Journal of Automated Reasoning **22:** 171–221, 1999.
© 1999 *Kluwer Academic Publishers. Printed in the Netherlands.*

Metalogical Frameworks II: Developing a Reflected Decision Procedure

WILLIAM E. AITKEN[⋆]
Microsoft Research, One Microsoft Way, Redmond, WA 98052, U.S.A.

ROBERT L. CONSTABLE[⋆⋆]
Department of Computer Science, Cornell University, Ithaca, NY 14853, U.S.A.

JUDITH L. UNDERWOOD[‡]
Quality Systems and Software Ltd., 13 Atholl Crescent, Edinburgh EH3 8HA, United Kingdom

(Received: 30 March 1995)

Abstract. Proving theorems is a creative act demanding new combinations of ideas and on occasion new methods of argument. For this reason, theorem proving systems need to be extensible. The provers should also remain correct under extension, so there must be a secure mechanism for doing this. The *tactic-style provers* pioneered by Edinburgh LCF provide a very effective way to achieve secure extensions, but in such systems, all new methods must be reduced to tactics. This is a drawback because there are other useful proof generating tools such as *decision procedures*; these include, for example, algorithms which reduce a deduction problem, such as arithmetic provability, to a computation on graphs.

The Nuprl system pioneered the combination of fixed decision procedures with tactics, but the issue of securely adding new ones was not solved. In this paper we show how to safely include user-defined decision procedures in theorem provers. The idea is to prove properties of the procedure inside the prover's logic and then invoke a *reflection rule* to connect the procedure to the system. We also show that using a rich underlying logic permits an abstract account of the approach so that the results carry over to different implementations and other logics.

Key words: theorem proving, Nuprl, decision procedures, tactic-style provers.

1. Introduction

1.1. MOTIVATION

The technical problem we solve is related to questions that arise in the design and deployment of software systems that we call *problem solving environments* (PSE). These include computer aided design (CAD) tools, symbolic algebra sys-

⋆ Supported in part by ONR grant N00014-92-J-1764 and NSF grant CCR-9244739.
⋆⋆ Supported by ONR grant N00014-92-J-1764 and NSF grant CCR-9244739.
‡ Supported in part by the United States Army Research Office through the Army Center of Excellence for Symbolic Methods in Algorithmic Mathematics (ACSyAM), Mathematical Sciences Institute of Cornell University, Contract DAAL 03-91-C-0027. Also supported by EPSRC grant GR/H73103 at the University of Edinburgh.

tems [18, 36, 37] and theorem provers [8, 9, 14, 26] all of which are designed to help people solve a specific class of problems. They are typically advanced systems in that they reside at the top of a hierarchy of support tools such as programming languages, databases, and special editors; and they are intended to help solve challenging problems. Most of them contain critical components that must be correct, such as simplification routines, e.g. Pbs [1].

Since these systems interact with users at the margin of what they understand, there is a need to extend the problem solving mechanisms over time; new knowledge is added (say definitions and theorems) as well as new techniques (say new symbolic algorithms for algebra). We call these *open systems*.

In the case of theorem provers, but to some extent in all PSE's, extensions of the system require great care lest its reliability suffer. The earliest provers did not allow any user extension of their *methods*. Users could add only new knowledge. The Edinburgh LCF system pioneered the idea of tactics which allow new methods as well. Milner [25] said "I needed a medium by which I could communicate to the machine certain general procedures for reasoning that it would later invoke... I would not have to lead it through the elementary steps every time. I wanted to be able to give it larger and larger chunks of reasoning powers, built up from the smaller chunks."

The LCF tactic mechanism enabled users to securely add new proof methods so the design of the prover was no longer the exclusive province of the designers ("power to the people"). Many other provers have now adopted this approach [8, 9, 14, 26].

Experience from the 1970s and earlier also showed that decision procedures were very effective in building proofs. The Stanford Pascal Verifier, PL/CV [6], and EHDM [29] among others used such procedures. In particular, the PL/CV-arith [6], congruence closure [19], and sup-inf [30] are from this period, and all are actively used today. New decision procedures are always being studied, monotone closure [24], real-closed fields [20], etc.

Here are typical uses of `arith`.

$$x < x^2 \;\&\; x \neq 0 \Rightarrow 2 \leqslant x \vee x < 0 \;\textit{by}\; \texttt{arith}$$

$$\begin{aligned}&(x - y \leqslant y \;\&\; y \leqslant x + 1 \;\&\; x - 1 \leqslant z \;\&\; z \leqslant x + 1\\ &\;\&\; y - 1 \leqslant z \;\&\; z \leqslant y + 1) \Rightarrow\\ &(x = y) \vee (y = z) \vee (z = x) \;\textit{by}\; \texttt{arith}\end{aligned}$$

Nuprl was the first tactic-oriented prover to incorporate decision procedures. Even though the `arith` and congruence closure algorithms were proved correct, informally, adding these mechanisms is *dangerous*. They substantially increase the complexity of the prover and raise the burden of checking that the system is correct. Our experience makes us very reluctant to add more procedures in this *raw* way, and we are especially wary of procedures as complex as BKR [20]. We eschew these *raw complex decision procedures*.

The collective wisdom of the theorem proving community has thus created a *design dilemma*: **How to securely add new decision procedures?** One approach has been proposed for HOL. Decision procedures would be used freely when the system executes in *unsafe mode*, then those steps previously solved by the procedure would later be reduced to primitive rules when the whole proof is done in *safe mode*. This is a secure method, but it would cost substantial CPU cycles.

Another approach would be to verify the decision procedure inside the system. The easiest way is to verify a purely "mathematical" algorithm and then recode it. But this violates the maxim that one should "execute exactly what was verified". One could attempt to verify the algorithm exactly as coded in the implementation language. This approach is being followed by E. Gunter for HOL90 [34, 22], where the implementation language is SML-NJ. This is a very large task which resembles the reflection work in Nuprl [2 – 4] on which this paper is based, but she must rely on the stability of a large programming language and must define a *substantially* more complex system.

A related technique involves constructing decision procedures in such a way that a full proof of any formula the procedure claims is true may optionally be constructed [28]. This requires a metatheory in which the verification of a tactic may be carried out. In our use of reflection, the metatheory and object theory are the same, and linked by the reflection principle (outlined below).

1.2. SUMMARY OF A SOLUTION

The solution to the design problem starts with the observation that tactic-style provers are just as effective for higher-order logics and general mathematical theories as they are for first-order ones. All of the tactic-style provers cited here [8, 9, 14, 26, 27] support versions of type theory, a rich foundational theory for mathematics [8, 7, 23]. These theories are expressly designed to support the formalization of large tracts of mathematics. As knowledge accumulates in this form, it becomes increasingly feasible to verify sophisticated algorithms such as decision procedures. Bledsoe [5] pointed the way already in the seventies.

Another critical point about these rich logics is that they allow an abstract development of algorithms, which is valuable for making results exportable.

The second key step in solving the design problem is to observe that for type theories with an identifiable computation system, as with *constructive type theories*, it is possible to write tactics and decision procedures directly in the logic itself. This is true for Nuprl, whose computation system is a functional programming language in the spirit of ML [15], which is the language of choice for writing tactics.

A critical part of writing tactics inside a type theory is to have a model of the logic inside the logic. Again this can be done in rich type theories and has been done for Nuprl.

Even with these pieces in place, we still do not have a solution. At this stage we can write and verify decision procedures in the logic, and they may apply to the logic. For example, in Nuprl we can define the syntax and proof rules of the Intuitionistic Propositional Calculus (IPC) abstractly. We can prove the decidability of provability of IPC in Nuprl and instantiate the result for the class of propositional Nuprl formulas, which form an instance of IPC. But *still* these decision procedures are beyond our reach; we need a way to *link* this result to the Nuprl logic itself.

The final step of our solution relies on the Nuprl *reflection rule*, a unique feature of the system [2–4]. This rule says that to prove a goal G under hypothesis H, it suffices to show that the reflected sequent, $\ulcorner H \vdash G \urcorner$, is provable. That is, *Reflection Principle:*

$$H \vdash G \text{ if } \exists P : \mathit{Proof} \text{ such that } \; P \text{ proves } \ulcorner H \vdash G \urcorner.$$

Now we can apply the verified decision procedure to $H \vdash G$ as we show in detail (Section 7).

2. Type Theoretic Preliminaries

Basic Operators on Types

We will be defining concepts from logic using type theory and the concepts of type theory using logic. Our account assumes acquaintance with ordinary propositional logic and working knowledge of a mathematical vocabulary which includes the notions of *type* and *set*. Our type theory includes operators for dependent product and function types, disjoint unions, subset types defined by predicates, and type universes *Type* and *Prop*. In the next few paragraphs, we will explain these ideas somewhat informally for the reader unfamiliar with type theory.

The fundamental notion is that of a type, rather than a set. We use type theory because of its close connection with computation, as we see later. Thus we say that $\mathbb{N}$, the natural numbers, is a type. We speak of its *subsets* of the form $\{x : \mathbb{N} | P(x)\}$, where P is a propositional function. In general, for any type A and propositional function P on A, we speak of the subset of those x in A such that P is true, written $\{x : A \mid P(x)\}$.

Given two types A and B, we talk about their *product*, $A \times B$, and their *disjoint union*, $A + B$. We also consider the type of functions from A to B, written $A \rightarrow B$. These constructions are usually thought of as operations on types, but they make sense for sets as well. However, in the language of sets we usually consider other concepts first, such as the power set $\mathcal{P}(A)$, of all subsets of A and the union $A \cup B$. These are less common operations in type theory, and we do not need them in this paper. By limiting ourselves to operators such as product and disjoint union, we can maintain a direct correspondence with the notion of type in programming languages (where the product $A \times B$ is often generalized to n-tuples, called a *record type*, $A_1 \times A_2 \times \cdots \times A_n$).

Both the function and product types are generalized to include infinitary versions. So if B_x is a type for each x in A, i.e., a family of types, then

$x : A \to B_x$	is the type of all functions f such that $f(a) \in B_a$ for all $a \in A$ (sometimes denoted $\Pi x : A.B_x$), and
$x : A \times B_x$	is the type of all pairs $\langle a, b\rangle$ such that $b \in B_a$ for $a \in A$ (sometimes denoted $\Sigma x : A.B_x$).

Among the types are the large types *Type* and *Prop*. These denote the collection of (small) types and the collection of (small) propositions. More will be said as necessary, but for now it suffices to know that a propositional function on a type A is just an element of $A \to \mathit{Prop}$.

Algebraic Structures and Abstract Types

We will be very concerned with algebraic structures. An example is a monoid which is a type M along with an associative binary operation on M, say $f : M \times M \to M$, and an identity i in M. The triple $\langle M, f, i\rangle$ is usually considered a *monoid*. It is an element of the product type which is written

$$M : \mathit{Type} \times f : (M \times M \to M) \times i : M.$$

Sometimes we think of $M : \mathit{Type}$, $f : M \times M \to M$ and $i : M$ as the *signature* of the monoid.

In programming language terms, these algebraic structures are called *abstract data types*. The idea is that the carrier M is any type which supports the operators called for in the signature. They are abstract because we do not say how to "implement" the data type. So, one instance or implementation of a monoid is $\mathbb{N}$ with the plus operator, $+$, and the additive identity element, 0; i.e. $\langle \mathbb{N}, +, 0\rangle$ is a monoid. In general, we will also need to describe an equality on the abstract type. In an algebraic context this equality is implicit, but later in the paper we shall find that the carrier set's own equality may not be the one we wish to use. For this example we do not explore this further.

To conclude that $\langle \mathbb{N}, +, 0\rangle$ is really a monoid, we must actually show that $+$ is associative, and that 0 is an identity. More formally, if we define *assoc* so that

$$\mathit{assoc}(M, f) \text{ iff } \forall x, y, z : M(f(x, f(y, z)) = f(f(x, y), z)),$$

we must show that $\mathit{assoc}(\mathbb{N}, +)$ holds. Likewise, if we define *id* so that

$$\mathit{id}(M, f, i) \text{ iff } \forall x : M(f(x, i) = x = f(i, x)),$$

we must show that $\mathit{id}(\mathbb{N}, +, 0)$ holds.

If we include these propositions in the signature, we can give a complete definition of the abstract data type of monoids.

$$M : \mathit{Type} \times f : (M \times M \to M) \times i : M \times \mathit{assoc}(M, f) \times \mathit{id}(M, f, i).$$

An element of this type is a 5-tuple. It consists of a type A called the *carrier*, an operation g on A, an element a of A, some evidence that g is associative on A, say a formal proof *assoc_proof*(A, g), and some evidence *id_proof*(A, g, a) that a is an identity. Such a 5-tuple,

$$\langle A, g, a, \mathit{assoc\_proof}(A, g), \mathit{id\_proof}(A, g, a)\rangle,$$

is a *(full) monoid*.

3. Logical Preliminaries

The logic which we will eventually formalize within our type theory is intuitionistic propositional logic. The proof system we use a Gentzen sequent calculus. An informal explanation follows; see [13] for a fuller account.

Formulas

We adopt these standard logical operations on formulas P and Q.

notation	meaning
$\bot$	the false proposition
$(P \mathbin{\&} Q)$	P and Q
$(P \vee Q)$	P or Q (constructively)
$(P \Rightarrow Q)$	P implies Q (constructively)

The constructive understanding of the operations is in terms of provability. To say that a formula P is *true* is to say that it is *provable*, i.e., that we have a proof of it.

$(P \mathbin{\&} Q)$ is provable iff both P and Q are.
$(P \vee Q)$ is provable iff either P is provable or Q is.
$(P \Rightarrow Q)$ is provable iff there is a method of transforming
any proof of P to a proof of Q
$\bot$ is not provable.

Proofs

The notion of proof formalized here is based on the notion of a *sequent* introduced by Gentzen. A sequent is a pair consisting of a goal formula, G, and a multiset of formulas which are the hypotheses which can be used to deduce G using certain *rules* of proof. We write a sequent as $A_1, \ldots, A_n \vdash G$. The multiset of hypotheses is $\{A_1, \ldots, A_n\}$. Other formulations are possible, for instance, using sets or lists of hypotheses instead of multisets. Multisets will require less proof bookkeeping than sets, and provide an interesting abstract data type with which to illustrate our approach.

A proof itself is basically a tree labelled by sequents and rules. Rules follow the sequent calculus pattern discovered by Gentzen. Here are some of them in the usual style:

$$A_1, \ldots, A_n \vdash A_i \text{ by Hypothesis, } A_i \text{ provided } 1 \leqslant i \leqslant n.$$

$$A_1, \ldots, A_n \vdash G \text{ by FalseLeft provided one of the } A_i \text{ is } \bot.$$

$$\frac{A_1, \ldots, A_n \vdash G}{A_1, \ldots, A_n \vdash G \vee B} \text{ by OrRight 1, } B$$

$$\frac{A_1, \ldots, A_n \vdash G}{A_1, \ldots, A_n \vdash B \vee G} \text{ by OrRight 2, } B$$

$$\frac{A_1, \ldots, A, \ldots, A_n \vdash G \quad A_1, \ldots, B, \ldots, A_n \vdash G}{A_1, \ldots, A \vee B, \ldots, A_n \vdash G} \text{ by OrLeft, } A, B$$

$$\frac{A_1, \ldots, A, \ldots, A_n \vdash B}{A_1, \ldots, A_n \vdash A \Rightarrow B} \text{ by ImpRight, } A$$

$$\frac{A_1, \ldots, B, \ldots, A_n \vdash G \quad A_1, \ldots, A_n \vdash A}{A_1, \ldots, A \Rightarrow B, \ldots, A_n \vdash G} \text{ by ImpLeft, } B$$

Here is an example of a simple proof according to these rules in this format.

$$\begin{array}{cl} A \vdash A & \text{by Hypothesis, } A \\ \hline A \vdash A \vee B & \text{by OrRight 1, } B \\ \hline \vdash A \Rightarrow (A \vee B) & \text{by ImpRight, } A \end{array}$$

4. Abstract Logic

Abstract data types provide a method for specifying types together with operations to construct and use elements of the types. In a sense, they allow the definition of types characterized by operations rather than members. They are abstract in the sense that they only specify the available operations, they do not give their implementation. In Nuprl, abstract data types are encoded using dependent tuple types: each admissible implementation is given as a tuple belonging to this type. The tuple contains a carrier type, constructor functions that produce elements of the carrier, primitive operations on the type, and evidence that the constructors and primitive operations interact correctly. A similar type theoretic approach to abstract data types is found in [21].

We use abstract data types in our completeness proof for two reasons. First, they allow the description of the decision procedure arising from the completeness proof to be made relative to a standard, natural presentation of the object logic. Since the object logic is usually much less rich than Nuprl, this simplifies the required proofs significantly. Second, they facilitate the use of our theorem by other members of the theorem proving community. All that is required to apply our theorem to other theorem provers is the provision of an appropriate instantiation for the abstract data types. The alternative approach, using Nuprl's types directly, requires that the user determine which features of Nuprl's logic we use, and show how these features are mapped into their target logic.

Abstract data types can depend on other objects. For example, we give an abstract data type of multisets which depends on the type over which multisets are to be formed, and on a decision function for equality on this type. The intention of this abstract data type is to specify a multiset type constructor. That is, we expect the implementation of multisets to be parametric in the type over which the multisets are formed (and its equality). This requires that the implementation of the multiset abstract data type be a function that maps a type A and an equality decision function for A to an implementation of multisets of A. This means that the multiset abstract data type must be a dependent function type.

The kind of dependence illustrated by the multiset abstract data type is not always appropriate. For example, we give an abstract data type for proofs. The type of proofs depends on the type of formulas. However, it would be excessively onerous to require that implementations of the proof abstract data type be parametric in the representation of formulas. This would require that an implementation be able to map any representation of formulas to a corresponding implementation of the proof data type. This is unreasonable because we need to be able easily to instantiate the proof abstract data type with concrete proof types. Concrete proof types are usually explicitly tailored to a particular concrete formula type. To make such a type parametric in formulas would require that we map formulas into the preferred representation. We would, in effect, have to implement the formula type all over again as part of the implementation of proofs.

Instead, we specify the proof abstract data type as a function. Roughly speaking, this function takes an implementation of formulas, and returns an *abstract* data type for proofs. Thus, it is the abstract data type of proofs that is parametric in formulas, rather than its instantiations. We instantiate particular instances of the proof abstract data type with tuples. An instantiation needs to be valid only for a particular instantiation of formulas.

For our presentation of the propositional calculus, we will define three abstract data types: *MULTISET*, *WFF* and *PROOF*. *MULTISET* is parameterized by a type and an equality decider for this type. *WFF* depends on the type used for variables, and the injection function used to make variables into formulas. *PROOF* depends on both *WFF* and *MULTISET*.

4.1. MULTISETS

A *multiset* over a type A is a collection of elements of A in which a given element may appear multiple times, but in which there is no ordering of the elements. Any finite multiset is either empty, or can be thought of as the result of adding an element to another (smaller) finite multiset. The order in which these elements are added is irrelevant; the same multiset is produced regardless of the order. An induction principle can be given for finite multisets.

We give a parameterized abstract data type of multisets. It gives a type theoretic specification of any implementation of the finite multiset type constructor. Given any type A, and a decision function for equality on A, each implementation of the multiset abstract data type must produce an object that implements multisets of A. This object is a tuple. It includes a carrier type M, a function that decides equality on M, multiset constructors, and a multiset induction function.

The equality decision function induces an equivalence relation on M, which is used as multiset equality. We use an explicit representation of multiset equality for two reasons. First, it makes explicit the decidability of equality for finite multisets on discrete types. Second, and more generally, to treat a data type abstractly, we should also treat equality on the type abstractly as well. This emphasizes the fact that we are not simply defining an inductive structure.

Two constructors are required in the implementation of multisets. A nullary constructor is used to produce the empty multiset. And a constructor is provided with which to add an element to an existing multiset. The induction function provides an implementation of the induction principle on multisets. It can be used to define functions and predicates by structural induction on multisets.

The induction principle itself is quite simple. Given a type constructor P parameterized by multisets, the induction principle gives us a means by which to produce an element of the type $P(x)$ for any multiset x. To do so, it requires that we provide an object of type $P(e)$, where e is the empty multiset. It also requires us to provide a function f that maps a multiset m, an element a, and an object of type $P(m)$, to an object of type $P(m')$, where m' is the multiset formed by the addition of a to m. The intuition is that we begin with the object of type $P(e)$ and use f repeatedly, once for each element of x, to build up an object of type $P(x)$.

The specification is somewhat complicated because we want to get the same answer regardless of the order in which we take the elements of x. To guarantee that this is the case, we must impose an additional restriction on f. As well as being functional in its arguments, we require that if a and a' are elements, m is a multiset, and x is an object of type $P(m)$, then the object based on x which is created by f through adding first a and then a' is the same as the object created by adding first a' and then a. The intuition behind this restriction is that f must not allow us to distinguish between multisets created in different orders.

We define *MULTISET*, the abstract type of multisets, formally as a dependent function type. The domain type is a type A, and its codomain is itself a function type. The domain of the result is the type of equality deciders for A, and its codomain is a dependent product type intended to specify implementations of multisets of A. An equality decider for A is a function from A to A to *bool* which induces an equivalence relation on A when its result is treated as a proposition in the natural way. In order to express the complete definition of *MULTISET* relatively concisely, we shall introduce some notation. We use $\equiv$ to represent the definition of an abbreviation or some notation; the symbol $=$ represents equality within the type theory.

DEFINITION 1 ($\uparrow \{P\}$ and *EqDecider A*).

$$\begin{array}{l}
\uparrow \{P\} \equiv \text{if } P \text{ then } \mathit{trueProp} \text{ else } \mathit{false\ Prop} \\
\mathit{EqDecider}\,A \equiv \\
\qquad \{f : A \rightarrow A \rightarrow \mathit{bool} \mid \lambda x, y.\uparrow \{f(x)(y)\} \text{ is an equivalence on } A\}
\end{array}$$

where *trueProp* and *falseProp* are some canonical true and false propositions, respectively.

An implementation of a multiset of A consists of a carrier type M, an equality decider for M that induces an equality for the multiset, a distinguished element *Empty* of M that serves as the empty multiset, a function *Add* mapping A to M to M that serves as the multiset insertion function, an implementation *MInd* of multiset induction, and evidence that these objects interact correctly. These interactions are expressed as logical axioms, and the evidence consists of several objects, one for each axiom, which belong to the type used to express the axiom. There are four such axioms.

The function *Add* is required to be functional with respect to the equivalence induced by *Aiseq* in its first argument and with respect to the one induced by *Eq* in its second. To facilitate our presentation we introduce some notation. For any type constructor $T[a, m]$ parameterized by an element a of A, and a multiset m, we define $a : A \rightrightarrows m : M \rightrightarrows T[a, m]$ to be the type of functions that map an element a of A and a multiset m to $T[a, m]$ and that are functional in the appropriate fashion.

DEFINITION 2 ($a : A \rightrightarrows m : M \rightrightarrows T[a, m]$).

$$\begin{array}{l}
a : A \rightrightarrows m : M \rightrightarrows T[a, m] \equiv \\
\quad \{f : a : A \rightarrow m : M \rightarrow T[a, m] \\
\qquad \mid \forall a, a' : A, m, m' : M. \\
\qquad\quad \uparrow \{\mathit{Aiseq}(a)(a')\} \Rightarrow \uparrow \{\mathit{Eq}(m)(m')\} \\
\qquad\quad \Rightarrow (f(a)(m) = f(a')(m') \in T[a, m]) \\
\quad \}
\end{array}$$

Then, *Add* must be a member of $A \rightrightarrows M \rightrightarrows M$. Note that, as usual, we omit the bindings if they are not used. Recall that the order in which elements are added to multisets is supposed to be irrelevant. This imposes a constraint on the interaction of *Add* and *Eq*. Given a multiset m, if we add elements a and a' to m, we should get equal multisets regardless of the order in which a and a' are added to m. This yields the first of the four axioms:

DEFINITION 3 (Axiom 1).

$$\begin{aligned}&\mathit{Axiom}1 \equiv\\&\quad \forall a, a' : A.m : M. \uparrow \{\mathit{Eq}(\mathit{Add}(a)(\mathit{Add}(a')(s)))(\mathit{Add}(a')(\mathit{Add}(a)(s)))\}.\end{aligned}$$

Rather than introducing this as an axiom, we could have included this as part of the type of *Add*, but it seems more natural to treat it as a separate axiom.

The function *MInd* implements multiset induction. As such, it must map a type constructor P on M, an element e of $P(\mathit{Empty})$, a function f, and a multiset m, to an element of $P(m)$. These arguments are required to meet certain constraints. The type constructor P is required to be functional with respect to the equivalence relation induced by *Eq*. And the function f is constrained as described above. In return, *MInd* is required to be functional with respect to the equivalence relation induced by *Eq* in its fourth argument m. Moreover, *MInd* is required to behave extensionally as our intuition suggests it should. If m is an empty multiset, it should return e. And if m can be viewed as $\mathit{Add}(a)(m')$ for some element a of A, and multiset m', then $\mathit{MInd}(P)(e)(f)(m)$ should be $f(a)(m')(\mathit{MInd}(P)(e)(f)(m'))$. These three conditions on *MInd* give rise to the other three axioms for multisets.

Before we give the type of *MInd* and the three remaining axioms, we introduce some more notation. First, for any type T, the type $M \rightrightarrows T$ contains all the functions from M to T that are functional with respect to the equivalence relation induced by *Eq*.

DEFINITION 4 ($M \rightrightarrows T$).

$$M \rightrightarrows T \equiv \{f : M \to T \mid \forall m, m' : \uparrow \{\mathit{Eq}(m)(m')\} \Rightarrow f(m) = f(m') \in T\}.$$

Then P must belong to the type $M \rightrightarrows \mathit{Type}$. The type *Type* is a *universe type*; in Nuprl there is a hierarchy of universe types to avoid problems with the *Type* : *Type* construction. For convenience, we ignore the details of the hierarchy and just consider one type universe called *Type*.

Next, we introduce a type constructor *IndFun* with which to specify the type of f. Let A be a type, and let *Aiseq* be an equality decider for this type. Suppose that M is the carrier type of an implementation of multisets for the type A and the equality decider *Aiseq*. Suppose that *Eq* is the equality decider for M. Suppose that *Add* is the multiset insertion function. If P is a type constructor that maps elements of M to types, then $\mathit{IndFun}(A, \mathit{Aiseq}, M, \mathit{Eq}, \mathit{Add}, P)$ is the type containing the acceptable functions f.

The operator *IndFun* is implemented as a subset of the type $a : A \rightrightarrows m : M \rightrightarrows (P(m) \to P(\mathit{Add}(a)(m)))$ of functions mapping elements a of A, multisets m, and objects x of $P(m)$, to elements of the multiset that arises by inserting a into m. A function f in *IndFun* must satisfy two conditions. First, f must be functional in its first two arguments with respect to the equalities induced by *Aiseq* and *Eq*, respectively. This is guaranteed by the definition of $a : A \rightrightarrows m : M \rightrightarrows (P(m) \to P(\mathit{Add}(a)(m)))$. Second, f must be insensitive to the order in which a multiset was constructed, in the sense described above. This is guaranteed by using the set construction.

DEFINITION 5 (*IndFun*).

$$\begin{aligned}
&\mathit{IndFun}(A, \mathit{Aiseq}, M, \mathit{Eq}, \mathit{Add}, P) \equiv\\
&\quad \{f : a : A \rightrightarrows m : M \rightrightarrows (P(m) \to P(\mathit{Add}(a)(m))) \mid\\
&\qquad \forall a, b : A, m : M, p : P(m).\\
&\qquad\quad f(a)(\mathit{Add}(b)(m))(f(b)(m)(p)) = f(b)(\mathit{Add}(a)(m))(f(a)(m)(p))\\
&\qquad\qquad \in P(\mathit{Add}(a)(\mathit{Add}(b)(m)))\}.
\end{aligned}$$

We shall abbreviate $\mathit{IndFun}(A, \mathit{Aiseq}, M, \mathit{Eq}, \mathit{Add}, P)$ as $\mathit{IndFun}(\ldots)$. The three axioms describing *MInd* are now straightforward:

DEFINITION 6 (Axioms 2–4).

$$\begin{aligned}
&\mathit{Axiom}2 \equiv\\
&\quad \forall P : M \rightrightarrows \mathit{Type}, e : P(\mathit{Empty}), f : \mathit{IndFun}(\ldots), m, m' : M.\\
&\qquad \uparrow \{\mathit{Eq}(m)(m')\} \Rightarrow \mathit{MInd}(P)(e)(f)(m) = \mathit{MInd}(P)(e)(f)(m') \in P(m)\\
&\mathit{Axiom}3 \equiv\\
&\quad \forall P : M \rightrightarrows \mathit{Type}, e : P(\mathit{Empty}), f : \mathit{IndFun}(\ldots).\\
&\qquad \mathit{MInd}(P)(e)(f)(\mathit{Empty}) = e \in P(\mathit{Empty})\\
&\mathit{Axiom}4 \equiv\\
&\quad \forall P : M \rightrightarrows \mathit{Type}, e : P(\mathit{Empty}), f : \mathit{IndFun}(\ldots), A : A, m : M.\\
&\qquad \mathit{MInd}(P)(e)(f)(\mathit{Add}(a)(m)) = f(a)(m)(\mathit{MInd}(P)(e)(f)(m))\\
&\qquad \in P(\mathit{Add}(a)(m))
\end{aligned}$$

We summarize the preceding discussion as a single type definition.

DEFINITION 7 (*MULTISET*).

$$\begin{aligned}
&\mathit{MULTISET} \equiv\\
&\qquad\quad A : \mathit{Type}\\
&\quad\ \to \mathit{Aiseq} : \mathit{EqDecider}(A)\\
&\quad\ \to M : \mathit{Type}\\
&\qquad \times \mathit{Eq} : \mathit{EqDecider}(M)\\
&\qquad \times \mathit{Empty} : M\\
&\qquad \times \mathit{Add} : A \rightrightarrows M \rightrightarrows M\\
&\qquad \times \mathit{MInd} : P : (M \rightrightarrows \mathit{Type}) \to e : P(\mathit{Empty}) \to f : \mathit{IndFun}(\ldots)\\
&\quad \to m : M \to P(m)\\
&\qquad \times \mathit{Axiom}1\\
&\qquad \times \mathit{Axiom}2\\
&\qquad \times \mathit{Axiom}3\\
&\qquad \times \mathit{Axiom}4
\end{aligned}$$

A more expository discussion of this abstract data type, and the use of type theory for abstract data types in general, can be found in [33].

Given *Multiset* : $MULTISET(A, Aiseq)$, we will use a record-like notation extract the components of the product; for example, $Multiset.M$ will denote the carrier type. This should be considered as merely a descriptive notation for the appropriate projection function for dependent products. Note that such a function will, in general, rely on earlier components of the product to describe the type of later components.

4.2. WELL-FORMED FORMULAS

For formulas, we define a family of abstract data types $WFF(Varname)$ indexed by the type *Varname*. Parameterizing the abstract type of formulas in this fashion makes it easy to provide representations for systems that have differing notions of variable names. It has the additional, fortuitous, benefit of making applied propositional logics very easy to represent: we can treat the nonpropositional terms of such logics as variable names. Thus, the theorems we prove are actually somewhat stronger than they first appear, since they can be immediately extended to propositional logics extended with constants.

Our abstract data type must allow the representation of the abstract syntax of formulas. This syntax is the free algebra generated by variables, the false constant, and three binary logical connectives (and, or, and implies). We provide a carrier type with which to represent formulas, and also provide constructor functions with which to build up formulas. We must also be able to destructure formulas and to prove theorems by induction over their structure. Obviously, we cannot provide operators for every conceivable destructuring operation and every conceivable theorem proving task. Instead, we provide a single structural induction form which subsumes these operations. Our definition of the abstract type of formulas is quite stylized, and it seems highly probable that it could be generated automatically from the statement that it must describe a free algebra on a particular set of constructors.

For a given type *Varname*, an instantiation of our formula abstract data type is a 8-tuple. Its first component is a carrier type *Wff*. Its next five components are constructors for formulas. First comes a function *VarInj* mapping *Varname* into *Wff* with which variables can be constructed. Next comes an object *False* of type *Wff* that provides the representation of the constant false. And then come three functions *And*, *Or* and *Implies*, each of which maps *Wff* to *Wff* to *Wff*. These represent, respectively, the and, or, and implies constructors. The seventh element of the tuple is an implementation *WffInd* of the formula induction principle, and the last element is evidence that the preceding elements are appropriately related.

The induction function *WffInd* is analogous to that for multisets. Just as was the case for multisets, the first argument is a type constructor P. For multisets this constructor is parameterized by multisets; here it is parameterized by formulas (i.e. it is a function from formulas to the intended result type) to allow for definition of functions with dependent types. The next several arguments specify how a value is computed for each constructor. For multisets there were two such arguments, one for *Empty*, and one for *Add*. Now there are five, one for each of the constructors. The last argument is a formula w, for which a value is to be computed. The result is of type $P(w)$. While this function takes more arguments than the corresponding function for multisets, it is actually somewhat easier to specify. This is true for two reasons. First, we have chosen to use the natural equality on *Wff* rather than some explicit equality as our representation of formula equality. Second, formulas are a free algebra, so formulas constructed differently should always be distinguished.

To build the value for variables, we require a function hv mapping variable names v to objects of type $P(\mathit{VarInj}(v))$. Our intention is that the value of the induction, for the variable $\mathit{VarInj}(v)$, should be $hv(v)$. For false, we need only to provide a value of type $P(\mathit{False})$. To facilitate our presentation of the remaining arguments, we introduce the following notation.

DEFINITION 8 (*WIndCase*).

$$\mathit{WIndCase}(A, P, K) \equiv a : A \to b : A \to P(a) \to P(b) \to P(K(a)(b)).$$

Using this notation, we can give a type for the function that specifies how values are to be computed for formulas built using the *And* constructor. This function takes two formulas, a and b, and two objects which are the appropriate values for a and b, respectively, and produces the appropriate value for the formula $\mathit{And}(a)(b)$. Thus, it should have type

$$a : \mathit{Wff} \to b : \mathit{Wff} \to P(a) \to P(b) \to P(\mathit{And}(a)(b)),$$

or, more concisely, $\mathit{WIndCase}(\mathit{Wff}, P, \mathit{And})$. The types of the corresponding functions for *Or* and *Implies* are given in an analogous fashion.

The only axiom we need is one asserting that for any type constructor P, and any set of value construction methods hv, hf, ha, ho, hi, the induction function behaves as we intuitively expect when applied to a formula formed using a particular operator. For example, we expect that evaluating $\mathit{WffInd}(P)(hv)(hf)(ha)(ho)(hi)$ (False) should produce the value hf. In stating this axiom, we shall use the function *WffInd* repeatedly, and in each instance, only the final, formula, argument differs. Thus, for any formula w, we use the notation $\mathit{WffInd}(w)$ to stand for $\mathit{WffInd}(P)(hv)$ $(hf)(ha)(ho)(hi)(w)$.

The abstract type of well-formed formulas is given by the definition in Figure 1.

4.3. AN EXAMPLE: MULTISETS OF FORMULAS

To illustrate the use of these types, we shall show how an implementation of multisets of formulas can be given. This construction is used in our definition of *PROOF*.

$$
\begin{array}{l}
\mathit{WFF}(\mathit{Varname} : \mathit{Type}) \equiv \\
\quad\quad \mathit{Wff} : \mathit{Type} \\
\quad \times\ \mathit{VarInj} : \mathit{Varname} \to \mathit{Wff} \\
\quad \times\ \mathit{False} : \mathit{Wff} \\
\quad \times\ \mathit{And} : \mathit{Wff} \to \mathit{Wff} \to \mathit{Wff} \\
\quad \times\ \mathit{Or} : \mathit{Wff} \to \mathit{Wff} \to \mathit{Wff} \\
\quad \times\ \mathit{Implies} : \mathit{Wff} \to \mathit{Wff} \to \mathit{Wff} \\
\quad \times\ \mathit{WffInd} : P : \mathit{Wff} \to \mathit{Type} \\
\quad\quad\quad \to hv : (v : \mathit{Varname} \to P(\mathit{VarInj}(v))) \\
\quad\quad\quad \to hf : P(\mathit{False}) \\
\quad\quad\quad \to ha : \mathit{WIndCase}(\mathit{Wff}, P, \mathit{And}) \\
\quad\quad\quad \to ho : \mathit{WIndCase}(\mathit{Wff}, P, \mathit{Or}) \\
\quad\quad\quad \to hi : \mathit{WIndCase}(\mathit{Wff}, P, \mathit{Implies}) \\
\quad\quad\quad \to w : \mathit{Wff} \\
\quad\quad\quad \to P(w) \\
\quad \times\ \forall P : \mathit{Wff} \to \mathit{Type}, hv : v : \mathit{Varname} \to P(\mathit{VarInj}(v)), hf : P(\mathit{False}), \\
\quad\quad ha : \mathit{WIndCase}(\mathit{Wff}, P, \mathit{And}), ho : \mathit{WIndCase}(\mathit{Wff}, P, \mathit{Or}), \\
\quad\quad hi : \mathit{WIndCase}(\mathit{Wff}, P, \mathit{Implies}). \\
\quad\quad\quad \forall v : \mathit{Varname}. \\
\quad\quad\quad\quad\quad \mathit{WffInd}(\mathit{VarInj}(v)) = hv(v) \in P(\mathit{VarInj}(v)) \\
\quad\quad\quad \&\ \mathit{WffInd}(\mathit{False}) = hf \in P(\mathit{False}) \\
\quad\quad\quad \&\ \forall a : \mathit{Wff}, b : \mathit{Wff}. \\
\quad\quad\quad\quad\quad \mathit{WffInd}(\mathit{And}(a)(b)) = ha(a)(b)(\mathit{WffInd}(a))(\mathit{WffInd}(b)) \\
\quad\quad\quad\quad\quad\quad\quad \in P(\mathit{And}(a)(b)) \\
\quad\quad\quad \&\ \forall a : \mathit{Wff}, b : \mathit{Wff}. \\
\quad\quad\quad\quad\quad \mathit{WffInd}(\mathit{Or}(a)(b)) = ho(a)(b)(\mathit{WffInd}(a))(\mathit{WffInd}(b)) \\
\quad\quad\quad\quad\quad\quad\quad \in P(\mathit{Or}(a)(b)) \\
\quad\quad\quad \&\ \forall a : \mathit{Wff}, b : \mathit{Wff}. \\
\quad\quad\quad\quad\quad \mathit{WffInd}(\mathit{Implies}(a)(b)) = hi(a)(b)(\mathit{WffInd}(a))(\mathit{WffInd}(b)) \\
\quad\quad\quad\quad\quad\quad\quad \in P(\mathit{Implies}(a)(b))
\end{array}
$$

Figure 1. The *WFF* abstract data type.

An instance *Multiset* of the *MULTISET* abstract data type is a function which maps a type A, and an equality decider for A, into an implementation of multisets of A. Thus, if *Wff* is a type representing formulas, and *WffEq* decides equality on *Wff*, $\mathit{Multiset}(\mathit{Wff})(\mathit{WffEq})$ is an implementation of multisets of formulas. Its carrier type, $\mathit{Multiset}(\mathit{Wff})(\mathit{WffEq}).M$, is the type that represents multisets of formulas.

Let *Varname* be some type that represents variable names. Then the type *WFF* (*Varname*) is the abstract data type of formulas. An instance *Wff* of this type is an implementation of formulas, and its carrier type *Wff.Wff* is a type that represents formulas. To provide a representation of multisets of formulas, we need only provide an equality decision function for this type. Given an equality decision function *VEq* for *Varname*, we can produce such a function using the formula induction function *Wff.WffInd*.

An equality decision function for formulas is a function that maps a pair of formulas a and b to the Boolean value true if the two formulas are to be regarded as equal, and to the Boolean value false otherwise. To define such a function, we use *Wff.WffInd* on a to produce a function that maps a formula to a Boolean value, and that returns true only if its argument is equal to a. We apply this function to b to complete our implementation. Thus, for suitable values of hv, hf, ha, ho, and hi, our definition is

$$\lambda a, b.(\mathit{Wff.WffInd}(\lambda w.(\mathit{Wff.Wff} \to \mathit{bool}))(hv)(hf)(ha)(ho)(hi)(a))(b).$$

Each of the functions hv, hf, ha, ho, and hi is itself defined using *Wff.WffInd*.

The function hv takes a variable name v and returns a function that maps a formula c to a Boolean value. The returned function should return true only when c is equal to the formula formed using v, that is, when it equals $\mathit{Wff.VarInj}(v)$. Clearly, c equals $\mathit{Wff.VarInj}(v)$ if and only if c is itself the injection of some variable name v', and v' and v are equal variable names. Thus, the returned function should return false when applied to a formula c which is not a variable injection and return $\mathit{VEq}(v)(v')$ when applied to the injection of v'. This means that hv ought to be defined as

$$\begin{aligned}&\lambda v.\lambda c.\mathit{Wff.WffInd}(\lambda w.\mathit{bool})(\lambda v'.\mathit{VEq}(v)(v'))(\mathit{false})(\lambda x, y, fx, fy.\mathit{false})\\&\qquad(\lambda x, y, fx, fy.\mathit{false})(\lambda x, y, fx, fy.\mathit{false})(c).\end{aligned}$$

The function ha takes two formulas x and y, and two functions fx and fy. The function fx maps formulas z to Boolean values, and returns true only if z equals x. The function fy is similar: it returns true only when applied to a formula equal to y. The function ha returns a function that maps a formula c to a Boolean value. The returned function should return true only when c is equal to the formula $\mathit{Wff.And}(x)(y)$. Clearly, this is the case if and only if c is itself of the form $\mathit{Wff.And}(x')(y')$ for formulas x' and y' equal to x and y, respectively. But x and x' are equal if and only if $fx(x')$ is true. Likewise, y and y' are equal if and only if $fy(y')$ is true. This suggests that the returned function should return false when applied to a non-and formula, and return the Boolean and of $fx(x')$ and $fy(y')$ when applied to an and formula $\mathit{Wff.And}(x')(y')$. This suggests the following definition.

$$\begin{aligned}&\lambda x, y, fx, fy.\lambda c.\mathit{Wff.WffInd}(\lambda w.\mathit{bool})(\lambda v'.\mathit{false})(\mathit{false})\\&\qquad(\lambda x', y', fx', fy'.fx(x')\&\&fy(y'))\\&\qquad(\lambda x', y', fx', fy'.\mathit{false})(\lambda x', y', fx', fy'.\mathit{false})(c).\end{aligned}$$

Here, the operator && is used to the denote the Boolean and operation.

The functions hf, ho, and hi are analogously defined. To save space, we omit their definitions. This entire construction defines an equality operator

$WffEq(VEq)(Wff)$ parameterized by an equality decision function *VEq* for *Varname*, and by an implementation of formulas that belongs to *WFF* (*Varname*).

4.4. PROOFS

Our last abstract type is used to represent proofs. Here, unlike formulas, we only need to be able to build proofs using inference rules, and to determine what a proof proves. We do not intend to do any nontrivial proof theory, so we don't need to be able to arbitrarily destructure proofs. In addition to simplifying our abstract data type, not providing a complete destructuring ability has practical benefits. In particular, it means that we can instantiate our type with a much greater set of implementations; we don't require that the particular implementation record the steps used to construct a proof, but only the top-level goal, and the fact that it is proven. This seems to correspond more closely to the view of proofs given by many extant theorem provers.

Instantiations of the proof abstract data type are tuples. They consist of a carrier type *Proof*, two functions *Hyps* and *Concl* which return respectively the hypotheses and the conclusion of a proof's top-level goal, and a function for each of the primitive inference rules.

Actually, we define a family of proof abstract data types indexed by a type *Varname*, used to represent variables; an equality decider for this type *VEq*; an instantiation W of the abstract data type *WFF*(*Varname*); and a M of *MULTISET*. The operator *WffEq*, sketched in the previous section, when applied to *VEq* and *Wff* provides an equality decider for $W.Wff$. We can use it, together with $W.Wff$ and M to provide an implementation of multisets of formulas. We use the notation *WffMset* to stand for the resulting tuple.

DEFINITION 9 (*WffMset*).

$$WffMset \equiv M(W.Wff)(WffEq(VEq)(W)).$$

The actual type that represents multisets is the carrier component $WffMset.M$ of this tuple. Its equality decider is *WffMset.Eq*. We also need operations with which to add an element to a multiset, and remove an element from a multiset. Finally, we need a predicate to test if an element belongs to a multiset. For the insertion operation, we can use *WffMset.Add*, but the others need to be constructed using the components of *WffMset*.

DEFINITION 10 (Membership test and remove operation for multisets).

$$\begin{aligned}
&a \in M \equiv \\
&\qquad WffMset.MInd(\lambda m.Prop)(False)(\lambda x, y, fy.\uparrow \{WffEq(a)(x)\} \vee fy)(M) \\
&remove(a, M) \equiv \\
&\qquad WffMset.MInd(\lambda m.WffMset.M)(WffMset.Empty) \\
&\qquad\qquad (\lambda x, y, fy.\text{if } VEq(a)(x) \text{ then } y \text{ else } WffMset.Add(x)(fy)) \\
&\qquad\qquad (M).
\end{aligned}$$

Note that while we have made our definitions only for *WffMset*, we could easily abstract away the explicit references to *WffMset* and *VEq* to make these definitions usable for arbitrary multisets. Note too that we are using the notation $x \in A$ ambiguously, both to denote multiset membership and type membership. Which one is intended in any given instance must be determined from the surrounding context. To simplify our presentation, we sometimes use the notation $\{a\} + M$ to stand for $\mathit{WffMset}.\mathit{Add}(a)(M)$, and $M - \{a\}$ to stand for $\mathit{remove}(a, M)$.

The specification of *Hyps* and *Concl* are very simple: all we need specify is that they are functions of the appropriate types. *Hyps* maps proofs (that is, elements of the carrier type) to multisets of formulas. *Concl* maps a proof to a single formula. The specifications of the primitive rule are, in contrast, complex. Each rule can be thought of as a function that maps zero or more proofs, and perhaps other parameters as well, to a new proof. The required relationship between these parameters and the resulting proof is stated precisely in the specification of each rule.

For example, consider the hypothesis rule, given earlier. It allows the proof of any sequent in which the conclusion is also a hypothesis. It can be thought of as a function that maps a conclusion formula c, and a hypothesis multiset h of which c is a member, to a proof that has c as its goal, and h as its hypothesis. We formalize this using the following definition.

DEFINITION 11 (Hypothesis Rule).

$$\begin{aligned}
&\mathit{Hypothesis} \equiv\\
&\quad c : W.\mathit{Wff} \to h : \{h : \mathit{WffMset}.M \mid c \in h\}\\
&\qquad \to \{p : \mathit{Proof} \mid\uparrow \{\mathit{WffEq}(\mathit{Concl}(p))(c)\ \&\&\ \mathit{WffMset}.\mathit{Eq}(\mathit{Hyps}(p))(h)\}\}
\end{aligned}$$

Note that this definition is implicitly parameterized by an implementation W of formulas, an equality decider *WffEq* for it, a corresponding implementation *WffMset* of formula multisets, a carrier type *Proof* of proofs, and proof destructuring functions *Hyps* and *Concl*. When we use *Hypothesis* in the definition of the proof abstract data type, we intend for these parameters to be instantiated in the obvious fashion.

Each of the other rules is specified in a similar fashion. For example, the *False* left rule, and one of the two *Or* right rules can be given as follows.

DEFINITION 12 (*False* left and *Or* right rules).

$$\begin{aligned}
&\mathit{FalseLeft} \equiv\\
&\quad c : W.\mathit{Wff} \to h : \{h : \mathit{WffMset}.M \mid W.\mathit{False} \in h\}\\
&\qquad \to \{p : \mathit{Proof} \mid\uparrow \{\mathit{WffEq}(\mathit{Concl}(p))(c)\ \&\&\ \mathit{WffMset}.\mathit{Eq}(\mathit{Hyps}(p))(h)\}\}\\
&\mathit{OrRight}1 \equiv\\
&\quad b : W.\mathit{Wff} \to\ pa : \mathit{Proof}\\
&\qquad \to \{p : \mathit{Proof} \mid\uparrow \{\mathit{WffMset}.\mathit{Eq}(\mathit{Hyps}(p))(\mathit{Hyps}(pa))\}\\
&\qquad\quad \&\uparrow \{\mathit{WffEq}(\mathit{Concl}(p))(W.\mathit{Or}(\mathit{Concl}(pa))(b))\}\}
\end{aligned}$$

$$
\begin{array}{l}
\mathit{PROOF}(\mathit{Varname} : \mathit{Type}, \mathit{VEq} : \mathit{EqDecider}(\mathit{Varname}), \\
\qquad W : \mathit{WFF}(\mathit{Varname}), M : \mathit{MULTISET}) \equiv \\
\qquad \mathit{Proof} : \mathit{Type} \\
\quad \times\ \mathit{Concl} : \mathit{Proof} \to W.\mathit{Wff} \\
\quad \times\ \mathit{Hyps} : \mathit{Proof} \to \mathit{WffMset}.M \\
\quad \times\ \mathit{Hypothesis} \times \mathit{FalseLeft} \times \mathit{AndRight} \times \mathit{AndLeft} \times \mathit{OrRight}1 \times \mathit{OrRight}2 \\
\quad \times\ \mathit{OrLeft} : \\
\qquad\quad a : W.\mathit{Wff} \to b : W.\mathit{Wff} \to pa : \{pa : \mathit{Proof} \mid a \in \mathit{Hyps}(pa)\} \\
\qquad \to pb : \{pb : \mathit{Proof} \mid \uparrow \{\mathit{WffEq}(\mathit{Concl}(pb))(\mathit{Concl}(pa))\} \\
\qquad\qquad\qquad \& \uparrow \{\mathit{WffMset}.\mathit{Eq}(\mathit{Hyps}(pb))(\{b\} + (\mathit{Hyps}(pa) - \{a\}))\}\} \\
\qquad \to \{p : \mathit{Proof} \mid \uparrow \{\mathit{WffEq}(\mathit{Concl}(p))(\mathit{Concl}(pa))\} \\
\qquad\qquad \& \uparrow \{\mathit{WffMset}.\mathit{Eq}(\mathit{Hyps}(p))(\{W.\mathit{Or}(a)(b)\} + (\mathit{Hyps}(pa) - \{a\}))\}\} \\
\quad \times\ \mathit{ImpRight} : \\
\qquad\quad a : W.\mathit{Wff} \to pa : \{pa : \mathit{Proof} \mid a \in \mathit{Hyps}(pa)\} \\
\qquad \to \{p : \mathit{Proof} \mid \uparrow \{\mathit{WffEq}(\mathit{Concl}(p))(W.\mathit{Implies}(a)(\mathit{Concl}(pa)))\} \\
\qquad\qquad \& \uparrow \{\mathit{WffMset}.\mathit{Eq}(\mathit{Hyps}(p))(\mathit{Hyps}(pa) - \{a\})\}\} \\
\quad \times\ \mathit{ImpLeft} : \\
\qquad\quad b : W.\mathit{Wff} \to pa : \mathit{Proof} \\
\qquad \to pb : \{pb : \mathit{Proof} \mid \uparrow\{\mathit{WffMset}.\mathit{Eq}(\mathit{Hyps}(pb)) \\
\qquad\qquad\qquad (\{W.\mathit{Implies}(\mathit{Concl}(pa))(b)\} + \{b\} + \mathit{Hyps}(pa))\}\} \\
\qquad \to \{p : \mathit{Proof} \mid \uparrow \{\mathit{WffEq}(\mathit{Concl}(p))(\mathit{Concl}(pb)\} \\
\qquad\qquad \& \uparrow \{\mathit{WffMset}.\mathit{Eq}(\mathit{Hyps}(p)) \\
\qquad\qquad\qquad (\{W.\mathit{Implies}(\mathit{Concl}(pa))(b)\} + \mathit{Hyps}(pa))\}\}
\end{array}
$$

Figure 2. Part of the *PROOF* abstract data type.

To save some space, we omit the two *And* rules, and the other *Or* right rule. We give the specifications of the *Or* left rule, and the rules governing implications as part of the definition of *PROOF*, which we give in Figure 2, to illustrate exactly how we intend the implicit parameters mentioned above to be instantiated.

5. Completeness Theorem

In this section, we present a constructive proof of the completeness of Kripke models for intuitionistic propositional logic in such a way that the algorithm extracted from the proof is a form of the tableau decision procedure [11, 32]. By proving this theorem in Nuprl using abstract data types, we may then instantiate the ADTs with Nuprl's reflected term and proof types, and hence add the tableau algorithm as a decision procedure via the reflection rule.

5.1. KRIPKE MODELS

The statement of the completeness theorem which we shall prove is essentially "for any formula, either there is a proof of the formula or there is a Kripke model in which the formula is not forced." Such a Kripke model will be called a *countermodel* for a formula. Since we will give a constructive proof of this statement, we shall be able to extract a procedure to decide, for any given formula, whether a proof or a countermodel exists, and hence decide the intuitionistic provability of the formula. Furthermore, this procedure will construct either a proof of the formula or a specific Kripke model in which the formula is not forced.

It is also the case that if a formula has a proof, then the formula is forced in all Kripke models. This is the soundness theorem for Kripke models, and ensures that if a proof of the formula exists, it will be found by the completeness theorem's decision procedure. However, the soundness theorem does not have interesting computational content. We use our knowledge of the soundness theorem to guarantee that the decision procedure behaves as desired.

Because Kripke models are sound and complete for this logic, we may use a countermodel as evidence that a formula is not provable. Furthermore, such a countermodel actually serves as evidence that the formula is not provable in any conservative extension of intuitionistic propositional logic. The tableau decision procedure allows us to easily construct a countermodel from a failed search for a proof.

Before the completeness theorem can be stated, we must describe how we represent the model theory within Nuprl. Since there is no useful interpretation of Kripke models under reflection, we do not need to use abstract data types for the Kripke models themselves. We will still need to define the models with parameters for the types of variable names and formulas. In fact, the type of Kripke models need only be parameterized by the type of variable names; however, the definition of forcing in Kripke models will depend on the ADT of formulas as well.

In mathematical terms, a Kripke model is a triple consisting of a set, a transitive and reflexive relation on that set, and a function, which is monotone with respect to the relation, and which maps elements of the set to sets of atomic formulas. The set can be thought of as "states of knowledge"; the order relation then describes increasing information, and the set of atomic formulas associated with each state is considered to be the atomic formulas known to be true at that state.

A reasonably formal definition of Kripke model in a logic text would look something like the following:

DEFINITION 13 (Kripke models in mathematics). A Kripke model is a triple $\langle T, R, \mathit{af}\rangle$ where T is a set of states, R is a reflexive, transitive relation on T, and af is a function from T to the set of atomic formulas such that, for all a, a' in T, $s \in \mathit{af}(a)$ and $R(a, a')$ implies $s \in \mathit{af}(a')$.

In Nuprl, we shall represent the set of states as a type, so that elements of the type are the states. The relation on the set is represented as a function R from pairs of states to the type *bool*, so that the value of the function is true when the pair is in the relation. Specifically, the type of the function is restricted to produce reflexive and transitive relations. The atomic formulas forced at each state are described by a function *af* ("atomic forcing") on states and variables which returns true when the variable is forced at that state. The type of this function is similarly restricted so that the resulting relation is monotone with respect to R.

Formally, the type of Kripke models is the following:

DEFINITION 14 (Kripke models in type theory).

$$\begin{aligned}
&\mathit{Kripke\_model}(V : \mathit{Type}) \equiv \\
&\qquad T : \mathit{Type} \\
&\quad \times\, R : \{R : T \to T \to \mathit{bool} \mid \\
&\qquad\quad \forall a, b, c : T.\ \uparrow\{R(a, a)\} \& (\uparrow\{R(a, b)\} \& \uparrow\{R(b, c)\} \to \uparrow\{R(a, c)\})\} \\
&\quad \times\, \mathit{af} : \{\mathit{af} : T \to V \to \mathit{bool} \mid \\
&\qquad\quad \forall a : T, v : V.\ \uparrow\{\mathit{af}(a, v)\} \to \forall b : T.\ \uparrow\{R(a, b)\} \to \uparrow\{\mathit{af}(b, v)\}\}
\end{aligned}$$

(For notational convenience, we have curried the functions on pairs which represent relations.) Strictly speaking, we should have written the type *Kripke_model* as a dependent function type from $V : \mathit{Type}$ to triples $\langle T, R, \mathit{af}\rangle$; this explicit parameterization will be omitted when no confusion results.

Given this definition of the type of Kripke models, we can describe when a formula is forced in a model. The definition of forcing is by induction on the structure of the formula. Given *af*, which describes which atomic formulas are forced at each state, we can describe which nonatomic formulas are forced. We do this by defining a function *forces* of type

$$\mathit{forces}(V, W) : K : \mathit{Kripke\_model}(V) \to W.\mathit{Wff} \to K.T \to \mathit{Type}$$

To avoid an intuitionistically troublesome use of negation in the statement of the theorem, we also define a function *notforces* of the same type. We wish these to mean, "state s in Kripke model K forces formula w if and only if $\mathit{forces}(V, W)(K)(w)(s)$ is an inhabited type," and "state s in Kripke model K does not force formula w if and only if $\mathit{notforces}(V, W)(K)(w)(s)$ is an inhabited type." (We shall often interpret predicates as functions returning a type. Such predicates are "true" if and only if they return an inhabited type.) The details of these definitions are given in Appendix A.

Finally, a formula is said to be forced in a Kripke model if it is forced at every state in the model. For our completeness theorem, if a countermodel for a formula exists, we shall provide evidence for this by specifying a state in the model at which the formula is not forced.

5.2. STATING THE THEOREMS

We may now state the completeness theorem as we would like to use it. Since we have used ADTs to describe the logic, any theorem about the logic must be stated in terms of these ADTs. In order to make the statement of the theorem more concise, we refer to the ADTs on which the theorem depends as an *ADT-LOGIC*.

Formally, "Given an $\mathit{ADT\text{-}LOGIC}(V, \mathit{VEq}, W, M, \mathit{Proof})$, ..." should be read as an abbreviation for

$$\begin{aligned}&\forall V : \mathit{Type}.\forall \mathit{VEq} : \mathit{EqDecider}(V).\forall W : \mathit{WFF}(V).\forall M : \mathit{MULTISET}.\\ &\qquad \forall \mathit{Proof} : \mathit{PROOF}(V, \mathit{VEq}, W, M)....\end{aligned}$$

So, in an $\mathit{ADT\text{-}LOGIC}(V, \mathit{VEq}, W, M, \mathit{Proof})$, V denotes the type of variable names, *VEq* is a function that decides equality for V, W is an instance of the abstract data type of well-formed formulas V, M is an instance of the abstract data type of multiset constructors, and *Proof* is an instance of the data type of proofs of formulas in W.

The ideal version of the completeness theorem could be stated mathematically:

THEOREM 1 *For all formulas φ, either there is a proof P of φ or there is a Kripke model K and a state s in K such that s does not force φ.*

The formalization of this ideal version of the theorem is as follows:

THEOREM 1 *Given an $\mathit{ADT\text{-}LOGIC}(V, \mathit{VEq}, W, M, \mathit{Proof})$,*

$$\begin{aligned}&\forall \varphi : W.\mathit{Wff}\\ &\quad (\exists P : \mathit{Proof}.\mathit{Proof}\\ &\qquad\quad (\mathit{Proof}.\mathit{Hyps}(P) = \mathit{WffMset}.\mathit{Empty}\ \&\ \mathit{Proof}.\mathit{Concl}(P) = \varphi))\\ &\quad \vee\\ &\quad (\exists K : \mathit{Kripke\_model}(V)\exists s : K.T.\mathit{notforces}(K, \varphi, s))\end{aligned}$$

However, this statement is not strong enough for the proof we wish to use. As a result, this theorem will be proved as a corollary to a more complex theorem which provides more inductive structure.

The proof of this theorem is designed so that its computational content is the tableau algorithm. Accordingly, the extra complexity in the inductive hypothesis can be most easily understood as representing information about the tableau. In the classical tableau algorithm, given a formula φ, the algorithm first marks the formula as false, and then determines the consequences of this assumption for the truth or falsity of the subformulas of φ. For example, if $\varphi = \alpha \wedge \beta$ is false, then we must have α is false or β is false. The tableau then branches, with one branch containing the assumption that α is false, and the other the assumption that β is false. Each branch is then further developed according to the principle connectives in α and β. If both of these branches lead to contradictions (some formula is assumed both true and false), then the original formula φ must be true.

The intuitionistic tableau algorithm is similar, but requires a notion of a state or world at which the formula is marked true or false. We refer to these worlds as *nodes* in the tableau. Formulas are thus labelled true or false at a particular node. A branch of the tableau describes a set of nodes and the formulas labelled true or false at each of them.

The tableau rules are essentially sequent proof rules (see, e.g. [13]) written "upside down", i.e. from conclusion to hypotheses. Rules with multiple hypotheses correspond to branches in the tableau. It is a straightforward exercise to translate a tableau proof to a sequent proof [11, 12].

To describe the developing tableau, the actual statement of the theorem uses two new definitions, *NODE* and *SYSTEM*. A node is a pair of multisets of formulas, and a system is a multiset of nodes. In the proof, an element of *SYSTEM* will correspond to a branch in the intuitionistic tableau, and an element of *NODE* will describe which formulas are labelled true and which are labelled false at a particular state on a branch of the tableau. Specifically, since a node is a pair of sets of formulas, the first set in the pair is the set of formulas labelled true, while the second set is the set of formulas labelled false. This corresponds to a (multiconclusion) sequent with the "true" formulas forming the assumptions and the "false" formulas forming the conclusion.

A branch (or system) is *closed* if it contains a node at which the same formula is labelled both true and false. If every branch in the tableau developing from a node is closed, we say that node is *provable*. In the event that a countermodel is produced, it will be defined by the element of *SYSTEM* corresponding to a branch of the tableau which fails to close. Nodes in this system will correspond to states in the countermodel, and will describe which formulas are forced and not forced at that state. This correspondence is made explicit by a function f which maps nodes to states in the Kripke countermodel.

To simplify the proof, we make one restriction on the systems to which the theorem applies. This restriction is not strictly necessary, as the theorem is valid without it, but it allows us to use a much simpler inductive measure. We shall call a node *complete* if no more formulas can be added to it under the rules of the tableau algorithm. Similarly, we call a system *complete* if no more nodes can be added to it. A node is *open* if it is not complete. The theorem then applies only to systems with no more than one open node, and, if there is an open node, it must not be contained within any other node in the system. Obviously, if the system contains only one node (as is the case when we wish to check the provability of a single formula), this condition is satisfied. We shall see that this condition is preserved at each stage.

The formal definition of when a node is complete is the definition of a function $\mathit{Node\_complete}(V, W, M)$ of type $(\mathit{Wffset} \times \mathit{Wffset}) \rightarrow \mathit{Type}$.

DEFINITION 15 (*Node_complete*)

$$\mathit{Node\_complete}(V, W, M)(N) \equiv$$

$$\begin{aligned}
&\forall a, b : W.Wff.\\
&\quad W.And(a,b) \in \pi_1(N) \to a \in \pi_1(N)\ \&\ b \in \pi_1(N)\\
&\&\ \ W.Or(a,b) \in \pi_1(N) \to a \in \pi_1(N)\ \vee\ b \in \pi_1(N)\\
&\&\ \ W.Implies(a,b) \in \pi_1(N) \to a \in \pi_2(N)\ \vee\ b \in \pi_1(N)\\
&\&\ \ W.And(a,b) \in \pi_2(N) \to a \in \pi_2(N)\ \vee\ b \in \pi_2(N)\\
&\&\ \ W.Or(a,b) \in \pi_2(N) \to a \in \pi_2(N)\ \&\ b \in \pi_2(N)
\end{aligned}$$

As we shall see, the rule for $Implies(a, b) \in \pi_2(N)$ is not applied until the node is complete, as this rule creates a new node. Here, *Wffset* is an abbreviation for *WffMset*.M. We shall continue to use this abbreviation for clarity. Note this is the carrier set of the multiset, so we can treat it as a set itself, with the appropriate operations assumed to come from the full description of the multiset type. Similarly, let *Wffseteq* be an abbreviation for *WffMset*.*Eq*.

The theorem then applies to systems which satisfy the following predicate:

DEFINITION 16 (Eligible systems).

$$\begin{aligned}
Eligible(S) \ \equiv\ &(\exists N \in S.\forall N' \in S.(Node\_complete(N') \vee N = N')\\
&\wedge N \not\subset N')
\end{aligned}$$

This is merely a formal description of the requirement that there be at most one open node, and that the open node not be contained in any complete node. This restriction avoids the possibility that two distinct nodes will become identical as rules are applied; since only a finite number of such confusions can occur, the theorem is valid without the restriction but requires a more complex induction measure to account for this.

We have used a containment predicate in the previous definition that is not part of the multiset definition. In general, we omit the definitions of such predicates, since they are straightforward, but tedious, inductive definitions.

The theorem we will prove is informally stated.

THEOREM 2 *For every eligible system S, either there exists a node N within S such that N is provable, or there is a Kripke model K such that for all nodes N in S there is a state in K which forces all formulas in $\pi_1(N)$ and does not force any formula in $\pi_2(N)$.*

The theorem we will prove is the following:

THEOREM 2 *Given an ADT-LOGIC(V, VEq, W, M, Proof), let*

$$\begin{aligned}
NODE &= Wffset \times Wffset\\
NodeEq &= \lambda x.\lambda y.Wffseteq(\pi_1(x), \pi_1(y))\ \&\ Wffseteq(\pi_2(x), \pi_2(y))\\
SYSTEM &= M(NODE)(NodeEq)
\end{aligned}$$

in

$$\begin{array}{l}
\forall S : \{S : \mathit{SYSTEM}.S | \mathit{Eligible}(S)\}. \\
\quad (\exists N : \mathit{NODE}.\exists P : \mathit{Proof}.\mathit{Proof}. \\
\qquad N \in S \;\&\; (\mathit{Proof}.\mathit{Hyps}(P) = \pi_1(N)) \;\&\; \mathit{Proof}.\mathit{Concl}(P) \in \pi_2(N)) \\
\quad \vee \\
\quad (\exists K : \mathit{Kripke\_model}(V) \\
\qquad \exists f : \{f : \{N : \mathit{NODE}\, | N \in S\} \to K.T\} \\
\qquad \forall N : \{N : \mathit{NODE} | N \in S\} \\
\qquad \forall w : W.\mathit{Wff}.w \in \pi_1(N) \to \mathit{forces}(K, w, f(N))\& \\
\qquad w \in \pi_2(N) \to \mathit{notforces}(K, w, f(N))\})
\end{array}$$

Once we have proved the second theorem, it is easy to obtain the first theorem as a corollary. Given a formula φ, apply the second theorem to the system $S = \{\langle \emptyset, \varphi\rangle\}$. Since there is only one node in this system, it trivially satisfies the condition that there be at most one open node in the system. The result of applying the theorem is either a proof of φ or a model and a state in that model in which φ is not forced.

5.3. THE TABLEAU ALGORITHM

The proof of Theorem 2 is found in Section 5.5. Since the proof is developed so that the computational content is a tableau algorithm, it will be helpful to review the algorithm first in more detail in order to understand the structure of the proof.

The algorithm can be interpreted as a systematic search for a countermodel, such that failure to find a countermodel gives a proof. Given a formula φ, we begin the tableau with $0F\varphi$, which asserts that node 0 in some Kripke model does not force the formula φ. We then systematically specify the consequences of this assertion for the forcing of subformulas of φ according to their main connectives, adding new nodes and new branches if necessary. If a branch ever contains the assertions $pF\alpha$ and $pT\alpha$ for some node p, then this is a contradiction and the branch is said to be *closed*. If every branch of a tableau is closed, the tableau is also said to be closed, and a cut-free sequent proof can be extracted from the result. If there is an open branch, it can be interpreted as the description of a Kripke model in which φ is not forced.

As an example, consider the tableau in Figure 3 for the formula $((A \to B) \to A) \to A$ (Peirce's law). The nodes 0, 00, and 000 are ordered by the prefix ordering, so $0 < 00 < 000$. There are two branches in this tableau. The left one is closed (signified by an X), since on that branch we have both 00T A and 00F A. The right branch, however, is open, and no further development of this branch will produce a contradiction. The model which this branch describes is a three-node set $\{0, 00, 000\}$ such that the only atomic formula forced is A, forced at 000. (In fact, since nodes 0 and 00 force the same set of formulas, they could be collapsed into one node.) In this model, the formula $((A \to B) \to A) \to A$ is not forced at node 0, so this is a countermodel for Peirce's law.

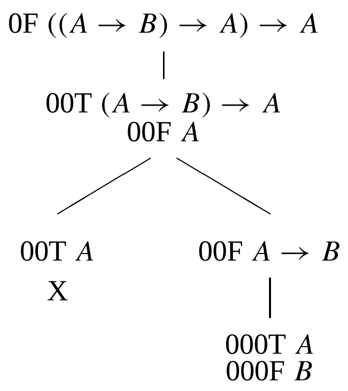

Figure 3. Tableau for Peirce's law.

In this tableau, it is not obvious that further development of the right branch will not yield a contradiction. One of the disadvantages of the tableau algorithm for intuitionistic propositional logic is that the termination condition is difficult to describe. An advantage of our presentation of the algorithm is that termination of the resulting function is automatic, since the algorithm is extracted from an inductive proof. The advantage lies in the fact that an incorrect induction measure will lead to an unprovable theorem, whereas an incorrectly coded termination condition may lead to a nonterminating computation.

5.4. TERMINATION

Intuitively, the proof is by induction on the number of formulas which can be added to the system. Each step in the tableau development above either adds a new formula to an existing node on some branch or adds a new node to some branch. In the proof, a system corresponds to a branch of the tableau. Each step in the proof either adds a new formula to a node in the system, or adds a new node to the system, and this must decrease the inductive measure. When the tableau branches, we create two systems, each of which has a new formula added to some node.

To understand how the actual inductive measure works, we must look more carefully at how formulas are added to nodes and how nodes are added to systems. We shall see that all formulas added to a node are subformulas of the formulas already in the node, so as long as at least one new formula is added at every stage, only finitely many such additions are possible. Similarly, a node is added only if it is not contained in another node in the system. Since new nodes also contain only subformulas of formulas in the original node, only finitely many nodes can be added. In addition, all formulas added are constructed from the finite alphabet of distinct propositional variables present at the outset. No simplification by renaming or α-conversion of proofs is considered.

In order to formalize this, we need to examine the data structures for nodes and systems, and describe how to use these to get a formal inductive measure which will be reduced at each stage. The data structure for nodes in the proof is a pair of multisets of formulas $\langle M_1, M_2\rangle$; M_1 is the set of formulas tagged T at that node in the tableau, and M_2 the set of formulas tagged F. A system is simply a multiset of nodes.

Formally, there will be two inductions in the proof of the theorem, one based on the number of formulas which can be added to nodes, and one based on the number of nodes which can be added to the system. These are both measured as set differences between a given node and system and a maximum node and maximum system constructed from the formulas in the original system. Neither the maximum node nor the maximum system is attainable if the theorem is originally applied to a system containing one node of the form $\langle\emptyset, \varphi\rangle$, so we shall also describe extra conditions which will terminate the induction.

These extra conditions can be motivated by considering the tableau procedure as a search for a countermodel. At each stage, the system can be seen as a partial model, with information about what needs to be forced and not forced at each node to produce a countermodel. The model is complete when, for each node $N = \langle M_1, M_2\rangle$, N forces everything in M_1 and doesn't force anything in M_2, using the forcing of atoms as a base. The order on the nodes in the model is defined by inclusion on the first set in each pair; since this represents the set of formulas which are forced, this ensures that forcing is monotone.

For example, the tableau development above corresponds to the following sequence of systems. The original system is

$$\{\langle\emptyset, \{((A \to B) \to A) \to A\}\rangle\}.$$

This asserts that at some node, $((A \to B) \to A) \to A$ is not forced. For that to be true, there must be some node greater than or equal to this one in the model such that $(A \to B) \to A$ is forced but A is not. So, we add such a node to the system, resulting in the system

$$\{\langle\emptyset, \{((A \to B) \to A) \to A\}\rangle, \langle\{(A \to B) \to A\}, \{A\}\rangle\}.$$

Now, in order to have $(A \to B) \to A$ forced at this node, we must have either $A \to B$ is not forced at this node, or A is forced. So, we create two new systems to account for these possibilities. The first is

$$\{\langle\emptyset, \{((A \to B) \to A) \to A\}\rangle, \langle\{(A \to B) \to A, A\}, \{A\}\rangle\}.$$

This system corresponds to the left branch, and is closed since A cannot be both forced and not forced. The other system is

$$\{\langle\emptyset, \{((A \to B) \to A) \to A\}\rangle, \langle\{(A \to B) \to A\}, \{A, A \to B\}\rangle\}.$$

Again, for it to be false that $A \to B$ is forced, there must be a node greater than or equal to this one such that A is forced but B is not. So, we add such a node to the system, ensuring monotonicity by including the formulas forced here (namely, $(A \rightarrow B) \rightarrow A$). This produces the system

$$\{\langle\emptyset, \{((A \rightarrow B) \rightarrow A) \rightarrow A\}\rangle,$$
$$\langle\{(A \rightarrow B) \rightarrow A\}, \{A, A \rightarrow B\}\rangle\}\langle\{(A \rightarrow B) \rightarrow A, A\}, \{B\}\rangle\}.$$

Now nothing need be done in this system, since we have, for all nodes, the formulas in the first set are forced and those in the second set are not forced.

The rest of the section describes the proof in detail. First we give several definitions needed to implement the two inductive measures. Then, we define the inductive measures themselves, and the extra conditions which also may terminate the induction. Finally, we describe the general argument for the cases of the inductive proof itself, and present the base case in detail.

In the remainder of this section, we assume we have fixed an arbitrary *ADT-LOGIC*, i.e. we have fixed $V : \mathit{Type}$, $\mathit{VEq} : V \times V \rightarrow \mathit{Type}$, $W : \mathit{WFF}(V)$, $M : \mathit{MULTISET}$, and $\mathit{Proof} : \mathit{PROOF}(V, \mathit{VEq}, W, M)$, and we wish to prove the general theorem stated above.

There are two inductive measures, each dependent upon the size of the set difference between two multisets. One measure is an upper bound for the number of formulas which can be added to a node before it is complete. This is calculated as the set difference between the node and a maximum node which contains all subformulas of the original formulas in the system. Adding a subformula of a formula in a node to that node will decrease the measure as long as the formula was not already present in the node. A node is complete when no further formulas can be added. Similarly, the other inductive measure is a set difference between the current system and a system which contains all possible nodes made up from subformulas of formulas in the original system. This gives an upper bound on the number of nodes which can be added to the system.

In order to define the maximum node and maximum system, we shall first define a function which, given a formula, produces a multiset containing all subformulas of the formula. Then, a multiset containing all the subformulas of formulas in an node is defined, and this is used to define a multiset of subformulas of formulas in a system. These definitions will also illustrate the use of *WffInd*, the induction form in the ADT of formulas, and *MInd*, the induction form in the ADT of multisets. We shall require a function which takes the union of two multisets; the definition of this function is omitted.

The function which produces a multiset containing all subformulas of a given formula has type $W \rightarrow \mathit{Wffset}$ is defined as follows:

DEFINITION 17 (*subformula_set*).

$$\mathit{subformula\_set} \equiv \lambda x.W.\mathit{WffInd}(\lambda x.\mathit{Wffset})$$
$$(\lambda v.\mathit{WffMset}.\mathit{Add}(\mathit{WffMset}.\mathit{VarInj}(v), \mathit{WffMset}.\mathit{Empty}))$$
$$(\mathit{WffMset}.\mathit{Add}(W.\mathit{False}, \mathit{WffMset}.\mathit{Empty}))$$

$$(\lambda a, b, pa, pb.\mathit{Union}(\mathit{WffMset.Add}(W.\mathit{And}(a, b), pa), pb))$$
$$(\lambda a, b, pa, pb.\mathit{Union}(\mathit{WffMset.Add}(W.\mathit{Or}(a, b), pa), pb))$$
$$(\lambda a, b, pa, pb.\mathit{Union}(\mathit{WffMset.Add}(W.\mathit{Implies}(a, b), pa), pb))(x)$$

Given a multiset of formulas, the function which produces a multiset containing all subformulas of any formula in the set is defined by multiset induction. The type of the function is *Wffset* → *Wffset*.

DEFINITION 18 (*set_subformulas*)

$$\begin{aligned} \mathit{set\_subformulas} \equiv\ & \lambda x.\mathit{WffMset.MInd}(\lambda x.\mathit{Wffset}) \\ & (\mathit{WffMset.Empty}) \\ & (\lambda a, s, p.\mathit{Union}(\mathit{subformula\_set}(a), p)(x)) \end{aligned}$$

This is easily extended to nodes:

DEFINITION 19 (*node_subformulas*)

$$\begin{aligned} & \mathit{node\_subformulas} \equiv \lambda n.\mathit{Union}(\mathit{set\_subformulas}(\pi_1(n)), \\ & \quad \mathit{set\_subformulas}(\pi_2(n))) \end{aligned}$$

Finally, we can define the set of all subformulas of formulas which appear in a system:

DEFINITION 20 ($SF(S)$)

$$\begin{aligned} SF(S) \equiv\ & \mathit{WffMset.MInd}(\lambda x.\mathit{Wffset}) \\ & (\mathit{WffMset.Empty}) \\ & (\lambda a, s, p.\mathit{Union}(\mathit{node\_subformulas}(a), p))(S) \end{aligned}$$

So, given a system S, we have defined $SF(S)$, the multiset of all subformulas of formulas in nodes in S. The "maximum node" based on this system is then $SF(S) \times SF(S)$. Note that this node is provable, i.e., the intersection of the two sets is nonempty.

Following a similar procedure, we can define the "maximum system" based on the original system. Given a maximum node $N = SF(S) \times SF(S)$, define $SN(S)$, the multiset of all nodes M such that $\pi_1(M) \subset SF(S)$ and $\pi_2(M) \subset SF(S)$ (i.e. the set of all nodes based on formulas in S). Note that since N is in $SN(S)$, this system is provable.

Finally, we may also define the set difference of two multisets using multiset induction. In this case, we insist that the set difference contain only one instance of any set element, since we shall use the cardinality of the set difference to define the inductive measure.

We are now ready to define the two inductive measures used in the proof. The first induction is, intuitively, on number of nodes which can be added to the system.

An upper bound on this number is the size of set difference between the current system S and $SN(S)$. So, let

$$i_1(S) \equiv |SN(S) - S|$$

be the first (outer) induction measure. Note that when a node is added to a system S to create a system S', we shall have to verify that $i_1(S') < i_1(S)$, which involves both $SN(S)$ and $SN(S')$. However, we will be able to show that $SN(S) = SN(S')$ since any node added will contain only subformulas of formulas in the system S.

The second induction is, intuitively, on the number of formulas that can be added to nodes in a given system. An upper bound on this is the sum of sizes of set differences between N and the maximum node $SF(S) \times SF(S)$ for all nodes N in S. So, let

$$i_2(S) \equiv \sum_{N \in S} (|SF(S) - \pi_1(N)| + |SF(S) - \pi_2(N)|)$$

be the second (inner) induction measure. Again, although we may add a formula to a node in system S to create system S', the new formula is a subformula of some formula already in S. This means that $SF(S) = SF(S')$, so we can show that $i_2(S') < i_2(S)$.

The final induction measure is defined as a pair:

DEFINITION 21 (Induction measure) Given a system S, let

$$i(S) \equiv \langle i_1(S), i_2(S) \rangle$$

Let $i(S) < i(S')$ be determined by the lexicographic ordering on pairs.

In fact, since we can compute an upper bound for the values of i_2, we could make this a single induction on natural numbers by an appropriate pairing function. It is useful, however, to consider the inductions separately.

Although the inductive measure is reduced at each stage, in practice it will never reach zero. For example, the theorem does not even apply to the maximum system $SN(S)$ (since it has more than one open node in it). So, each of the inductions will terminate either when the inductive measure is zero, or when some predicate is true of the system.

In the case of the inner induction, we have already defined this predicate. Since this induction is on the number of formulas which can be added to a node, the predicate *Node_complete*, defined above, expresses precisely when no more formulas can be added. So, the inner induction will terminate either when $i_2(S) = 0$ or when, for all nodes N in S, *Node_complete*(N) is true.

Similarly, we define a predicate *System_complete*(V, VEq, W, M) of type *SYSTEM* $\rightarrow$ *Type*, for the outer induction, as follows:

DEFINITION 22 (*System_complete*)

$$\mathit{System\_complete}(V, \mathit{VEq}, W, M)(S) \equiv$$

$$\begin{aligned}
&\forall N : (\mathit{Wffset} \times \mathit{Wffset}).\forall a, b : W.\mathit{Wff}.N \in S \rightarrow \\
&\quad W.\mathit{Implies}(a, b) \in \pi_2(N) \rightarrow \\
&\qquad \exists N'.N' \in S \,\&\, a \in \pi_1(N') \,\&\, b \in \pi_2(N') \,\&\, \pi_1(N) \subseteq \pi_1(N')
\end{aligned}$$

Since a node is only added for the sake of an implication formula in the second multiset of some node, and then only when the node which would result is not contained in another node in the system, this predicate is true only when there is no such implication. Thus, this predicate characterizes when all possible nodes that can be added have been added. So, the outer induction will terminate either when $i_1(S) = 0$ or when $\mathit{System\_complete}(S)$. We shall see that if $i_1(S) = 0$ then the system is closed, but a complete system does not necessarily correspond to provability.

5.5. PROOF OF THE COMPLETENESS THEOREM

We shall now outline the argument used to prove the inductive cases, and present the base case in some detail. Although there are many inductive cases (two for each logical connective), their proofs are very similar.

Inductive Cases

If we are in the inner induction of the proof, we have $\exists N \in S.\neg \mathit{Node\_complete}(N)$ and $i_2(S) > 0$.

Now, $\neg\mathit{Node\_complete}(N)$ implies that there are formulas $a, b : W.\mathit{Wff}$ such that one of the following holds:

$W.\mathit{And}(a, b) \in \pi_1(N)$	but	$a \notin \pi_1(N) \vee b \notin \pi_1(N)$
$W.\mathit{Or}(a, b) \in \pi_1(N)$	but	$a \notin \pi_1(N) \,\&\, b \notin \pi_1(N)$
$W.\mathit{Implies}(a, b) \in \pi_1(N)$	but	$a \notin \pi_2(N) \,\&\, b \notin \pi_1(N)$
$W.\mathit{And}(a, b) \in \pi_2(N)$	but	$a \notin \pi_2(N) \,\&\, b \notin \pi_2(N)$
$W.\mathit{Or}(a, b) \in \pi_2(N)$	but	$a \notin \pi_2(N) \vee b \notin \pi_2(N)$

So, we have five cases depending on which one holds. If more than one holds, we may pick one arbitrarily. Given a system S satisfying one of the above conditions, we create a new system which is the same as the old system but with the appropriate formulas added to the node N which was not complete. For example, if the first case was true, then the new system would contain a node N' equal to N with the formulas a and b added to $\pi_1(N)$. Since at least one of these formulas was not there before, this decreases the inductive measure of the system by decreasing the set difference between that node and $SF(S)$.

The inductive hypothesis is then applied to this new system. If the new system is unprovable, then the original system is also unprovable, and the same Kripke countermodel suffices for both. If the new system is provable, then the original system is provable, and its proof is generally obtained by applying a single proof rule to the proof given for the new system. In the example of the previous paragraph, the proof rule would be *and-left*.

If we are in the outer induction, a similar procedure is followed. Two cases of the inner induction and the case of the outer induction are presented in detail in an Appendix B.

Base Case

Now we consider the base case. The base case must include the cases where either induction measure is zero, although when the theorem is applied to systems corresponding to a sequent with a single conclusion, the measure will never reach zero.

Induction Measures Equal to Zero. We shall show that if either induction measure is zero, the system is provable. We shall use this fact to conclude that, if a countermodel exists for some system S, we must have both $System\_complete(S)$ and, for all N in S, $Node\_complete(N)$.

The inner induction measure can only be zero if S is a system containing only the node $N = SF(S) \times SF(S)$. In this case, since $\pi_1(N)$ and $\pi_2(N)$ intersect, this system is provable by an axiom.

The outer induction measure can only be zero if S is a system such that $S = SN(S)$. Since $SF(S)$ is a node in $SN(S)$, this system is also provable, by the argument above.

Now suppose that S is a system such that $System\_complete(S)$ is true and, for all N in S, $Node\_complete(N)$.

Constructing a Proof. If there is an N in S such that $\pi_1(N) \cap \pi_2(N) \neq \emptyset$, then pick $a \in \pi_1(N) \cap \pi_2(N)$ and let $P = Proof.Hypothesis(a, \pi_1(N))$. This N is then the provable node in the system. The complete proof is built from axioms obtained by this rule. This situation corresponds to the closure of a branch of the tableau.

A branch also closes if it requires that $W.False$ be true, that is, if there is an N in S such that $W.False \in \pi_1(N)$. In this case, we may pick any formula a in $\pi_2(N)$ and prove this node with the rule $Proof.FalseLeft(a, \pi_1(N))$. If $\pi_2(N)$ is empty, we may pick a to be *False*; this case does not arise in systems generated by a normal tableau development.

Constructing a Countermodel. If there is no such provable node N, we construct a Kripke countermodel for this system as follows. Let $\subseteq_1 \colon \{N : NODE | N \in S\} \to \{N : NODE | N \in S\} \to bool$ be defined so that

$$\subseteq_1 (N_1, N_2) = true \leftrightarrow \pi_1(N_1) \subseteq \pi_1(N_2)$$

Note that this is a reflexive, transitive relation on the set $\{N : NODE | N \in S\}$.

Let $af : \{N : \mathit{NODE} | N \in S\} \to V \to \mathit{bool}$ be defined so that

$$af(N, a) = true \leftrightarrow W.\mathit{VarInj}(a) \in \pi_1(N)$$

Note that this defines a relation on $\{N : \mathit{NODE} | N \in S\} \times V$ which is monotone with respect to $\subseteq_1$, since $W.\mathit{VarInj}(a) \in \pi_1(N)$ and $N \subseteq_1 N'$ implies $W.\mathit{VarInj}(a) \in \pi_1(n')$.

Now let $K = \langle\{N : \mathit{NODE} | N \in S\}, \subset_1, af\rangle$. It is clear by the choice of the set and the relations $\subset_1$ and *af* that this is in the type of Kripke models. We must now produce the mapping from nodes in the system to states in the Kripke model, and verify that the appropriate formulas are forced at each state.

Since we have built the countermodel from the system, we may define f to be the identity function on $\{N : \mathit{NODE} | N \in S\}$. We now need to show

$$\begin{aligned} &\forall N : \{N : \mathit{NODE} | N \in S\}.\\ &\quad \forall w : W.\mathit{Wff}.w \in \pi_1(N) \to \mathit{forces}(K, w, f(N))\&\\ &\qquad w \in \pi_2(N) \to \mathit{notforces}(K, w, f(N)) \end{aligned}$$

Checking the Countermodel. To verify this property, we shall use the fact that, since we do not have a proof of any node in the system, we must have reached the base case with both $\mathit{System\_complete}(S)$ true and for all N in S, $\mathit{Node\_complete}(N)$ true, as noted above.

We show the property above by induction on formulas. Since $f(N) = N$ by definition of f, we actually need to show, for any formula w, $w \in \pi_1(N) \to \mathit{forces}(K, w, N)$ and $w \in \pi_2(N) \to \mathit{notforces}(K, w, N)$. This is true for variables by the definition of *af*. For the constant $W.\mathit{False}$, since we know $W.\mathit{False}$ is not in $\pi_1(N)$ for any node N (otherwise the system would be provable), we have $W.\mathit{False}$ is not forced at any node.

For more complex formulas, the result follows almost immediately from the inductive hypothesis, together with the fact that all nodes satisfy $\mathit{Node\_complete}(N)$ and the system satisfies $\mathit{System\_complete}(S)$. For instance, suppose $W.\mathit{And}(a, b)$ is in $\pi_1(N)$ for some N, and we wish to show $\mathit{forces}(K, W.\mathit{And}(a, b), N)$. By definition of forcing, we must show $\mathit{forces}(K, a, N)$ and $\mathit{forces}(K, b, N)$. But since $\mathit{Node\_complete}(N)$, we know $a \in \pi_1(N)$ and $b \in \pi(N)$, so they are forced by inductive hypothesis. The other cases for $W.\mathit{And}$ and $W.\mathit{Or}$ proceed similarly. The case in which we wish to show $\mathit{forces}(K, W.\mathit{Implies}(a, b), N)$ is slightly more complicated, since we must show that for all N' such that $N \subseteq_1 N'$, $\mathit{notforces}(K, a, N')$ or $\mathit{forces}(K, b, N')$. This follows from the fact that all nodes are complete, so $W.\mathit{Implies}(a, b) \in \pi_1(N)$ implies $a \in \pi_2(N)$ or $b \in \pi_1(N)$, and hence $\mathit{notforces}(K, a, N)$ or $\mathit{forces}(K, b, N)$ by inductive hypothesis. But, by definition of $\subseteq_1$, any node N' such that $N \subseteq_1 N'$ must have $W.\mathit{Implies}(a, b)$ in $\pi_1(N')$. Finally, the case of $\mathit{notforces}(K, W.\mathit{Implies}(a, b), N)$ follows from $\mathit{System\_complete}(S)$, and the inductive hypothesis.

This completes the treatment of the base case. Further details of the proof of the inductive cases are presented in the Appendix B.

5.6. EFFICIENT INDUCTION

Note that in the above proof that the actual values of the inductive measures were never used in constructing the proof or counterexample. Thus, it would be unfortunate if the algorithm extracted from the proof had to include the information needed to construct these values. Furthermore, since the algorithm is naturally recursive, it would be nice if the algorithm extracted from the proof were recursive instead of inductive.

To achieve this in Nuprl, we can use a theorem ("efficient induction") for the proof by induction instead of the built-in induction. Specifically, we can prove a modified induction principle for well-founded orders $<$ on systems. In this case, given the ADTs necessary to define the data type *SYSTEM*, it is sufficient to prove a theorem of the form

THEOREM 3 *(Efficient Induction)*

$$\begin{array}{l}\forall System : \mathit{SYSTEM}.\\ \forall P : System.S. \to \mathit{Type}.\\ \forall f : (System.S) \to N.\\ \quad (\forall S : System.S.\\ \quad (\forall S' : \{S' : System.S | f(S') < f(S)\}.P(S')) \Rightarrow P(S))\\ \qquad \Rightarrow (\forall S : System.S.P(S))\end{array}$$

The modification uses the set type $\{S' : System.S | f(S') < f(S)\}$, which guarantees that the computational content of the proof that the inductive measure of S' is less than the measure of S is not used in the extraction. This means that the numerical induction measures need not be explicitly calculated at run time as long as they were shown to have been reduced in the proof of the theorem which uses this induction theorem.

The induction principle above may be proved in Nuprl by supplying an explicit witness for its truth. In this case, the witness is essentially

$$\lambda P.\lambda f.\lambda g.(\mathbf{Y}\lambda h.\lambda x.(gxh))$$

where $\mathbf{Y} = \lambda f.(\lambda x.f(x(x)))(\lambda x.f(x(x)))$ is the usual Y combinator. (The actual term must explicitly list all the parameters needed to instantiate the datatype for *SYSTEM*. However, this does not affect the computation.) This term is shown to be an inhabitant of the appropriate type using Nuprl's computation rules, which allow the $\mathbf{Y}$ combinator to be "unrolled" in order to use the inductive hypothesis.

6. Instantiating ADT's

6.1. OVERVIEW

In the previous sections of the paper we have defined abstract data types for propositional logic, and proven that for any instantiation of these abstract data types, there is a function that maps a propositional formula either to a propositional proof of the formula or a Kripke countermodel. In this section, we provide a particular instantiation. Our intention is to instantiate the *WFF* and *PROOF* abstract data types so that the function guaranteed to exist by our abstract proof of the completeness theorem has certain properties. Let *Term* be a Nuprl type which is the internal description of Nuprl's terms, and similarly let *Proof* describe Nuprl proofs. We want the function extracted from the theorem to map certain elements of the Nuprl *Term* type either to elements of the Nuprl *Proof* type, or to Kripke models. The domain of this function should be those elements of the type *Term* that actually represent propositional formulas. Let w be a propositional formula, and let W be an element of *Term* that represents w. Moreover, suppose that, when

```
$Multiset \equiv \lambda A.\lambda Aiseq.\langle$
    $A\ list$
    $\lambda x.\lambda y.is\_permutation(x, y, Aiseq)$
    $nil,$
    $\lambda a.\lambda m.cons(a; m),$
    $\lambda P.\lambda hb.\lambda hi.\lambda m.list\_ind(m; hb; u, v, w.hi(u)(v)(w)),$
    $\lambda a.\lambda b.\lambda m.$ (proof of Axiom 1)
    $\lambda P.\lambda hb.\lambda hi.\lambda m.\lambda n.\lambda x.$ (proof of Axiom 2)
    $\lambda P.\lambda hb.\lambda hi.$ (proof of Axiom 3)
    $\lambda P.\lambda hb.\lambda hi.\lambda a.\lambda m.$ (proof of Axiom 4)
  $\rangle$
```

Figure 4. An instantiation of the *MULTISET* ADT.

applied to W, our function returns an element p of *Proof*. Then, p should represent a proof of w. These considerations dictate that the carrier element our instantiation of *WFF* must be a subtype of *Term*, and that the carrier component of our instantiation of *PROOF* must be a subtype of *Proof*.

6.2. MULTISETS

In addition to instantiating the *WFF* and *PROOF* abstract data types, we must also provide an instantiation for the *MULTISET* abstract type. Nuprl provides a list type constructor, and for the sake of simplicity, we use it in our instantiation of *MULTISET*. A given multiset is represented by a list containing its elements. Order is significant in lists, but not in multisets; therefore, the equality decision procedure we provide must identify lists that differ only in the order of their elements. Otherwise, this instantiation is straightforward. The empty multiset is implemented as the empty list; multiset element addition is implemented using *cons*; multiset induction is implemented by list induction. The various axioms are most easily instantiated by proving them for these instantiations of the multiset operations, and extracting an appropriate instantiation from the resulting theorem object; here, though, we simply give an appropriate term.

Here, $\mathit{is\_permutation}(x, a, \mathit{Aiseq})$ returns the Boolean true value if x and y are permutations of each other, and returns false otherwise. If x is empty, then y is a permutation of x if and only if it too is empty. Otherwise, y is a permutation of x if and only if y contains an element a equal to the head of x, and the list obtained by removing a from y is a permutation of the tail of x. The function *Aiseq* is used to decide equality between elements of x and y. The implementation of *is_permutation* is a straightforward application of Nuprl's *list_ind* operator. Alternatively, the **Y** combinator can be used.

The form of the proofs of the equational axioms is computationally irrelevant. In Nuprl, this is reflected by the fact that all such proofs are considered equal. Intensional type theories treat equality differently; in any case, however, the proofs that the relevant axioms are satisfied are straightforward.

6.3. FORMULAS

The *WFF* abstract data type is very precise. It specifies that *any* instantiation must have as its carrier the free type generated by the instantiations of the constructors. Together with the fact that we want our carrier to be a subtype of the Nuprl *Term* type, and our desire to make our instantiation as natural as possible, this determines most of our instantiation. *False* is instantiated with the member of *Term* that represents the term $\mathit{false}()$: $\texttt{False}()$. Our instantiation of *And* maps two members A and B of *Term* representing the terms a and b respectively to the element of *Term* that represents $\mathit{and}(a; b)$: $\texttt{And}(a; b)$. *Or* and *Implies* are instantiated similarly. *VarInj* is instantiated by the function mapping a member V of *Varname* that represents the variable name v to the member of *Term* that represents the term v: $\texttt{var}(V)$. The subtype of *Term* used for the carrier is simply the subtype containing those members of *Term* built up using only these constructors. This means that it contains only those members of the *Term* type that represent terms in which every subterm is a variable injection, or the term *false*, or one of the terms $\mathit{and}(a; b)$, $\mathit{or}(a; b)$, and $\mathit{implies}(a; b)$. *WffInd* is instantiated using a recursive function that destructures its final, *Term* valued, argument w, and applies one of its earlier arguments to the appropriate subobjects of w. It decides which of these functions to use based on the leading operator of the term that w represents, using a case operator like that of ML. The correct arguments to use are determined by the axioms of the *WFF* abstract type. As was the case for multisets, the axiom is most easily instantiated by theorem proving and extraction.

```
⟨
    $PropTerm$,
    $\lambda V.$var$(V)$,
    False(),
    $\lambda A.\lambda B.$And$(A; B)$,
    $\lambda A.\lambda B.$Or$(A; B)$,
    $\lambda A.\lambda B.$Implies$(A; B)$,
    $\lambda P.\lambda hv.\lambda hf.\lambda ha.\lambda ho.\lambda hi.\lambda w.\mathbf{Y}$
      $(\lambda f.\lambda w.$case $LeadingOperator(w)$ of
        opid{$false$} $\to hf$
        opid{$and$} $\to$
          $ha(STerm(1; w))(STerm(2; w))(f(STerm(1; w)))(f(STerm(2; w)))$
        opid{$or$} $\to$
          $ho(STerm(1; w))(STerm(2; w))(f(STerm(1; w)))(f(STerm(2; w)))$
        opid{$implies$} $\to$
          $hi(STerm(1; w))(STerm(2; w))(f(STerm(1; w)))(f(STerm(2; w)))$
        _ $\to hv(NthParameterObject(1; w)))$
      $(w)$,
    $\lambda P.\lambda hv.\lambda hf.\lambda ha.\lambda ho.\lambda hi.$
      $\langle \lambda v.axiom, axiom, \lambda a.\lambda b.axiom, \lambda a.\lambda b.axiom, \lambda a.\lambda b.axiom \rangle$
  ⟩
```

Figure 5. An instantiation of the *WFF* ADT.

6.4. PROOFS

The *PROOF* abstract data type is quite a bit looser; in particular the carrier may contain objects that cannot be built with the constructors. We do have certain constraints. First, we want to instantiate the carrier with a subtype of *Proof*. For practical reasons, this subtype should include only the representations of complete, reflection-free proofs. Second, if p is a member of our carrier type, $Hyps(p)$ is the empty multiset, and $Concl(p)$ represents the formula w, then p ought to represent a proof of w. Thus, *Hyps* and *Concl* should return an object determined by the hypotheses and conclusion of the top-level goal of the proof represented by their argument, a member of the Nuprl *Proof* type.

As a first try, we attempt to instantiate the carrier with the subtype of *Proof* that contains representations of all the complete, reflection-free proofs. We instantiate *Concl* with a function that is defined so that if P is an element of *Proof* represents a proof p, then $Concl(P)$ is an element of *Term* that represents the conclusion of the topmost goal sequent of p. Likewise, *Hyps* is defined so that $Hyps(P)$ is an element of *Term list* that represents the hypotheses of the topmost goal sequent of p.

This instantiation isn't quite correct for a number of reasons. First, the instantiation of *Concl* must always return a member of the carrier of the instantiation of

```
$PropTerm \equiv \{t : Term \mid IsPropForm(t)\}$

$IsPropForm(t) \equiv$
  let fun $IsPropForm\ t =$
    case $LeadingOperator(w)$ of
      $\texttt{opid}\{false\} \rightarrow NumArgs(w) = 0 \wedge NumParams(w) = 0$
      $\texttt{opid}\{and\}, \texttt{opid}\{or\}, \texttt{opid}\{implies\} \rightarrow$
        $NumArgs(w) = 2 \wedge NumBindings(1; w) = 0$
          $\wedge NumBindings(2; w) = 0 \wedge NumParams(w) = 0$
          $\wedge IsPropForm(STerm(1; w)) \wedge IsPropForm(STerm(2; w))$
      $\texttt{opid}\{\text{var}\} \rightarrow$
        $NumArgs(w) = 0 \wedge NumParams(w) = 1$
          $\wedge NthParameterFamily(1; w) = \texttt{ifid}\{\text{var}\} \in IFName$
      $\_ \rightarrow false$
  in $IsPropForm\ t$
```

Figure 6. The *PropTerm* type.

WFF, that is, of *PropTerm*. The easiest solution to this problem is to restrict our instantiation of the carrier to include only those members of the *Proof* type that have propositional formulas as their topmost conclusion. Second, the instantiation of *Hyps* must return a list of *PropTerm*s, but instead returns a list of triples consisting of a *Varname*, a *Term*, and a Boolean flag. We would like to restrict our proofs so that the *Term* component of this triple is always propositional, and have our instantiation of *Hyps* return the list made up of the *Term* component of each hypothesis in order. Unfortunately, this is too restrictive; it makes the proof constructors impossible to instantiate. Instead, we make use of the function that returns a list containing the *Term* component of those hypotheses of the proof's topmost goal that have propositional *Term* components, and ignores the other hypotheses completely.

With these instantiations of the *Hyps* and *Concl* functions fixed, we turn our attention to the instantiation of the constructors. These elements of the abstract data type specify functions that take zero or more elements of the carrier, that is of some subtype of the Nuprl *Proof* type, take zero or more other arguments, and produce a member of the Nuprl *Proof* type. Our strategy in implementing these functions is to produce a *Proof* that contains the argument proofs as subproofs pieced together using a small number of new inference steps to yield a proof with an appropriate toplevel goal.

Our task is easiest if there are as few extraneous hypotheses as possible, and if the hypotheses that are present obey certain conventions. Thus, we again restrict the elements of our carrier. We allow only those elements of *Proof* that represent proofs in which the topmost goal has a hypothesis list in which no hypothesis is hidden. We only allow hypotheses in which the type is either a propositional formula or the term *Prop*. If a hypothesis has a propositional formula as its type, it must bind the variable $\%i$, where i is the integer index of the hypothesis. In a given sequent's hypothesis list, all hypotheses with *Prop* as their type must occur before any hypothesis that has a propositional formula as its type. All sequents are required to be closed. And there should be as few hypotheses with *Prop* as their type as possible. The last two conditions simply say that a hypothesis x : *Prop* is present if and only if x appears in one of the propositional formulas of the sequent. From here, the instantiation of the constructors is simply a matter of working out the exact sequence of rules necessary, and accounting for the differences between Nuprl proofs and our abstract propositional proofs.

There are several differences between Nuprl proofs and abstract propositional proofs. Easiest to handle is the fact that Nuprl's inference rules require that hypotheses be referenced by their index rather than by their contents. This difference is easily handled by using a function that maps a hypothesis list H and a propositional formula w to the index of the first hypothesis of H that matches w. Also easy to handle is the fact that some of Nuprl's rules leave us with extra hypotheses in our subgoals. This is solved by always using instances of the thinning rule immediately after invocations of these rules to remove the unwanted hypotheses. Somewhat more complex is the fact that certain of Nuprl's rules generate additional proof obligations, which require us to prove that propositions are well formed. This is necessary because Nuprl's logic is sufficiently rich to make this question undecidable. In propositional logic, however, well-formedness is simply a syntactic property. Fortunately, it is simple to write a function that for any propositional formula w returns a proof that w is well formed. We use this function to satisfy these proof obligations. Most difficult is the fact that the order of hypotheses is significant in Nuprl, but not in our abstract propositional logic. Moreover, while abstract propositional logic builds proofs from the bottom up, Nuprl is set up to build them top down. What this means is that Nuprl's inference rules often produce subgoals with hypotheses in the wrong order. Our solution to this problem is to produce a tactic that given a proof of $H \vdash C$, and a permutation H' of H, produces a proof of $H' \vdash C$ consisting of a sequence of hypothesis reordering steps followed by the original proof. This is made slightly more complicated by Nuprl's restriction that a variable may only be bound once in a hypothesis list, and by our naming convention for hypothesis variables. There is a tradeoff here between abstractness and efficiency, and in fact implementation work is made easier if the types are treated less abstractly.

Thus, our instantiation of *PROOF*(*Varname*; *VEq*; *Wff*; *Multiset*) is a 12-tuple. The first three elements are *PropProof*, which is defined below, and the two functions $\lambda p.\mathit{Hyps}(\mathit{TopGoal}(p))$ and $\lambda p.\mathit{Concl}(\mathit{TopGoal}(p))$.

DEFINITION 23 (*PropProof*)

$$\begin{aligned}&\mathit{PropProof} \equiv\\ &\quad \{p : \mathit{Proof} \mid \mathit{complete}(p) \wedge \mathit{reflection\_free}(p)\\ &\qquad \wedge\ \mathit{IsClosedPropSequent}(\mathit{TopGoal}(p))\}\end{aligned}$$

The remaining elements are simply functions that build up the appropriate superstructure around their arguments. We give a couple of examples; the remainder are similar. *Hypothesis* is instantiated by

$$\lambda h.\lambda c.\langle \mathit{MakeNuprlSequent}(h; c), \mathit{hypothesis\_rule}(\mathit{index}(c; h)), \mathit{nil}\rangle.$$

This function simply constructs a single node proof consisting of the sequent $h \vdash c$, justified by the Nuprl hypothesis rule. Note the use of *index* to find the index of the hypothesis c in h, which is required as a parameter to the Nuprl inference rule.

A more intricate example is the following function used to instantiate *ImpRight*.

```
$\lambda a, pa.$
   let  $t = \mathit{TopGoal}(pa)$
        $h = \mathit{Hyps}(t)$
        $c = \mathit{Concl}(t)$
        $m = a; h)$
        $h' = \mathit{delete\_nth}(m; h)$
   in
      $\langle \mathit{MakeNuprlSequent}(h'; \mathtt{Implies}(a; b)),$
       $\mathit{implies\_intro\_rule}(\mathtt{Prop}, \mathit{tempvar}(m)),$
       $[\mathit{permute\_hyps}(pa; h'@\mathit{MakeNuprlHyp}(\mathit{tempvar}(m); a)]),$
       $\mathit{prop\_auto}(a; h)]\rangle$
```

Here, `Prop` represents the term $\mathtt{U}_i$ (which is the term to which *Prop* expands) and $\mathit{tempvar}(m)$ represents the variable name $\%n$. The function *permute_hyps* is the function that takes a proof representation p and a hypothesis list representation h, and produces a proof representation p' with the same hypotheses and conclusion as p, but with the hypotheses rearranged to match h. And the function *prop_auto* takes a representation of propositional formula a, and a representation of hypothesis list h, which includes a binding of the form $x : \mathit{Prop}$ for each free variable of a, and returns a representation of a proof of $h \vdash a \in \mathit{Prop}$.

Let $H, a \vdash b$ be the topmost goal sequent of pa. We construct a proof consisting of a representation of the sequent $H \vdash a \to b$, an invocation of the Nuprl implies introduction rule, and two subproofs. The implies introduction rule takes two arguments: one indicates the universe level at which to do the introduction, and the other provides a variable name for the new hypothesis generated by the rule. The first subproof is, essentially, pa. However, because its topmost goal's hypotheses must have the hypothesis a last, while pa's topmost goal's hypotheses may have a in any position, some rearrangement of hypotheses may be required. The second subproof is a proof that a is well formed and is constructed by the function *prop_auto*.

When we instantiate *MULTISET*, *WFF* and *PROOF* like this, our proof of the completeness theorem produces a function f that maps elements of the type *Term* either to elements of the type *Proof*, or to Kripke models. Moreover, if C represents the propositional formula c, then if $F(C)$ is an element p of the type *Proof*, p represents a proof of the sequent

$$x_1 : \mathit{Prop}, \ldots x_n : \mathit{Prop} \ \vdash \ c$$

in which the variable names x_i are exactly the names of the free variables of c, in some order. Furthermore, we know that if c is provable, $f(C)$ will always be an element of *Proof*.

Using f we can construct a function that maps certain terms S that represent provable sequents s to terms that represent a proof of s. For our function to be applicable, s must, in addition to being provable, be a closed sequent of the form

$$x_1 : \mathit{Prop}, \ldots x_n : \mathit{Prop} \ \vdash \ c,$$

c must be propositional, and the variable names x_i must all name free variables of c. To construct our function, we first project out the component of S that represents c to produce a term C. We apply f to C. Because s is provable, we obtain an element P of the Nuprl type *Proof*. P represents a proof of the sequent

$$x'_1 : \mathit{Prop}, \ldots x'_n : \mathit{Prop} \ \vdash \ c,$$

in which $x'_1, \ldots, x'_n$ is a permutation of $x_1, \ldots, x_n$. Using the hypothesis permuting tactic described earlier, we can produce a term that represents the proof of s, as required.

We can also implement a function that decides if a term like S, which represents a certain sort of sequent s, as described above, represents a provable sequent. Simply project out the component C of S that represents C, apply f to C, and return true if f returns a proof, and false if f returns a Kripke model.

6.5. APPLIED FORMULAS

What we have just shown is the existence of a procedure mapping a particular class of Nuprl sequents to either their proof or a counter-model. The sequents must have a pure propositional formula as their conclusion, and must have only declarations of propositional variables as hypotheses. We would really like to generalize this procedure to allow applied propositional formulas, in which the class of atomic formulas is extended to include terms we wish to consider as constants. We call these extended formulas *applied formulas*, after Curry [10]. Our goal is to produce a proof of these sequents if they are propositionally provable, that is, provable without regard to the interpretation of the atomic formulas. It turns out that producing such a procedure is very easy: it requires only very minor changes to the instantiations given above.

Most importantly we must extend the class of terms. The easiest way to do this is to change our instantiation of *Varname*. We instantiate it with an appropriate subtype of the *Term* type. This type must not contain representatives of terms formed using the propositional connectives *and*, *or*, *implies* or *false*. This restriction ensures that propositional formulas may easily be distinguished from non-propositional formulas. It is logically unrestrictive since we can define an operator

$$\mathit{guard}(x) \equiv x$$

to protect propositional formulas we wish to treat as atomic. Variable equality is instantiated as *Term* equality (or rather, by a decision procedure for term equality, since that is what is required by the abstract type), and variable injection is instantiated by the identity function. We must also change the body of the final case in the instantatiation of *WffInd* to be $hv(w)$ instead of $hv(\mathit{NthParameterObject}(1; w))$. Everything else in the instantiation of *WFF* remains the same—we deliberately took variables to be the default case when we defined the instantiation of *WffInd* above, so that we could easily change the instantiation of *Varname* without disruption.

Second, we must allow this richer class of terms to appear as propositional formulas in proofs. The only other change that needs to be made is in the procedure that maps propositional formulas to proofs that they are propositional formulas. Before, the variable case simply required that we invoke the hypothesis that bound the variable with type *Prop*. Now, nontrivial work is required. The easiest way to extend the system is to change the restrictions on the hypotheses. Now, three kinds of hypotheses may appear: (1) variable bindings, which may now bind variables of any type, and which are required to ensure that the sequent is closed; (2) hypotheses of the form $t \in \mathit{Prop}$; and (3) propositional hypotheses, as before. We further require that any atomic formula used either in a propositional hypothesis or in the conclusion must be declared as propositional by a hypothesis of the first two sorts. In the atomic case, the propositional well-formedness function simply needs to invoke the appropriate hypothesis. *Nothing* else need be changed.

7. Reflection

The Nuprl reflection mechanism includes types that represent the various classes of primitive Nuprl objects, in particular, terms, sequents, and proofs. It also includes a rule that allows us to justify a sequent s by showing that the type used to represent proofs, the *Proof* type, has an element that represents a proof of s. We call this rule the reflection rule. It is parameterized by a Nuprl term, and a positive integer. The term parameter is used to generate subgoals, and the integer parameter, called the reflection level, ensures that the reflection rule does not allow circular reasoning.

The reflection rule can be used, with a subgoal generator f and a reflection level n, to refine any sequent g. f is a applied to the canonical representation of g. The resulting term is evaluated. If evaluation produces the left injection of a list of sequent representations, the refinement succeeds; otherwise, it fails. Suppose the refinement succeeds, and that the list of sequent representations is of length m; $m + 1$ subgoals are generated. The second and subsequent subgoals are exactly those sequents represented by the elements of the list. Thus, they can be thought of as generated by f. The first subgoal justifies the use of the reflection subgoal. It always has the same basic syntactic form. It has a single hypothesis asserting that the application of f to the representation of g evaluates to a left injection. Its conclusion asserts the existence of a member P of the *Proof* type satisfying three conditions. First, the proof P represents must have g as its top-level goal. Second, this proof must have as its open assumptions the remaining subgoals generated by the reflection rule. Third, any instance of the reflection rule occurring in this proof must have a reflection level smaller than n. Schematically,

$$\begin{array}{l}
H \vdash G \ \texttt{by reflection}\ f, n \\
\quad isl(f(\ulcorner H \vdash G \urcorner)) \vdash \exists P : Proof.root(P) = \ulcorner H \vdash G \urcorner \in Sequent \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \& \, frontier(P) = outl(f(\ulcorner H \vdash G \urcorner)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \in \ Sequent\ list \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \& level(P) < n \\
\quad H_1 \vdash G_1 \\
\qquad \vdots \\
\quad H_m \vdash G_m
\end{array}$$

provided $f(\ulcorner H \vdash G \urcorner)$ evaluates to a term $inl([S_1, \ldots, S_m])$, and S_i represents the sequent $H_i \vdash G_i$, for each i between 1 and m.

Suppose we know of some computable function that takes a sequent (or more accurately, the representation of one) and returns the Boolean value *true* only if the sequent is provable. Suppose also that the function operates very quickly, and that it returns *true* sufficiently often. It would facilitate theorem proving if we could use this function as a new inference rule with which to refine sequents. If the function returns *true* the sequent is justified without further ado; if it isn't, the refinement fails. Not much has been lost in the latter case, since the function is assumed to run quickly. In the remainder of this section, we sketch the use of the reflection rule to do just this.

First, we must write the decision procedure as a Nuprl term. The abstraction mechanism is sufficiently rich to ensure that this is not significantly more onerous than writing it in any other programming language. Call the resulting Nuprl object *dec_proc*. We can prove that this function has the properties we expect of it: namely, that it map sequents to *true* or *false*, and that if it maps a sequent to *true*, the sequent is provable in Nuprl. We state and prove these theorems:

$$dec\_proc \in Sequent \rightarrow bool$$

$$\begin{array}{l}
\forall s : Sequent.dec\_proc(s) = true \in bool \\
\quad \Rightarrow \exists p : Proof.complete(p) \& root(p) = s \in Sequent
\end{array}$$

In practice, two modifications are actually necessary. Often, it is inconvenient to arrange that *dec_proc* behave appropriately on all sequents, but it is possible to ensure that it works on some easily recognizable subclass of sequents. Thus, we replace all instances of Sequent above with the type $\{s : \mathit{Sequent} \mid \Phi(s)\}$, where Φ is some easily decidable predicate on sequents. Also, as becomes apparent when we attempt to use the latter theorem, we need a bound on the level of reflection used in the proof represented by p. These refinements are particularly important because we typically derive our decision procedures relative to an abstract description of a logic. This treatment deliberately obscures the fact that the object logic is to be embedded in Nuprl. Our decision procedures cannot be expected to check that their arguments are objects of the appropriate logic: they are developed assuming that the object logic is all there is. Thus, the predicate Φ is used to limit the application of the decision procedure to appropriate sequents. For the sake of concreteness, we take the bound on the level of reflection to be zero – thus requiring that the proof be reflection-free – but any bound would do. Thus, our second theorem is actually stated as

$$\begin{aligned}&\forall s : \{s : \mathit{Sequent} \mid \Phi(s)\}.\mathit{dec\_proc}(s) = \mathit{true} \in \{\mathit{true}, \mathit{false}\}\\ &\quad \Rightarrow \exists p : \mathit{Proof}.\mathit{complete}(p)\&\mathit{root}(p) = s \in \mathit{Sequent}\&\mathit{reflection\_free}(p)\end{aligned}$$

All we have required so far is that the function be implemented, and that our knowledge about its behavior be formalized. This is the only preparation that is required to use this function as an inference rule. With this done, we can use *dec_proc*, the reflection rule, and some simple, inexpensive reasoning (all wrapped up as a tactic) as a new inference rule. The cost of using the rule will be essentially the cost of applying *dec_proc*. System security is maintained: no unsound reasoning is permitted.

To safely use *dec_proc* as an inference rule on a sequent $H \vdash G$, we use the reflection rule. The reflection level is 1 (if we had chosen a nonzero upper bound on the reflection level, in our refined statement of the theorem characterizing *dec_proc*, we would have to choose a number bigger than that bound instead). The subgoal generator is a function f that, when applied to the representation S, of a sequent returns $\mathit{inl}(\mathit{nil})$ if $\Phi(S)$ holds and $\mathit{dec\_proc}(S)$ returns *true*, and returns $\mathit{inr}(\mathit{nil})$ otherwise. This yields a single subgoal

$$\begin{aligned}&\mathit{isl}(f(\ulcorner H \vdash G\urcorner)) \vdash\\ &\quad \exists p : \mathit{Proof}.\mathit{root}(p) = \ulcorner H \vdash G\urcorner \in \mathit{Sequent}\\ &\qquad \& \mathit{frontier}(p) = \mathit{outl}(f(\ulcorner H \vdash G\urcorner)) \in \mathit{Sequent\ list}\\ &\qquad \& \mathit{level}(p) < 1\end{aligned}$$

We cut in the formula $\Phi(\ulcorner H \vdash G\urcorner)$, which we know is true because f returned successfully. It is assumed that the cut formula will also be easy to prove. This allows us to prove that $\ulcorner H \vdash G\urcorner$ belongs to the type $\{s : \mathit{Sequent} \mid \Phi(s)\}$. Because f returned successfully, our definition of f allows us to conclude that $\mathit{dec\_proc}(\ulcorner H \vdash G\urcorner)$ evaluates to *true*. Thus, the theorem we stated to characterize *dec_proc* holds. We use it to conclude that

$$\exists p : \mathit{Proof}.\mathit{complete}(p)\&\mathit{root}(p) = \ulcorner H \vdash G\urcorner \in \mathit{Sequent}\&\mathit{reflection\_free}(p).$$

Note that $\mathit{outl}(f(\ulcorner H \vdash G \urcorner))$ is *nil*, and $\mathit{reflection\_free}(p)$ is equivalent to $\mathit{level}(p) < 1$, so the remaining subgoal follows immediately.

The only nontrivial theorem proving is the running of f, and the proof that $\Phi(\ulcorner H \vdash G \urcorner)$ actually holds. The latter is assumed to be very easy, and in any case is necessary only to avoid the need for *dec_proc* to validate its input. The definition of f can be given as follows

$$f(s) = \text{if } \mathit{decide\_phi}(s) \&\& \mathit{dec\_proc}(s) = \mathit{true} \text{ then } \mathit{inl}(\mathit{nil}) \text{ else } \mathit{inr}(\mathit{nil}).$$

Here *decide_phi* is a decision procedure for Φ, which we assume is very fast. Thus, running f is roughly as expensive as running *dec_proc*.

8. Conclusions

A general problem-solving environment should allow extensions of both the information contained within it and the techniques available for manipulating that information. In the case of a theorem proving assistant, we should be able to add to both the store of knowledge (through libraries of proved theorems, lemmas, etc.) and the methods available for proof search (such as tactics and decision procedures). Since these methods are just special-purpose programs, verification questions arise when new methods are added to a system. Ideally, we would like to be able to prove properties of these proof techniques in the same way as we would prove other theorems in the system.

Reflection provides a uniform mechanism for applying metalevel reasoning to the object level. In this paper, we have described how reflection may be used to develop a verified decision procedure as the computational extract of a constructive proof. Implementation of this proof is ongoing; the development of a library of multiset functions and theorems about their properties has dominated this work. Since the only role of this library is to support the definition of the induction measure, the full power of constructive logic is not necessary, and we are exploring the possibility of using classical logic for such purposes. Current work also includes verification of the Sup-Inf decision procedure for Presburger arithmetic [31].

The authors would like to thank Jim Caldwell for careful reading of earlier drafts of this paper.

Appendix

A. Formal Definition of Forcing in Kripke Models

We wish to define two functions, $\mathit{forces}(V, W)$ and $\mathit{notforces}(V, W)$, where V is the type of variable names and W is an instance of the ADT $\mathit{WFF}(V)$. The type of these functions is

$$K : \mathit{Kripke\_model}(V) \to W.\mathit{Wff} \to K.T \to \mathit{Type}.$$

Mathematically, these functions are defined by induction on the construction of the formula:

$$
\begin{array}{lcll}
\mathit{forces}(K, s, \alpha) & = & \mathit{af}(s, \alpha) & (\alpha \in Atom) \\
 & & \mathit{false} & (\alpha = \mathit{false}) \\
 & & \mathit{forces}(K, s, \beta) \wedge \mathit{forces}(K, s, \gamma) & (\alpha = \beta \wedge \gamma) \\
 & & \mathit{forces}(K, s, \beta) \vee \mathit{forces}(K, s, \gamma) & (\alpha = \beta \vee \gamma) \\
 & & \forall s'.sRs' \rightarrow & \\
 & & \mathit{notforces}(K, s', \beta) \vee \mathit{forces}(K, s', \gamma) & (\alpha = \beta \rightarrow \gamma)
\end{array}
$$

$$
\begin{array}{lcll}
\mathit{notforces}(K, s, \alpha) & = & \neg \mathit{af}(s, \alpha) & (\alpha \in Atom) \\
 & & \mathit{true} & (\alpha = \mathit{false}) \\
 & & \mathit{notforces}(K, s, \beta) \vee \mathit{notforces}(K, s, \gamma) & (\alpha = \beta \wedge \gamma) \\
 & & \mathit{notforces}(K, s, \beta) \wedge \mathit{notforces}(K, s, \gamma) & (\alpha = \beta \vee \gamma) \\
 & & \exists s'.sRs' \wedge & \\
 & & \mathit{forces}(K, s', \beta) \wedge \mathit{notforces}(K, s', \gamma) & (\alpha = \beta \rightarrow \gamma)
\end{array}
$$

Several issues arise when interpreting these definitions in type theory, since the metalevel connectives are usually interpreted classically. However, in this setting we are dealing only with finite Kripke models, for which forcing is decidable. Thus, we may reformulate the definitions slightly. In type theory, using *WffInd* for induction on the structure of the formula, these definitions become

$$
\begin{array}{l}
\mathit{forces}(V, W) \equiv \\
\lambda K : \mathit{Kripke\_model}(V). \\
\quad W.\mathit{WffInd}(\lambda x.(K.T \rightarrow \mathit{Type})) \\
\quad (\lambda v.\lambda s.(K. \uparrow \{\mathit{af}(s, v)\})) \\
\quad (\lambda s. \uparrow \{\mathit{false}\}) \\
\quad (\lambda a, b, pa, pb.\lambda s.pa(s) \wedge pb(s)) \\
\quad (\lambda a, b, pa, pb.\lambda s.pa(s) \vee pb(s)) \\
\quad (\lambda a, b, pa, pb.\lambda s.\forall s' : K.T.(K.R(s, s') \rightarrow (pa(s') \rightarrow pb(s'))))
\end{array}
$$

$$
\begin{array}{l}
\mathit{notforces}(V, W) \equiv \\
\lambda K : \mathit{Kripke\_model}(V). \\
\quad W.\mathit{WffInd}(\lambda x.(K.T \rightarrow \mathit{Type})) \\
\quad (\lambda v.\lambda s.\mathit{not}(K. \uparrow \{af(s, v)\})) \\
\quad (\lambda s. \uparrow \{\mathit{true}\}) \\
\quad (\lambda a, b, pa, pb.\lambda s.pa(s) \vee pb(s)) \\
\quad (\lambda a, b, pa, pb.\lambda s.pa(s) \wedge pb(s)) \\
\quad (\lambda a, b, pa, pb.\lambda s.\exists s' : K.T.(K.R(s, s') \\
\quad \wedge (\mathit{forces}(V, W)(K)(a)(s') \wedge pb(s'))))
\end{array}
$$

Recall that $\uparrow \{p\}$ maps the Boolean value *true* to an inhabited type, and the value *false* to an uninhabited type. This allows us to interpret Boolean truth values as intuitionistic truth values, i.e. as members of *Type*.

B. Details of the Completeness Proof

We shall now consider a few cases of the induction. All the cases of the inner induction are very similar; we shall present two so that the pattern becomes clear. We shall also describe the case which reduces the outer inductive measure.

Recall that in the inner induction of the proof, we have that $\exists N \in S. \neg \mathit{Node\_complete}(N)$ and $i_2(S) > 0$.

Also recall that $\neg \mathit{Node\_complete}(N)$ implies that there are formulas $a, b : W.\mathit{Wff}$ such that one of the following holds:

$$\begin{array}{ccl}
W.\mathit{And}(a, b) \in \pi_1(N) & \text{but} & a \notin \pi_1(N) \ \vee \ b \notin \pi_1(N) \\
W.\mathit{Or}(a, b) \in \pi_1(N) & \text{but} & a \notin \pi_1(N)\, \&\, b \notin \pi_1(N) \\
W.\mathit{Implies}(a, b) \in \pi_1(N) & \text{but} & a \notin \pi_2(N)\, \&\, b \notin \pi_1(N) \\
W.\mathit{And}(a, b) \in \pi_2(N) & \text{but} & a \notin \pi_2(N)\, \&\, b \notin \pi_2(N) \\
W.\mathit{Or}(a, b) \in \pi_2(N) & \text{but} & a \notin \pi_2(N) \ \vee \ b \notin \pi_2(N)
\end{array}$$

We will give two cases as examples; the other cases proceed similarly.

Case: $W.\mathit{And}(a, b) \in \pi_1(N)$ but $a \notin \pi_1(N) \ \vee \ b \notin \pi_1(N)$

Suppose the first case holds. Then we have $W.\mathit{And}(a, b) \in \pi_1(N)$ but either $a \notin \pi_1(N)$ or $b \notin \pi_1(N)$. Let $N' = \langle \{a, b\} \cup \pi_1(N), \pi_2(N)\rangle$. (Formally, this is really $\langle \{a\} \cup (\{b\} \cup \pi_1(N)), \pi_2(N)\rangle$.) Let S' be S with N replaced everywhere by N'. Since a and b are subformulas of a formula in S, it follows that $SF(S') = SF(S)$. Furthermore, since we know that one of a and b was not in $\pi_1(N)$ but both are now in $\pi_1(N')$, we have $|SF(S') - \pi_1(N')| < |SF(S) - \pi_1(N)|$. Since no other node has changed, we have $i_2(S') < i_2(S)$.

Note that the requirement that systems be eligible guarantees that $i_1(S') = i_1(S)$. The only way this measure could be reduced is if the new node were equal to one already in the system. But since adding a formula to an open node cannot make it equal to another node if it wasn't contained in it beforehand, this is impossible if the first system is eligible. Similarly, because the first system is eligible, the new system is also, since there is still at most one open node, and it cannot be contained in a complete node in the system.

We then apply the induction hypothesis to this new system. The result is either

$$\begin{array}{l}
\exists N : \mathit{NODE}.N \in S' \& \\
\quad \exists P : \mathit{Proof.Proof}.(\mathit{Proof.Hyps}(P) = \pi_1(N) \& \mathit{Proof.Concl}(P) \in \pi_2(N))
\end{array}$$

or

$$\begin{array}{l}
\exists K : \mathit{Kripke\_model}(V) \\
\quad \exists f : \{f : \{N : \mathit{NODE} \mid N \in S'\} \to K.T \mid \\
\quad\quad \forall N : \{N : \mathit{NODE} \mid N \in S'\} \\
\quad\quad\quad \forall w : W.\mathit{Wff}.w \in \pi_1(N) \to \mathit{forces}(K, w, f(N))\& \\
\quad\quad\quad\quad w \in \pi_2(N) \to \mathit{notforces}(K, w, f(N))\}
\end{array}$$

Subcase: S' is provable. In the first case, let M be the node in S' which is proved. If $M \neq N'$, then the node M is in the original system S, so we have immediately that there exists a provable node in S.

If $M = N'$, then, given that P proves N', a proof of N is

$$\mathit{Proof.AndLeft}(a, b, P).$$

Observe that, by definition of the rule *Proof.AndLeft*, the formulas a and b added to $\pi_1(N)$ to make N' are removed by application of the rule. Thus, we know that the hypotheses set of P is $\pi_1(N)$. Since $\pi_2(N) = \pi_2(N')$, and the conclusion of the proof does not change, we have a proof of N, as desired.

Subcase: S' has a countermodel. If, instead, we have a countermodel K and function f for S', take the same K and define $f(N) = f(N')$. Since $\pi_1(N) \subset \pi_1(N')$, and state $f(N')$ in K forces everything in $\pi_1(N')$, we must have that state $f(N)$ forces everything in $\pi_1(N)$.

Case: $W.And(a, b) \in \pi_2(N)$ but $a \notin \pi_2(N) \,\&\, b \notin \pi_2(N)$.

As another example, one which corresponds to an or-branch in the tableau, consider the case where $W.And(a, b) \in \pi_2(N)$ but $a \notin \pi_2(N) \,\&\, b \notin \pi_2(N)$.

We create new nodes $N_0 = \langle \pi_1(N), \{a\} \cup \pi_2(N)\rangle$ and $N_1 = \langle \pi_1(N), \{b\} \cup \pi_2(N)\rangle$.

Let S_0 be the system S with N replaced by N_0 and S_1 be S with N replaced by N_1.

By the same argument in the previous case, $SF(S_0) = SF(S_1) = SF(S)$, and since neither a nor b was in $\pi_2(N)$, we have $|SF(S_0) - \pi_2(N_0)| < |SF(S) - \pi_2(N)|$ and $|SF(S_1) - \pi_2(N_1)| < |SF(S) - \pi_2(N)|$. Since no other nodes have changed, we have $i_2(S_0) < i_2(S)$ and $i_2(S_1) < i_2(S)$.

As in the previous case, the requirement that systems be eligible ensures that $i_1(S_0) = i_1(S_1) = i_1(S)$. Also, it is easy to check that these new systems are eligible using the fact the old system was eligible.

We now apply the inductive hypothesis to both S_0 and S_1. There are a number of cases to consider.

Subcase: S_0 or S_1 has a countermodel. If we have a Kripke countermodel for either

S_0 or S_1, the same model is a countermodel for S, by an argument similar to the one above. The value of $f(N)$ for the system S should be the value of $f(N_0)$ or $f(N_1)$; this ensures that all formulas in $\pi_1(N)$ are forced and no formula in $\pi_2(N)$ is forced.

Subcase: S_0 and S_1 are provable. If the inductive hypothesis applied to either S_0 or S_1 produces a proof of a node which is not N_0 or N_1, the same proof satisfies the theorem for S, since that node will also be in S.

If either application of the inductive hypothesis produces a proof of N_0 or N_1 such that the formula proved is not a or b, that formula must be in N and the same proof satisfies the theorem for S.

The last possibility is that the proofs produced by the inductive hypothesis are proofs of a and b, i.e. we have P_0 with $\mathit{Proof.Hyps}(P_0) = \pi_1(N_0)$ and $\mathit{Proof.Concl}(P_0) = a$, and P_1 with $\mathit{Proof.Hyps}(P_1) = \pi_1(N_1)$ and $\mathit{Proof.Concl}(P_1) = b$. Since

$$\pi_1(N) = \pi_1(N_0) = \pi_1(N_1),$$

a proof of N is simply $\mathit{Proof.AndRight}(P_0, P_1)$.

Reducing the Outer Inductive Measure

We now consider the outer induction. The outer inductive measure is reduced when the inner inductive cannot be further reduced, but the system is not yet complete. In this case, we have for all N in S that *Node_complete*(N), but not *System_complete*(S). Since we do not have *System_complete*(S), we must have

$$\begin{aligned}&\exists N : (\mathit{Wffset} \times \mathit{Wffset}) \exists a, b : W.\mathit{Wff} N \in S \& \\ &\quad W.\mathit{Implies}(a, b) \in \pi_2(N) \,\&\, \forall N'. N' \in S \rightarrow a \notin \pi_1(N') \vee b \notin \pi_2(N').\end{aligned}$$

Let $N' = \langle \{a\} \cup \pi_1(N), \{b\} \rangle$ and let $S' = \{N'\} \cup S$.

Since the formulas in the new node N' are subformulas of those in the system S, we have $SN(S) = SN(S')$. Therefore, $i_1(S') = |SN(S) - S'| < |SN(S) - S| = i_1(S)$, and the outer induction measure is reduced when this node is added. Furthermore, since there were no open nodes in S, there is at most one open node in S'. Since the new node N' cannot be contained in any other node in S', we have that S' is eligible and we can apply the inductive hypothesis to it.

When we apply the inductive hypothesis, as before, we have either

$$\begin{aligned}&\exists N : \mathit{NODE}. N \in S' \& \\ &\quad \exists P : \mathit{Proof.Proof.Proof.Hyps}(P) = \pi_1(N) \& \mathit{Proof.Concl}(P) \in \pi_2(N)\end{aligned}$$

or

$$\begin{aligned}&\exists K : \mathit{Kripke\_model}(V) \\ &\quad \exists f : \{f : \{N : \mathit{NODE} \mid N \in S\} \rightarrow K.T \mid \\ &\qquad \forall N : \{N : \mathit{NODE} \mid N \in S'\} \\ &\qquad\quad \forall w : W.\mathit{Wff}. w \in \pi_1(N) \rightarrow \mathit{forces}(K, w, f(N)) \& \\ &\qquad\qquad w \in \pi_2(N) \rightarrow \mathit{notforces}(K, w, f(N))\}\end{aligned}$$

In the first case, let M be the node in S' which is proved. If $M \neq N'$, then M is in S and the same proof suffices for S.

If $M = N'$, then the proof of N in S is simply

$$\mathit{Proof.ImpRight}(a, P)$$

If a countermodel K for S' is created, the same model K is a countermodel for S as well. The only difference is that the node N' is removed from the domain of the function f.

References

1. Aagaard, M. and Leeser, M.: Verifying a logic synthesis tool in Nuprl, in Gregor Bochmann and David Probst (eds), *Proceedings of the 4th International Workshop on Computer-Aided Verification*, Springer-Verlag, June 1992, pp. 69–81.
2. Aitken, W.: A formal introduction to the lambda calculus, Computer Science Dept., Cornell University, 1993.
3. Aitken, W. and Constable, R. L.: Reflecting on Nuprl Lessons 1–4, Technical Report, Cornell University, Computer Science Dept., 1992. Internal report.
4. Allen, S., Constable, R., Howe, D. and Aitken, W.: The semantics of reflected proof, in *Proc. of Fifth Symp. on Logic in Comp. Sci.*, IEEE, June 1990, pp. 95–197.
5. Bledsoe, W. W.: A new method for proving certain Presburger formulas, *Fourth Intl. Joint Conf. on A.I.*, Tbilisi, USSR, September 1975.
6. Chan, T.: An algorithm for checking PL/CV arithmetic inferences, in G. Goos and J. Hartmanis (eds), *An Introduction to the PL/CV Programming Logic*, Lecture Notes in Computer Science 135, Springer-Verlag, 1982, pp. 227–264.
7. Church, A.: *Introduction to Mathematical Logic, Vol. I*, Princeton University Press, 1956.
8. Constable, R. L. et al.: *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
9. Coquand, T. and Huet, G.: The calculus of constructions, *Information and Computation* **76** (1988), 95–120.
10. Curry, H. B.: *Foundations of Mathematical Logic*, Dover, 1977.
11. Fitting, M.: *Intuitionistic Logic, Model Theory, and Forcing*, North-Holland, Amsterdam, 1969.
12. Fitting, M.: *First-Order Logic and Automated Theorem Proving*, Springer-Verlag, 1990.
13. Gentzen, G.: Investigations into logical deduction (1934), in M. Szalo (ed.), *The Collected Papers of Gerhard Gentzen*, North-Holland, Amsterdam, 1969.
14. Gordon, M.: HOL: A machine oriented formalization of higher order logic, Technical Report 68, Cambridge University, 1985.
15. Gordon, M., Milner, R. and Wadsworth, C.: *Edinburgh LCF: A Mechanized Logic of Computation*, Lecture Notes in Computer Science 78, Springer-Verlag, New York, 1979.
16. Hickey, J.: Nuprl–light: An implementation framework for higher-order logics, in W. McCune (ed.), *Automated Deduction – CADE-14*, Lecture Notes in Artificial Intelligence 1249, Springer-Verlag, 1997, pp. 395–399.
17. Jackson, P. B.: *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*, Ph.D. thesis, Cornell University, 1994.
18. Jenks, R. D. and Sutor, R. S.: *Axiom: The Scientific Computation System*, Springer-Verlag, New York, 1992.
19. Kozen, D.: *Complexity of Finitely Presented Algebras*, Ph.D. thesis, Computer Science Department, Cornell University, Ithaca, New York, 1977.

20. Kozen, D., Ben-Or, M. and Reif, J.: The complexity of elementary algebra and geometry, in *Proc. 16th ACM Symp. Theory of Comput.*, 1984, pp. 457–464. Invited special issue *J. Comput. Syst. Sci.* **32**(2) (1985), 251–264.
21. Luo, Z.: Program specification and data refinement in type theory, in *Proc. Fourth International Joint Conference on the Theory and Practice of Software Development (TAPSOFT)*, 1991.
22. Maharaj, S. and Gunter, E.: Studying the ML module system in HOL, in T. F. Melham and J. Camilleri (eds), *Higher Order Logic Theorem Proving and Its Applications*, Lecture Notes in Computer Science 859, Springer-Verlag, 1994, pp. 346–361.
23. Martin-Löf, P.: Constructive mathematics and computer programming, in *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, North-Holland, Amsterdam, 1982, pp. 153–175.
24. McAllester, D. A.: *ONTIC: A Knowledge Representation System for Mathematics*, MIT Press, Cambridge, Mass., 1989.
25. Milner, R.: Elements of interaction, *Comm. Assoc. Comput. Mach.* **36**(1) (1993), 78–89.
26. Owre, S., Rushby, J. M. and Shankar, N.: Pvs: A prototype verification system, in Deepak Kapur (ed.), *11th International Conference on Automated Deduction (CADE), Saratoga, NY, 1992*, Lecture Notes in Artificial Intelligence 607, Springer-Verlag, 1992, pp. 748–752.
27. Paulson, L. C.: A formulation of the simple theory of types (for Isabelle), in P. Martin-Löf and G. Mints (eds), *Proc. Int. Conference on Computer Logic*, Lecture Notes in Computer Science 417, Springer-Verlag, New York, 1988, pp. 246–274.
28. Pollack, R.: On extensibility of proof checkers, in Dybjer, Nordstrom, and Smith (eds), *Types for Proofs and Programs: International Workshop TYPES'94, Båstad, June 1994, Selected Papers*, Lecture Notes in Computer Science 996, Springer-Verlag, 1995.
29. Rushby, J., von Henke, F. and Owre, S.: An introduction to formal specification and verification using EHDM, Technical Report CSL-91-2, Computer Science Laboratory, SRI International, February 1991.
30. Shostak, R. E.: A practical decision procedure for arithmetic with function symbols, *J. Assoc. Comput. Mach.* **26** (1979), 351–360.
31. Shostak, R. E.: On the SUP-INF method for proving Presburger formulas, *JACM* **24**(4) (1977), 529–543.
32. Underwood, J.: A constructive completeness proof for the intuitionistic propositional calculus, Technical Report 90-1179, Cornell University, 1990.
33. Underwood, J.: Typing abstract data types, in Egidio Astesiano, Gianna Reggio, and Andrzej Tarlecki (eds), *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science 906, Springer-Verlag, 1994, pp. 437–452.
34. VanInwegen, M. and Gunter, E. L.: HOL-ML, in J. Joyce and C. Seger (eds), *Higher Order Logic Theorem Proving and Its Applications*, Lecture Notes in Computer Science 780, Springer-Verlag, 1994, pp. 61–73.
35. Wallen, L. A.: *Automated Deduction in Non-Classical Logics*, MIT Press, 1990.
36. Wolfram, S.: *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, 1988.
37. Zippel, R.: The Weyl computer algebra substrate, Technical Report TR 90-1077, Computer Science Dept., Cornell University, Ithaca, NY, 1990.