
\begin{array}{l}
\quad Top : Type \\
\# \ SubGoals : \ Top \rightarrow MarkedSequents \\
\# \ Goal : \ w : Wff \\
\quad \rightarrow \{p : Top | SubGoals(p) = \{\} + Mark(MkSeq(Nil, w), UnRef)\} \\
\# \ Axiom : \ s_1 : \{s : Sequent | Concl(s) \in Hyps(s)\} \\
\quad \rightarrow p_1 : \{p : Top | Mark(s_1, UnRef) \in SubGoals(p)\} \\
\quad \rightarrow \{p : Top | Subgoals(p) = Subgoals(p_1) - Mark(s_1, UnRef) + Mark(s_1, Ref)\} \\
\# \ FalseE : \ s_1 : \{s : Sequent | False \in Hyps(s)\} \\
\quad \rightarrow p_1 : \{p : Top | Mark(s_1, UnRef) \in SubGoals(p)\} \\
\quad \rightarrow \{p : Top | Subgoals(p) = Subgoals(p_1) - Mark(s_1, UnRef) + Mark(s_1, Ref)\} \\
\# \ AndI : \ s_1 : \{s : Sequent | IsConjunct(Concl(s))\} \\
\quad \rightarrow p_1 : \{p : Top | Mark(s_1, UnRef) \in SubGoals(p)\} \\
\quad \rightarrow \{p : Top | Subgoals(p) = Subgoals(p_1) - Mark(s_1, UnRef) + Mark(s, Ref) \\
\qquad\quad + Mark(MkSeq(Hyps(S), LeftConjunct(Concl(s_1))), UnRef) \\
\qquad\quad + Mark(MkSeq(Hyps(S), RightConjunct(Concl(s_1)), UnRef)))\} \\
\ldots Other \ Proof \ Rules \ Similar
\end{array}
$$

Figure 5: Top Down "Unstructured" LJ ADT

used within our framework when the implementation of the framework logic supports a metalanguage. This capability is shared by other approaches to logical encoding.

But there is an alternative that appears to be possible when the following two conditions are met.

1. the framework logic is a programming logic,
2. logics have been encoded in a way that renders it possible to reason about the encodings and define functions over them.

When the first condition is satisfied, which it is when the framework logic is a constructive type theory, then the logic itself is expressive enough to serve as both its own metalanguage and metalogic. When the second condition is also satisfied, which it is when induction and computation axioms are provided for the encoded data-types, we can use this metalogic to reason about the logical encodings and functions defined over these encodings. Functions may be defined explicitly (using the appropriate axiomatized recursion combinators), or alternatively built implicitly by constructive proofs (e.g., the proofs-as-programs paradigm). These functions can implement general kinds of derived inference rules including term normalizers, decision procedures, and the like.

The approach to constructive metatheory we are proposing is very similar in spirit to the kind of metatheory developed in [26, 16] in which term rewriters, matchers, and normalizers, were implemented and formally reasoned about within the Nuprl type theory. The primary difference is that in the Cornell work, the metatheory was about a specific encoding of terms (e.g., Nuprl terms in [26]) and to be used in Nuprl theorem proving a connection had to be made between the encoded terms and Nuprl's terms. Under our approach, data-types such as terms and proofs are specified within the context and metatheorems are never used at the "object level" (e.g., we never work with implementations of the encoded logic) so there are no difficulties involving reflection or moving between syntactic entities and their encodings. Derived rules are simply theorems about properties of the relevant proof types and functions defined over these types and are used to construct and manipulate members of these types within the framework logic. (More will be said on these points in Section 6.)

In this section, we provide a simple example: a derived inference rule about the provability of a subclass of propositional Wffs. We take this example directly from Weyhrauch [46] where he states:

> If you have a propositional Wff whose only sentential connective is the equivalence sign, then the Wff is a theorem if each sentential symbol occurs an even number of times.

Weyhrauch stated this theorem for a logic in which the Wff type includes the "iff" connective (which we denote by the prefix binary relation *Iff*) and the proof system is classical. We can easily extend our Wff type (with this connective) and LJ proof type (with the rule of excluded middle) to encode this desired logic. Let $\mathcal{C}$ denote the context containing declarations of these types as well as definitions of the supporting ADTs. Furthermore, let *parity* be a function of type $\mathit{Wff} \to \{0,1\}$ that returns 1 exactly when its argument satisfies Weyhrauch's structural criteria. This can be defined using the Wff recursion combinator in a straightforward way.

Now, within this context, we may formalize Weyhrauch's theorem as

$$\mathcal{C} \vdash \forall w\colon \mathit{Wff}.\, \mathit{parity}(w) = 1 \Rightarrow \mathit{Thm}(w).$$

Let us sketch a constructive proof of this by induction on the complexity of the expression w. Given a Wff w such that $\mathit{parity}(w) = 1$, then w is a term built from *Iff* that must contain at least two occurrences of the same variable. Call this variable a; there are two cases. The first is $w = \mathit{Iff}(a,a)$, but then we are done as $\mathcal{C} \vdash \mathit{Thm}(\mathit{Iff}(a,a))$ is provable. In the second case, w must contain more than one *Iff* connective. Then, because *Iff* is associative and commutative, we can rearrange and reassociate the variables in w to "shuffle" together two occurrences of a. More formally, there must be Wff w' such that $\mathit{parity}(w') = 1$ and

$$\mathcal{C} \vdash \mathit{Thm}(\mathit{Iff}(\mathit{Iff}(a,a), w')) \Leftrightarrow \mathit{Thm}(w).$$

Furthermore, we can "cancel" the $\mathit{Iff}(a,a)$, that is

$$\mathcal{C} \vdash \mathit{Thm}(w') \Leftrightarrow \mathit{Thm}(\mathit{Iff}(\mathit{Iff}(a,a), w')).$$

Since $\mathit{parity}(w') = 1$ and w' is simpler (e.g., fewer variables) than w, we have $\mathit{Thm}(w')$ by the induction hypothesis. Hence, we can conclude $\mathit{Thm}(w)$.

The above proof sketch should be formalizable within our framework logic in a straightforward way. Formalizing the theorem statement requires that we can define the parity function by recursion over the structure of Wffs. This is not difficult and can for example be implemented by a function which makes several passes over its Wff argument: first to check the connective structure and second to gather up the variables at the leaves to verify that each occurs an even number of times. Formalizing the proof requires reasoning about Wffs (and the parity) function) using the Wff induction principle. In particular, we must be able to derive alternative induction schemas such as induction over well-founded measures on Wffs. In the above proof, this measure would be the number of variables in the formula. Complete induction over this well-order (although not all well-orders) is interderivable with our axiomatized structural induction principle for Wffs.

A proof of the parity theorem will (within a constructive framework logic) construct a function that itself constructs proofs. Specifically, let *par* be a function inhabiting the type of the parity theorem and let w be a member of *Wff* such that $\mathit{parity}(w) = 1$. The existence of *par* formally asserts that w is provable and we may use this assertion in other proofs. Moreover, should we actually require the proof of w, we may generate it by evaluating *par*.

5 Proof Theory

The implementation of the object theory lets one investigate whether a specific proposition is provable. The metatheory lets us investigate whether there is a general method of settling the provability of a class of propositions, and whether there is even a *good* method for doing this.

For the IPC, Intuitionistic Propositional Calculus, there are well-known algorithms to decide provability. The first is due to Gentzen, based on his Hauptsatz, but it is not a good method. Basically he suggests searching the space of all possible proofs up to a certain depth. We can consider this space to be an infinite tree whose nodes are sets of sequents and whose edges are labeled by the rule name used to replace a sequent, call it the goal, by a finite number of subgoal sequents. We are thinking then of generating proofs *top-down*, and it is natural to use a top-down proof type. This tree of all possible proofs will contain any proof that exists, and if we knew that any proof must occur within a certain depth, then we could just search to that

$$\frac{\Gamma, C \Rightarrow (D \Rightarrow B) \vdash G}{\Gamma, (C \wedge D) \Rightarrow B \vdash G}\, ImpE_1 \qquad \frac{\Gamma, C \Rightarrow B, D \Rightarrow B \vdash G}{\Gamma, (C \vee D) \Rightarrow B \vdash G}\, ImpE_2$$

$$\frac{\Gamma, D \Rightarrow B \vdash C \Rightarrow D \quad \Gamma, B \vdash G}{\Gamma, (C \Rightarrow D) \Rightarrow B \vdash G}\, ImpE_3 \qquad \frac{\Gamma \vdash G}{\Gamma, False \Rightarrow B \vdash G}\, ImpE_4$$

$$\frac{\Gamma, B, A \vdash G}{\Gamma, A \Rightarrow B, A \vdash G}\, ImpE_5 \qquad \textit{(for A atomic)}$$

Figure 6: LJT Rules Replacing $ImpE$

depth, which is a finite process. But indeed we do know from the Hauptsatz that we only need to search to a finite depth. Namely we know that if there is a proof, then there is one which uses only subformulas of the goal. Moreover, if in the search for a proof in the tree we encounter a node whose set of sequents appears earlier on the same path, then we can terminate the search along that path. Since only subformulas can appear in the sequents, there are only finitely many different nodes that can occur in this tree, so along any path, either we find a proof or repeat a node thus ending the search. Moreover we can precompute the depth at which this repetition must occur, so the argument is constructive.

It is natural to ask however, if there is a simpler search procedure analogous to the tableau procedure in the classical propositional calculus. The complication is that Gentzen's original system, LJ, uses the implication elimination rule

$$\frac{\Gamma, A \Rightarrow B \vdash A \quad \Gamma, B \vdash G}{\Gamma, A \Rightarrow B \vdash G}$$

and the subgoal sequent $\Gamma, A \Rightarrow B \vdash A$ might in fact be more complex than the goal sequent if A is more complex than G. So one cannot use a multiset ordering. Various methods have been proposed to circumvent this difficulty. One can compute the number of times the formula $A \Rightarrow B$ needs to be used for example. Roy Dyckhoff has discovered a method [19], and in his paper he surveys other ones and recommends the source [18]. His idea is to replace the implication elimination rule with the five others given in Figure 6; he calls the resulting system LJT. In LJT all rules generate simpler subgoal sequents so that termination of the proof search can be guaranteed via the multiset ordering. He proves:

A formula of IPC is provable in LJ iff it is provable in LJT.

Here is a metamathematical result which is very useful. It is the basis for a good decision procedure for the IPC. Indeed Dyckhoff has implemented such an algorithm for the system MacLogic (see [20]). Suppose then that in an implemented object logic based on LJ one is looking for a decision procedure for IPC formulas. If we restrict ourselves to derived rules based on LJ, we will not find Dyckhoff's method. But if we can prove metatheorems like his in the metalogic, then we can use them. So for instance, from a constructive proof of his theorem, we can build a decision procedure which takes a formula G, applies Dyckhoff's procedure to find an LJT proof or report that there is no proof. If there is no LJT proof, then we know there is no LJ proof either, so we report failure. If there is an LJT proof, then by the constructive nature of the proof of the metatheorem, we can transform it to an LJ proof as required.

Indeed Dyckhoff's result can be proved constructively. We outline such a proof below and suggest in a bit more detail how a formalization of it in type theory is used. First define the depth of a formula as follows:

$$depth(A) = depth(False) = 1 \text{ for } A \text{ atomic},$$

$$depth(A \vee B) = depth(A \Rightarrow B) = depth(A) + depth(B) + 1,$$

$$depth(A \wedge B) = depth(A) + depth(B) + 2.$$

This introduces an ordering on formulas which we extend to an ordering on multisets of formulas in the standard way. Now, we may prove two theorems:

Theorem 1 *If a formula G is provable in LJ, then it is provable in LJT.*

Theorem 2 *If a formula G is provable in LJT, then it is provable in LJ.*

We only need Theorem 2 to be proved constructively, but in fact constructive proofs of each theorem are just as clear and compact as classical proofs and provide more information, so we prove them this way. For Theorem 1 we follow Dyckhoff and prove a lemma about how to postpone uses of implication elimination. First a definition.

Definition 1 *Call a proof* awkward *iff the final step is an elimination on $A \Rightarrow B$ where A is atomic and does not occur as an hypothesis (i.e. on the left side of the sequent).*

Lemma 1 *If $\Delta \vdash G$ is provable in LJ, then it has a nonawkward proof.*

Proof: Let P be a proof of $\Delta \vdash G$, proceed by induction on the structure of P, i.e., by the induction rule provided by the proof data type. If it is not awkward we are done, so suppose it is, and the last step is

$$\dfrac{\Gamma, A \Rightarrow B \vdash A \quad \Gamma, B \vdash G}{\Gamma, A \Rightarrow B \vdash G}$$

where A is atomic and does not occur in Γ. Let P_A be the subproof of A from Γ and $A \Rightarrow B$. This must end with an elimination step since A is atomic and does not occur in Γ. We consider all (four) ways that this can happen, which are: False-elim, $\wedge$-elim, $\vee$-elim and $\Rightarrow$-elim. In each case we show how to build a nonawkward proof.

1. If the rule is False-elim, then $False$ occurs in Γ. So we can use False-elim to prove $\Delta \vdash G$ in the first place. Call this proof P', it is not awkward.

2. If the rule is $\wedge$-elim, say on $C \wedge D$, then the proof looks like

$$\dfrac{\dfrac{\Sigma, C, D, A \Rightarrow B \vdash A}{\Sigma, C \wedge D, A \Rightarrow B \vdash A} \quad \Sigma, C \wedge D, B \vdash G}{\Sigma, C \wedge D, A \Rightarrow B \vdash G.}$$

Change this to the nonawkward proof P'

$$\dfrac{\dfrac{\Sigma, C, D, A \Rightarrow B \vdash A \quad \Sigma, C, D, B \vdash G}{\Sigma, C, D, A \Rightarrow B \vdash G}}{\Sigma, C \wedge D, A \Rightarrow B \vdash G.}$$

We know that there is a proof of $\Sigma, C, D, B \vdash G$ because we can cut in the formula $C \wedge D$ and use the given proof of $\Sigma, C \wedge D, B \vdash G$ known by our assumption on the shape of P. Then we obtain the proof P' we need by cut elimination on this subproof.

3. If the rule is $\vee$-elim, say on $C \vee D$, then the proof looks like

$$\dfrac{\dfrac{\Sigma, C, A \Rightarrow B \vdash A \quad \Sigma, D, A \Rightarrow B \vdash A}{\Sigma, C \vee D, A \Rightarrow B \vdash A} \qquad \Sigma, C \vee D, B \vdash G}{\Sigma, C \vee D, A \Rightarrow B \vdash G}$$

This can be converted to

$$\dfrac{\dfrac{\Sigma, C, A \Rightarrow B \vdash A \quad \Sigma, C, B \vdash G}{\Sigma, C, A \Rightarrow B \vdash G} \qquad \dfrac{\Sigma, D, A \Rightarrow B \vdash A \quad \Sigma, D, B \vdash G}{\Sigma, D, A \Rightarrow B \vdash G}}{\Sigma, C \vee D, A \Rightarrow B \vdash G}$$

The sequent

$$\Sigma, C, B \vdash G$$

can be proved by just cutting in $C \vee D$ which requires proofs of $\Sigma, C, B, C \vee D \vdash G$ and $\Sigma, C, B \vdash C \vee D$. The first one can be proved by weakening to remove the hypothesis C and then using the proof of $\Sigma, D, B \vdash G$. Call this proof $\hat{P}$. A similar modification is used to produce say $\bar{P}$ which proves $\Sigma, D, B \vdash G$.

4. If the rule is $\Rightarrow$-elimination, say on $C \Rightarrow D$, then the proof looks like

$$\cfrac{\cfrac{\begin{array}{c}P_A\\ \Sigma, D, A \Rightarrow B \vdash A\end{array} \quad \begin{array}{c}P_C\\ \Sigma, C \Rightarrow D, A \Rightarrow B \vdash C\end{array}}{\Sigma, C \Rightarrow D, A \Rightarrow B \vdash A} \quad \begin{array}{c}P_B\\ \Sigma, C \Rightarrow D, B \vdash G\end{array}}{\Sigma, C \Rightarrow D, A \Rightarrow B \vdash G}$$

Since we have already transformed the subproof of $\Gamma, A \Rightarrow B \vdash A$ to be nonawkward, we know that in the proof of $\Sigma, C \Rightarrow D, A \Rightarrow B \vdash A$, C is either nonatomic or occurs in Σ. We now convert the given proof P so that the last step is by elimination on $C \Rightarrow D$. This step is not awkward. This produces the left subgoal of showing $\Sigma, C \Rightarrow D, A \Rightarrow B \vdash A$ and the right subgoal of showing $\Sigma, D, A \Rightarrow B \vdash G$. The left subgoal can be proved from one of the subproofs of P and the right one is proved by elimination on $A \Rightarrow B$. One of the subgoals here is proved using a subproof of P and the other one, $\Sigma, D, B \vdash G$ is proved by cutting in $C \Rightarrow D$ and applying cut elimination to the resulting proof. □

Now Theorem 1 is proved by induction on the multi-set ordering using the lemma, and Theorem 2 is proved by induction on the proof structure using cut followed by cut elimination. The straight forward proofs here are constructive.

Let $LJTProof(G)$ denote the LJT proofs of G, likewise for $LJProof(G)$. Then Dyckhoff's decision procedure can be derived from a proof of the following theorem. First let S be a list of IPC formulas, say H_1,...,H_n, G. We regard it as a sequent with goal G and the other formulas as hypotheses and write $LJTProof(S)$ for the set of its LJT proofs.

Theorem 3 *Given any set $\{S_1, \ldots, S_n\}$ where S_i is a sequent over IPC formulas, then for each S_i, $\exists p : LJTProof(S_i) \vee \neg\exists p : LJTProof(S_i)$.*

Corollary 1 *For all formulas G of IPC either there is an LJT proof of it or not.*

The corollary is proved by taking $\{G\}$ as the set of formulas in the theorem. The theorem is proved by the multi-set induction defined above. For each S_i we consider all possible rules r that can be applied, noting that each one decomposes S_i into $\{U_1, \ldots, U_k\}$ for which the induction hypothesis holds. If the U_j are all provable for some rule r, then S_i is. If not, then there is no rule for proving S_i, so there is no proof of it.

The constructive proof of Theorem 3 and its corollary provide a function from formulas into a disjoint union. If the result is in the left disjunct, then we can pull out a proof of the formula; if it is in the right disjunct, then we know there is no proof (and if we wanted to we could pull out the evidence for this conclusion). So Theorem 3 gives a decision procedure for provability and a proof generating procedure for true formulas. The constructive proofs of Theorem 2 and Theorem 1 provides functions $J : LJTProofs \Rightarrow LJProofs$ and $T : LJProofs \Rightarrow LJTProofs$. With these we can prove another corollary.

Corollary 2 *For all formulas G of IPC either there is an LJ proof of it or not.*

6 Comparison To Other Work

We conclude with a brief comparison to closely related work. First, we compare our framework logic with other frameworks, both in terms of the logic itself and how we encode logics within it. After, we discuss how our approach relates to other research in metatheory.

6.1 Framework Logics and Logical Encodings

Framework logics such as the ELF and Isabelle were designed with concerns different from metatheoretic reasoning and they are too weak to support it. Furthermore, as their concerns are different, logics are encoded within these frameworks in a way that does not support metatheoretic analysis. Let's address these points individually.

The logical basis of the framework we propose is a constructive type theory and, in this respect, it is most similar to the LF logic of the ELF system. However, LF was designed to be as weak as possible (e.g., one cannot abstract over members of *Type*) to make faithfulness and adequacy proofs easier. But this minimality only supports a very weak kind of metatheory, e.g., one can demonstrate that specific formulas have proofs. Both Taylor and Pollack, who have proved theorems about LF encodings (as reported in [45]), have had to formalize their proofs within the stronger Calculus of Constructions. Other framework logics, such as Isabelle and λ-Prolog, are based on higher-order intuitionistic logic and lack the propositions-as-types correspondence. Hence, demonstrations of provability do not construct proofs, and any kind of metatheoretic reasoning within the framework (none has been tried to our knowledge) would not construct functions over proofs. Furthermore, these logics do not provide a unified view of the relationship between the specification of a logic and its implementation.

Equally significant as differences in logics are differences in the way that they are used. Within the frameworks mentioned above one seeks to encode logics in a manner that *internalizes* parts of the structure of terms and proofs within the framework logic. By this we mean that syntactic notions and operations are subsumed by operations implemented in the framework logic. The alternative to this identification is to *externalize* these entities and operations by explicitly encoding syntax and functions over syntax.

The standard example of internalization is representing variable binding by λ-abstraction in the framework logic so that substitution can be implemented by β-reduction; supporting this internalization is one of principles behind the design of these framework logics. It is true that this has the advantage of liberating the logic encoder from concerns about substitution, bound variable renaming, and the like. But from a metalogical perspective, this choice seems inappropriate since it obscures the separation between the encoded logic and the framework logic; the terms in the framework logic which represent the terms and proofs of the encoded logic combine both meta and object level syntax. One can no longer view a logic's presentation as a specification of an implementation. More importantly, the presentation cannot easily be the subject of metatheoretic analysis because analyzing variables and operations on them requires reasoning about the framework logic itself. Consider for example term equality decision procedures, such as those presented in [7], where the term orderings used for term normalization may depend on the names of variables. Alternatively, consider decision procedures where the complexity of the procedure can depend on the number of variables or the number of time each variable occurs. For example, propositional 2-satisfiability is in PTIME, where as the general problem is NP-complete.

Internalization manifests itself in other ways as well. Consider for example, the difference between how we formulate inference rules in the LJ proof type (Section 3.2), and how these rules would be formulated in the ELF-style. In ELF, *AndI* would be formulated along the lines of

$$AndI : \forall A, B : \mathit{Wff}.\ \mathit{Tr}(A) \rightarrow \mathit{Tr}(B) \rightarrow \mathit{Tr}(And(A, B)).$$

This rule is simpler than ours, but that is because it effectively internalizes much of the structure of proofs; rather than formalizing and reasoning about an explicit representation of proofs, which requires explicitly formalizing sequents (in the case of the sequent calculus) or assumption dependencies and discharging (in the case of natural deduction) and the like, these concepts are instead captured by the meaning of implication in the framework logic. Again there are tradeoffs. This style of natural deduction presentation has advantages in that it allows rapid specification and prototyping of theorem provers. But again, it is unsuitable for our needs. First, it is not always possible to find an ELF-style encoding, especially for more exotic consequence relations. It should, we suspect, be straightforward to directly describe the inference rules and proofs of a very wide range of logics using ADTs. Second, unlike our rules, ELF-style proof-rules cannot be seen as specifying an implementation of the logic, except in an indirect way via an interpreter for proof construction in the framework logic. Finally, there is a significant difference when it comes to metareasoning. Since we specify a single inductively constructed proof type (as opposed to a family of proof types parameterized by Wffs), we can equip it with induction and computation rules which are necessary for most nontrivial kinds of metatheory.

With respect to encoding style, the use of the framework logic FS_0 (as proposed by Feferman in [21] and carried out by Matthews [34]) is closest to our approach. FS_0 is a theory of functions and classes of expressions embedded in second-order predicate calculus. The basic data-type is the S-expression and through this one defines by inductive definition the classes of formulas and proofs for desired logics. As with our approach, functions may be defined by primitive recursion (on S-expressions) and reasoned about by induction. FS_0 appears to be designed as a minimal logic for allowing such reasoning (it is a conservative extension of primitive recursive arithmetic). By necessity, one cannot mix the syntax of the encoded logic and the framework logic; all notions of syntax must be explicitly encoded using S-expressions. In practice, this does not appear to cause significant problems. In [35] an example is provided about how one can formalize and reason about bound variables in FS_0 in the context of a prenex normal form theorem for predicate calculus. The conclusion there, which we find comforting, is that this kind of formalization and reasoning is not as hard as is commonly assumed. It is worth noting that the FS_0 account could be given directly in type theories such as Nuprl using inductive types to define terms and formulas.

6.2 Categorical Logics

Category theory like type theory is a unifying general language. Computer scientists have also found it a suitable context for investigating the basic questions being addressed here, such as what is a logic, what is an implementation, and so forth [25, 36, 31]. Moreover category theory has been the language of choice for some of the pioneering work on how to combine and relate large structures such as theories [13, 14]. It is also the standard setting in which to frame the algebraic approach to data types [10].

Our use of ADT's to define logics shows a clear and simple connection to the study of algebraic data types in a general setting. The closest related work in categorical logic is Burstall and Goguen's abstract account of model theory as *institutions* [25] and Meseguer's abstract account of proof calculi and general logics [36]. Our approach has concentrated on proof theory, but the generality of type theory suggests ways of expressing many of the concepts from institutions. In relation to the categorical account of proofs, we can say that our approach is more concrete, for example our notion of an LJ proof is far more specific than Meseguer's *effective proof subcalculus*. We have not tried to say what a proof calculus would be in general.

We are not aware of work in categorical logic which is directed at generalizing or supporting metamathematics. But the formulation of logic in categorical terms may help us see how to generalize our account of metamathematics.

6.3 Metatheory

Our approach to formalized metareasoning bears closest resemblance to the research program on constructive metatheory carried out at Cornell. In [16], Constable and Howe demonstrate how one can synthesize tactics, decision procedures, and the like, through theorem proving in the Nuprl system. In [26], Howe describes a substantial implementation of this kind of approach in which theorems were proved about representations of Nuprl terms (e.g., that one could rewrite and normalize such terms). Moreover, Howe's development included a means of associating representations of Nuprl terms, with Nuprl terms themselves, so his verified tactics and procedures could be applied to construct Nuprl proofs.

Another line of research was that of Constable and Knoblock [30] in which they formalized the structure of Nuprl proofs within an extension of Nuprl which contained data types for Nuprl terms and proofs. So again one could prove theorems about proofs and construct tactics by theorem proving. They, in fact, defined a hierarchy of metalogics; recent work [3] has refined this technique, providing a single proof type within Nuprl that refers to itself and can be formally reasoned about within the Nuprl system.

The kind of metatheory that we propose in this paper is similar in spirit to the previous Cornell research, but differs in several significant respects. First, the Cornell work has been concerned with encoding the terms and proofs of a specific logic (e.g., Nuprl) with specific types, and developing metatheory over these types. Here we are suggesting that this approach generalizes and can be used for a wide variety of logics. Moreover, we propose that this can be done in an abstract way (using ADTs) that does not commit the logic encoder to choose a specific representation in the framework logic. Of course, the downside of this abstraction is that we cannot directly compute over our representations of terms and proofs as can be done with the more direct Nuprl encodings.

A second important difference is that the Cornell work has been concerned with using such encodings by means of some kind of reflective mechanism. That is, to use the results of their metatheory, a connection must be made between representations of Nuprl terms and proofs (and statements proved about them) and actual Nuprl syntax. Here we suggest that all theorem proving can be done at one level: the metalevel. Although our encodings can be seen as specifying an implementation of that logic, no implementation ever need be constructed; instead all reasoning (about provability, derived rules, etc.) can take place within the framework logic. Hence, reflection is not central to our work since there is only one theory, the framework theory.

It is of course possible to propose that the framework logic be reflective (indeed if one were using Nuprl it would be). The advantages of using reflection in any practical formalization is one of the themes of the Cornell work, but we are not considering these practical concerns here. Moreover, it is interesting to consider an ADT account of object theories which are reflexive. We have not carried this out, but it is a line of inquiry which is in the spirit of this paper. The significant point is that one might be able to reduce the amount of detail needed in any reflexive axiomatization by separating the specification from its many possible implementations.

A Appendix: Predicate Calculus

In this appendix, we give a very brief account of how terms, formulas, and proof rules that involve side conditions on variables may be formalized within the ADT setting. Our account closely follows [23].

We begin by defining the terms of the predicate calculus which are built from variables and function applications. Function names are specified by an ADT FNAME that has an associated arity function and a decision procedure for equality of names. 0-ary functions are constants.

$$\begin{array}{l} FNAME \mathrel{\hat{=}} \\ \quad\quad Fname : Type \\ \quad \#\ Arity : Fname \to Nat \\ \quad \#\ \forall x, y{:}Fname.\, x = y \vee x \neq y \end{array}$$

Note that we don't insist that members of $Fname$ are enumerable. This could be added if required in the metatheory. The ADT of terms may now be given.

$$\begin{array}{l} TERM \mathrel{\hat{=}} fn{:}FNAME \to \\ \quad\quad Term : Type \\ \quad \#\ Var : Atom \to Term \\ \quad \#\ Func : f{:}Fname(fn) \times NTuple(Arity(f), Term) \to Term \end{array}$$

$NTuple(n, T)$ is the type of n-tuples with members from the type T. An ADT of predicate names and predicate calculus wffs may be defined in an analogous way.

To define proof rules for terms of the predicate calculus we must formalize the concept of free variables, eigenvariables, and substitution. All of these may easily be defined by recursion on $Term$ and Wff using the induction principles these ADTs are equipped with. We use considerable syntactic sugar in the definitions that follow.

Let $SET(Atom)$ be the ADT of sets parameterized by the type of variables $Atom$ with carrier Set. Define FV_T on terms as follows.

$$\begin{array}{lcl} FV_T(x) & = & \{x\} \\ FV_T(f(t_1, ...t_n)) & = & FV_T(t_1) \cup ... \cup FV_T(t_n) \end{array}$$

where the arity of f is n and $\cup$ is the union operator in the ADT of sets. Similarly define FV on Wffs by the following equations.

$$\begin{array}{lcl} FV(p(t_1, ..., t_n)) & = & FV_T(t_1) \cup ... \cup FV_T(t_n) \\ FV(p_1 \to p_2)) & = & FV(p_1) \cup FV(p_2) \\ FV(\forall a.\, p_1) & = & FV(p_1) - \{a\} \end{array}$$

Substitution must be defined both on formulas and on terms. On terms, we define a substitution function which we shall display as $t_1[t_2/x]_T$.

$$\begin{array}{lcl} y[t/x]_T & = & \mathit{if}\ y = x\ \mathit{then}\ t\ \mathit{else}\ y \\ f(t_1, ..., t_n)[t/x]_T & = & f(t_1[t/x]_T, ..., t_n[t/x]_T) \end{array}$$

Substitution on formulas will be displayed as $f[t/x]$ and is as follows.

$$\begin{array}{lcl} P(t_1, ..., t_n)[t/x] & = & P(t_1[t/x]_T, ..., t_n[t/x]_T) \\ (p_1 \to p_2)[t/x] & = & p_1[t/x] \to p_2[t/x] \\ (\forall y.B)[t/x] & = & \mathit{if}\ x = y\ \mathit{then}\ \forall y.B\ \mathit{else}\ \forall y.(B[t/x]) \end{array}$$

We say that a term t is *free for* a variable x *in* a Wff p when when t can be substituted for occurrences of x in a capture avoiding way. We define the propositional function $FreeFor$ as follows.

$$\begin{array}{lcl} FreeFor(t, x, p(t_1, ..., t_n)) & = & TRUE \quad \text{(For } TRUE \text{ an non-empty type)} \\ FreeFor(t, x, p_1 \to p_2) & = & FreeFor(t, x, p_1) \wedge FreeFor(t, x, p_2) \\ FreeFor(t, x, \forall y.p) & = & x = y \vee x \neq y \wedge \neg(y \in FV(t)) \wedge FreeFor(t, x, p) \end{array}$$

In addition, to formalize proof rules we will need to know when a variable can be serve as an eigenvariable; that is, when it does not occur free among the hypotheses $\Gamma = [h_1, ..., h_n]$ of a sequent.

$$EigenVar(\Gamma, x) = \neg y \in (FV(h_1) \cup ... \cup FV(h_n)).$$

A proof type for the predicate calculus may now be constructed by simple additions to the type LJ_PROOF given in Figure 4. Specifically, parameterize this type over the newly defined Wff type instead of the type of propositional Wffs. The rules of propositional calculus hold in predicate calculus so they carry over directly. We must add rules for quantifier introduction and elimination such as the following.

$$\frac{\Gamma \vdash P}{\Gamma \vdash \forall x.P}\, AllI \qquad \frac{\Gamma, P[t/x] \vdash G}{\Gamma, \forall x.P \vdash G}\, AllE$$

Both rules have side-conditions. In the first, we insist that $EigenVar(\Gamma, x)$ holds. In the second, we must have that $FreeFor(t, x, P)$ holds. Adding these rules is straightforward. We extend our proof type with the two conjuncts given in Figure 7. There, the auxiliary function $IsAll$ is true when its argument is a universally quantified Wff. $AllVar$ and $AllBody$ decompose $\forall x.P$ into the binding variable x and the body P.

Note that there are alternative ways to formalize these rules. In particular $\forall$-introduction is often formulated to allow the user to choose the name of the eigenvariable. Another option is that in $\forall$-elimination, rather than restricting w_2 to instances where $FreeFor$ is true, one may appropriately rename bound variables so that the substitution is made in a capture avoiding way.

$\# \ AllI : x{:}Atom$
$\quad \to p_1 : \{p{:}LJ | EigenVar(Hyps(Seq(p)), x)\}$
$\quad \to \{p{:}LJ | Hyps(Seq(p)) = Hyps(Seq(p_1)) \wedge Concl(Seq(p)) = All(x, Concl(Seq(p_1)))\}$
$\# \ AllE : w_1 : Wff$
$\quad \to w_2 : \{w{:}Wff | IsAll(w) \wedge FreeFor(w_1, AllVar(w), AllBody(w))\}$
$\quad \to p_1 : \{p{:}LJ | AllBody(w_2[w_1/AllVar(w_2)]) \in Hyps(p_1)\}$
$\quad \to \{p{:}LJ | Hyps(Seq(p)) = Hyps(Seq(p_1) - AllBody(w_2[w_1/AllVar(w_2)]) + w_2$
$\quad\quad \wedge Concl(Seq(p)) = Concl(Seq(p_1))\}$

Figure 7: Quantifier Proof Rules

References

[1] Peter Aczel. Frege structures and the notions of proposition, truth and set. In *The Kleene Symposium*, pages 31 – 59, North Holland, 1980.

[2] Peter Aczel. Term Declaration Logic and Generalized Composita. In *Proceedings of the Symposium on Logic in Computer Science*. IEEE, Washington, DC, 1991.

[3] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In *Symposium on Logic in Computer Science*, Computer Society Press of the IEEE, 1990.

[4] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1(1):19–84, 1989.

[5] Jon Barwise. Axioms for abstract model theory. *Ann. Math. Logic*, 7:221-265, 1974.

[6] Jon Barwise and Solomon Feferman (editors). *Model-Theoretic Logics*. Springer-Verlag, NY, 1985.

[7] David A. Basin. Equality of terms containing associative-commutative functions and commutative binding operators is isomorphism complete. In *Proc. of 10th International Conference On Automated Deduction (CADE-10)*, pages 251 – 260, Springer-Verlag, Kaiserslautern, Germany, 1990.

[8] David Basin, Fausto Giunchiglia, and Paolo Traverso. Automating meta-theory creation and system extension. In *AI*IA-91 (Italian Association for Artificial Intelligence*, Palermo, Italy, 1991.

[9] F.L. Bauer, H. Wössner, H. Partsch, P. Pepper. *Algorithmische Sprachen*. Technical Report, Technische Universität München, 1977.

[10] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In *Automata, Languages and Programming. Lecture Notes in Computer Science, vol 81*, Springer-Verlag, NY, 1980, 76-90.

[11] N.G. de Bruijn. A survey of the project Automath. In *To H.B. Curry: Essays in combinatory logic, lambda calculus and formalism.* Academic Press, Reading, 1980, 579-606.

[12] Rod Burstall. *Computer Assisted Proof for Mathematics: an Introduction Using the LEGO Proof System.* Technical Report ECS-LFCS-91-132, University of Edinburgh, 1991.

[13] Rod M. Burstall and Joseph A. Goguen. Putting theories together to make specifications. In *5th IJCAI*, pages 1045 – 1058, Boston, Mass, 1977.

[14] Rod M. Burstall and Joseph A. Goguen. The semantics of clear, a specification language. In *Advanced Course on Abstract Software Specification*, LNCS 86, 1980.

[15] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[16] Robert L. Constable and Douglas J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In *A source book of formal approaches to A.I.*, North Holland, 1990.

[17] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 95–120, 1988.

[18] Albert G. Dragalin. *Mathematical Intuitionism – introduction to proof theory* Translations of mathematical monographs vol. 67. American Mathematical Society, Providence, RI, 1988.

[19] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. Technical Report, University of St Andrews, CS/91/5, 1991. Accepted for publication by JSL.

[20] Roy Dyckhoff, Neil Leslie, and Stephen Read. MALT St Andrews (MacLogic – a proof assistant for first-order logic. *Computerised Logic Teaching Bulletin*, 2(1):51–60, 1989.

[21] Solomon Feferman. Finitary inductively presented logics. In *Logic Colloquim '88*, North-Holland, 1988.

[22] Solomon Feferman. Inductively presented systems and the formalization of meta-mathematics. In D. van Dalen *et al*, editor, *Logic Colloquium '80*, pages 95–128, North-Holland, Amsterdam, 1980.

[23] Jean H. Gallier. *Logic for Computer Science.* Harper & Row, 1986.

[24] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen.* North-Holland, Amsterdam, 1969.

[25] Joseph Goguen and Rod Burstall. Institutions: Abstract Model Theory for Computer Science. Technical Report CSLI-85-30 Center for the Study of Language and Information, Stanford University, Palo Alto, CA, 1985.

[26] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, 1988.

[27] Douglas J. Howe. Computational metatheory in Nuprl. In *9th International Conference On Automated Deduction*, pages 238–257, Argonne, Illinois, 1988.

[28] Douglas J. Howe. Equality in lazy lambda calculi In *Proc. of the Fourth Annual Symp. on Logic in Computer Science*, IEEE.,1990.

[29] Jörg Hudelmaier. *A Decision Procedure for Intuitionistic Propositional Logic.* Technical Report 90-32, Universität München, 1990.

[30] Todd B. Knoblock and Robert L. Constable. Formalized metareasoning in type theory. In *Proc. of the First Annual Symp. on Logic in Computer Science*, IEEE., 1986.

[31] Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, Cambridge, 1986.

[32] F. William Lawvere. Adjointness in foundations. *Dialectica*, 23:281-296, 1969.

[33] David B. MacQueen. Using dependent types to express modular structure. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, ACM, SIGACT, SIGPLAN, 1986.

[34] Sean Matthews. *Meta-Level and Reflexive Extension in Mechanical Theorem Proving.* PhD thesis, University of Edinburgh, DAI, 1991. Expected.

[35] Sean Matthews, Alan Smaill, and David Basin. *Experience with FS_0 as a Framework Theory.* In *Proceedings of the Second Workshop on Logical Frameworks*, Edinburgh Scotland, 1991.

[36] Jose Meseguer. General Logics. In H.D. Ebbinghaus *et al.*(editors), *Proceedings of Logic Colloquium 87.* North-Holland, 1989.

[37] John C. Mitchell. Representation independence and data abstraction. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, ACM, SIGACT, SIGPLAN, 1986.

[38] John C. Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, 1988.

[39] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, ACM, SIGACT, SIGPLAN, 1985.

[40] Christine Paulin-Mohring. Inductive definitions in the calculus of constructions. In *The Calculus of Constructions, Documentation and users's guide*, Project Formel, 1989.

[41] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[42] Frank Pfenning. Logic programming in the lf logical framework. In *Logical Frameworks*, pages 149 – 181, Cambridge University Press, 1991.

[43] Don Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165-210,1988.

[44] Don Sannella and Andrzej Tarlecki. *Toward formal development of ML programs: foundations and methodology.* Technical Report 71, Laboratory for Foundations of Computer Science, 1989.

[45] Paul Taylor. *Using Constructions as a MetaLanguage.* Technical Report ECS-LFCS-88-70, University of Edinburgh, 1988.

[46] Richard W. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.