Extracting Propositional Decidability: A proof of propositional decidability in constructive type theory and its extracted program.

James L. Caldwell

March 21, 1997

Abstract

This paper describes a formal constructive proof of the decidability of a sequent calculus presentation of classical propositional logic. The Nuprl theories and proofs reported on here are part of a larger program to safely incorporate formally justified decision procedures into theorem provers. The proof is implemented in the Nuprl system and the resulting proof object yields a "correct-by-construction" program for deciding propositional sequents. In the case the sequent is valid, the program reports that fact; in the case the sequent is falsifiable, the program returns a falsifying assignment. Also, the semantics of the propositional sequents is formulated here in Kleene's strong three-valued logic which both: agrees with the standard two valued semantics; and gives finer information in case the proposition is falsifiable.

Contents

1 Introduction

In this paper, a Nuprl proof of the decidability of a sequent calculus presentation of propositional logic is presented. Nuprl is an implementation of a constructive type theory; as a result of the constructivity, and the design of the system, proofs yield programs in the form of terms of the untyped lambda calculus. Here, Nuprl is used as meta-theory [1, 4, 24] to formalize the syntax and semantics of a sequent presentation of classical propositional logic. Decidability for this embedded formal system is proved within the Nuprl system and the program extracted from the proof is a decision procedure for the logic.

1.1 Related Work

The idea of extending theorem prover capabilities by the addition of formally verified decision procedures is not new, proposals we made as as early as 1977 [9] and there have been many more since. Actual formal verifications of decision procedures are less common. One example that has been repeated a number of times is Boyer and Moore's propositional tautology checker in the form of an `IF-THEN-ELSE` normalization procedure [5, 16, 11, 22, 21]. The paper by Paulin-Mohring and Werner [21] is the closest in spirit to the work presented here in that they extract the program for the tautology checker from a constructive proof in the Coq system. They address issues related to the efficiency of the extracted program. The proof presented here is for a sequent formulation of propositional logic which includes the classical propositional operators for negation, conjunction, disjunction, and implication.

1.2 Overview of the Approach

The development follows the natural proof informally presented by Constable and Howe in their paper [6]. In the formalization presented here the syntax of propositional formulas is formalized as a recursive type in the Nuprl system. Sequents are formalized as pairs of lists of formulas. The presentation here differs from [6] in that the semantics for the logic are provided via a Kleene valuation defined over formulas which is then lifted to sequents.

To avoid confusion the reader should bear in mind there are three distinct logics in the presentation:

i.) a sequent formulation of classical propositional logic, decidability is being proved for this logic;

ii.) a formalization of Kleene's strong three valued logic which provides the semantic basis for the decidability result; and

iii.) the logic of Nuprl which serves as the formal metalanguage for the statement and proof of the result.

The program for deciding tautologies is extracted from the formal proof. The extracted algorithm is the decision procedure obtained by searching for a sequent calculus proof via recursive application of the propositional sequent rules, as long as any still apply, and then checking to see if all resulting sequents, which contain only variables, are axioms. The core of the algorithm is a recursive procedure which is extracted from a normalization theorem.

The decidability theorem itself is proved from the normalization lemma which is proved via a well-founded induction on sequents. The measure for the induction is the operator complexity of the sequent. Using the Nuprl extraction mechanism a recursive procedure for normalizing sequents is extracted from the proof of the normalization lemma. This program is applied to a sequent by the decision procedure extracted from the proof of decidability. The decision procedure applied to a tautologous sequent halts and indicates it is a tautology. If the decision procedure is applied to a sequent which is not a tautology, it returns a falsifying assignment (a counter-example).

Constable and Howe's informal proof uses a finite function representation for assignments and does not consider the problem of partial assignments, *i.e.* an assignment that does not provide truth values to all variables occurring in the formula. This situation occurs frequently, whenever a proof rule having two hypotheses is invoked. As will be seen, the variables not included in the domain of such assignments are "don't care" conditions; the value to assign such undefined variables can be arbitrarily chosen. In the formalization presented in this paper, the semantics for the formula are defined over Kleene's strong three valued logic. This proves to be the natural partial valuation semantics, the third value in Kleene's logic informally being interpreted as "unknown" or "don't care". To make the semantics for the three valued case coincide with the classical two valued logic, a unique but natural definition for validity is given. Informally, a formula is valid exactly when every assignment that contains enough information (assigns values to enough variables) to determine truth or falsity asserts the truth of the formula. This notion of formula validity under a Kleene valuation is lifted to sequents in the natural way.

2 A whirlwind introduction to Nuprl

The Nuprl type theory is a sequent presentation of a constructive type theory via type assignment rules. It supports an untyped lambda-calculus as its programming language. Following Barendregt [3] we can distinguish the Nuprl type theory from the perhaps more familiar Church style typed lambda-calculi by calling it a lambda calculus in the style of Curry or a lambda calculi with type assignment. In Nuprl the underlying programming language is untyped and the objective of a proof is either: to prove a type is inhabited, *i.e.* to show some program (term) is a member of the type or to show a term inhabits a particular type. A complete presentation of the type theory can be found in the Nuprl book [7] (which will be referred to subsequently as "the book.").

The Nuprl system, as distinguished from the type theory, implements a rich environment to support reasoning about and computing with the Nuprl type theory. The system implementing the type theory has evolved since publication of the book but (with a few extensions) the type theory presented there is faithfully implemented by the Nuprl system. Complete documentation is included in the Nuprl V4.2 distribution. [1]

The following sections give a brief introduction to the Nuprl computation system, the type theory and some aspects of the system. An example defining a bounded list quantification operator via general recursion is also included.

2.1 The computation system

Nuprl *terms* include the constructs of its untyped functional programming language with additional constructs for denoting types and propositions. Terms will be printed here in typewriter font and will sometimes be enclosed in term quotes: $\ulcorner$`this is a term`$\urcorner$. Variables denoting terms will be printed in italics (t). Terms are either *canonical* or *noncanonical.* A term is canonical if its outermost forms are canonical (they may contain non-canonical subterms.) The Nuprl computation system provides reduction rules for evaluation of noncanonical terms. The Nuprl evaluator is the computer implementation of these rules. A complete description of canonical and non-canonical terms and the semantics for the computation system is provided in the book.

For terms `t` and `t'` we will write `t` $\triangleright$ `t'` to indicate that `t` (the *redex*) evaluates to `t'` (the *contractum*) under the reduction rules. In later sections we will apply an extended version of the basic computation system via the

[1]The Nuprl system is available from Cornell via the World Wide Web at `http://www.cs.cornell.edu/Info/Projects/Nuprl/nuprl.html` or by anonymous ftp from `ftp.cs.cornell.edu`.

rewrite facility. For terms `t` and `t'` we will write $\mathtt{t} \rhd_R \mathtt{t'}$ to indicate that `t` reduces to `t'` in the extended system.

As usual, the notation `t[t'/x]` denotes the term resulting from the substitution of `t'` for free occurrences of `x` in `t`. Similarly, $\mathtt{t}[\mathtt{t}_1,\cdots,\mathtt{t}_n/\mathtt{x}_1,\cdots,\mathtt{x}_n]$ denotes the simultaneous substitution of each $\mathtt{t}_i$ for each $\mathtt{x}_i$ in `t`.

The reduction rules used in the decidability proofs are presented here for quick reference.

$$
\begin{array}{lll}
(\boldsymbol{\lambda} x.b)(t) & \rhd & b[t/x] \\
\mathtt{list\_ind}([\,];b;x,y,z.u) & \rhd & b \\
\mathtt{list\_ind}(h::t;b;x,y,z.u) & \rhd & u[h,t,\mathtt{list\_ind}(t;b;x,y,z.u)/x,y,z] \\
\mathtt{decide}(\mathtt{inl}(t);x.l;y.r) & \rhd & l[t/x] \\
\mathtt{decide}(\mathtt{inr}(t);x.l;y.r) & \rhd & r[t/y] \\
\mathtt{spread}(<t_1,t_2>;x,y.t) & \rhd & t[t_1,t_2/x,y] \\
\mathtt{atom\_eq}(a;b;t_1;t_2) & \rhd & t_1 \quad \mathtt{if}\ a{=}b \\
\mathtt{atom\_eq}(a;b;t_1;t_2) & \rhd & t_2 \quad \mathtt{if}\ a{\neq}b \\
\mathtt{rec\_ind}(a;h,z.d) & \rhd & d[a,\boldsymbol{\lambda} z.\mathtt{rec\_ind}(z;h,z.d)/h,z]
\end{array}
$$

The first rule is the ordinary beta-reduction rule. Since it is included in the computation system, and since Nuprl terms are not tagged with type-information (*a la* Church), the evaluator is an interpreter for the untyped lambda calculus (extended with the computation rules just defined.) The Nuprl term evaluator implements a left-most outermost (lazy) evaluation strategy, although for terms which can be assigned a type in the system, evaluation order is immaterial since the typeable terms are strongly normalizing [7, 2].

2.2 Propositions-as-types and Proofs-as-programs

The principle of constructive proof can be summed up by saying a proof provides *direct evidence* for the proposition being proved [10]. Basing the semantics for a logic on this principle is sometimes known as the Heyting interpretation [12].

In the Heyting interpretation, truth corresponds to provability and falsity corresponds to the absence of any proof. A proof of the conjunction $P \wedge Q$ is a pair of proofs $\langle a, b\rangle$ where a is a proof of P and b is a proof of Q. A proof of the disjunction $P \vee Q$ is a proof of P or a proof of Q and there is some means of distinguishing which of the two was proved. A proof of the implication $P \Rightarrow Q$ is an operator (a function) that transforms a proof of P

into a proof of Q. A proof of $\neg P$ (which is just an encoding for $P \Rightarrow$ *false*) is a function that, from a proof of P, proves absurdity, *i.e.* it constructs a proof of *false*. A proof of $\exists x : T.P[x]$ is a pair $\langle a, b\rangle$ where a is an element of T and b is a proof of $P[a/x]$. A proof of $\forall x : T.P[x]$ is a function mapping a proof a of T to proofs of $P[a/x]$.

Thus, a proposition is true when it has a proof, when there is direct evidence for its truth. Thinking of the proofs of a proposition as somehow being "members" of the proposition is a long step toward understanding the propositions-as-types notion. Thus a proposition is *true* when it has members (proofs) and false otherwise. On the type theory side, we say a type is *inhabited* when it has members and is empty otherwise.

The proofs-as-programs notion is explained by seeing that lambda-terms can encode proofs under the Heyting interpretation. For the case of implications it is trivial; by the Heyting interpretation, a proof of the proposition $P \Rightarrow Q$ is a function mapping a proof of P to a proof of Q. Thus, a proof of the implication must be a lambda-term of the form $\lambda x.M$ such that, for all proofs p of P, the application $(\lambda x.M)(p)$ (which, under one step of beta-reduction, is $M[p/x]$) is a proof of Q. Completing the translation, if for all p in P, the term $M[x/p]$ is in Q the term $\lambda x.M$ is a proof of $P \Rightarrow Q$.

The Heyting interpretation motivates the next section which presents the the Nuprl type system. The connection will be made explicit in a later section when the constructive logic encoded by Nuprl types is presented.

2.3 The type theory

A Nuprl type is a term `T` of the computation system with an associated transitive and symmetric relation denoted by the term `x=y∈T`. This relation is known as *type membership equality* and respects evaluation in terms `x` and `y` (it is an equivalence relation when restricted to members of `T`). A point of confusion to the novice is that, unlike set theory, type membership equality is well-formed (is a proposition) only when `T` is a type and `x` and `y` are both elements of type `T`; if `T` is not a type or either of `x` or `y` (or both) are not elements of `T` then the term `x=y∈T` denotes nothing, it is nonsense.

In addition to the type membership equality provided with each type, there is an equality on types. Equality of types is intensional *i.e.* type equality in Nuprl is a structural equality modulo the direct computation rules. This means that, unlike sets which enjoy extensional equality, two types may contain the same elements and share an equality relation but not be equal types. For example, although `T` and `{x:T | True}` have the same members and equality relations, they are not equal types in Nuprl.

Interpreting the type membership equality relation and type membership as types is made sensible via the propositions-as-types interpretation [7, pg.29–31].

`x=y∈T` is an *equality* term. It denotes a type when `T∈𝕌` and `x∈T` and `y∈T`, otherwise it denotes nothing, it is nonsense . If `x` and `y` are not equal elements in `T` then the type is empty. If `x` and `y` denote equal elements in `T` the type in inhabited by the single element denoted by the term `Axiom`.

`x∈T` is a *membership* term. It is an encoding for the equality term `x=x∈T`. It denotes nothing if `T` is not a type or if `x` is not in `T` and is inhabited by the single term `Axiom` otherwise.

Like the related type theory of Marin Löf [17] or the type theory of Whitehead and Russell's *Principia Mathematica*, the Nuprl's type theory is a predicative type theory supporting an unbounded cumulative hierarchy of type universes. Every universe is itself a type and every type is an element of some universe.

$\mathbb{U}\{i\}$ denotes the type *universe* where i is a universe level expression.[2] The members of the universe $\mathbb{U}\{\mathtt{i}\}$ are types and other universes $\mathbb{U}\{\mathtt{j}\}$ for `j<i`. The property of being a type is formally written `T∈𝕌`.

$\mathbb{P}\{\mathtt{i}\}$ is a synonym for $\mathbb{U}\{\mathtt{i}\}$ and is sometimes used to emphasize the propositional side of the propositions-as-types interpretation.

The other Nuprl types and their members include the following:

`Void` is the *empty* type of which there are no members. Given a declaration `x:Void` (absurdly declaring the existence of an element of the empty type) the constant `any` is such that for all types `T`, `any(x)∈T`.

$\mathbb{Z}$ is the type *integer* whose members are denoted by the numerals $\cdots, -1, 0, 1, 2, \cdots$.

[2]Level expressions are not documented in the book. The Nuprl V4 level expressions provide a means of polymorphically referring to universe levels without specifying explicitly which level is intended [14, pg.23].

`Atom` is the type whose elements are *strings* of the form `''⋯''` where ⋯ is any character string. Atoms are equal when they are the same character string.

`T list` is the type of *lists* of elements of type `T`. The elements of `T list` include the empty list, denoted `[]` and conses of the form `a::t` where `a∈T` and `t∈T list`. Lists are equal either when they are both the empty list or when they have equal heads and their tails are equal.

`y:A⟶B[y]` is the *dependent function* type containing functions with domain of type `A` and range type `B[y]` where `y` is a variable possibly occurring free in `B`. When `a∈A` and `M[a/x]∈B[a/y]`, a lambda abstraction of the form `λx.M` is an element of the type `y:A⟶B[y]`. These are the functions whose range may depend on the element of the domain applied to. Function equality is extensional.

`A⟶B` is the *function* type which is an encoding of terms of the form `y:A⟶B` when `y` does not occur free in `B`.

`x:A×B[x]` is the *dependent product* type consisting of pairs `<a,b>` where `a∈A` and `b∈B[a/x]`. Two pairs `<a,b>` and `<a',b'>` are equal in `x:A×B[x]` when `a=a'∈A` and `b=b'∈B[a/x]`.

`A×B` is the *product* type and is an encoding of terms `x:A×B` where `x` does not occur free in `B`.

`A | B` denotes the *disjoint union* of types `A` and `B`, elements of this type are tagged elements of the form `inl(a)` for `a∈A` and `inr(b)` for `b∈B`. Two elements of the disjoint union are equal when their tagged elements are equal in the underlying type `A` (if the tag is `inl`) or `B` (if the tag is `inr`).

`rec(x.T)` is the Nuprl *inductive type* constructor where `x` is a variable bound in term `T`, free occurrences of `x` in `T` denote inductively smaller elements of the type, thus its members are the members of `T[rec(x.T)/x]`. There are some technical constraints on the form of `T` but we do not include them here. Whenever `rec(x.T)` is a type, members `a` and `b` are equal if `a=b∈T[rec(x.T)/x]`.

`{y∈T|P[y]}` denotes a *set type* when `T` is a type and `P[y]` is a proposition possibly containing free occurrences of the variable `y`. Elements `x` of this type are elements of T such that P[x/y] is true. Equality for set types is just the equality of T restricted to {y∈T|P[y]}.

x,y:A//E[x,y] denotes a *quotient* which is a type whenever A is a type, and E[x,y] is an equivalence on A. Its members are elements of A and it identifies elements a and b whenever the equivalence E[a,b/x,y] is inhabited.

2.4 Judgements

Nuprl judgements are the assertions one proves in the system. Nuprl judgements take one of two forms[3]:

$$\mathtt{x}_1\mathtt{:T}_1\mathtt{,}\cdots\mathtt{,x}_n\mathtt{:T}_n \;\mathtt{>>}\; \mathtt{S}\ \mathtt{[ext\ s]}$$

or

$$\mathtt{x}_1\mathtt{:T}_1\mathtt{,}\cdots\mathtt{,x}_n\mathtt{:T}_n \;\mathtt{>>}\; \mathtt{s} \in \mathtt{S}\ \mathtt{[ext\ Axiom]}$$

where $\mathtt{x}_1,\cdots,\mathtt{x}_n$ are distinct variables and $\mathtt{T}_1,\cdots,\mathtt{T}_n$, S, and s are terms (n may be 0), every free variable of $\mathtt{T}_i$ is one of $\mathtt{x}_1,\cdots,\mathtt{x}_{i-1}$ and every free variable of S or of s is one of $\mathtt{x}_1,\cdots,\mathtt{x}_n$. The list $\mathtt{x}_1\mathtt{:T}_1,\cdots,\mathtt{x}_n\mathtt{:T}_n$ is called the *hypothesis list*, each $\mathtt{x}_i\mathtt{:T}_i$ a declaration (of $\mathtt{x}_i$), each $\mathtt{T}_i$ is a *hypothesis*, S (s∈S) is the *consequent* or *conclusion*, the term following the keyword ext is the *extract*, and the entire form is a Nuprl *sequent*. Because the inhabitant of the type in the first form is in the extract (which is not displayed by the system) it is sometimes called an *implicit* judgement. By the same reasoning, the second form is sometimes referred to as an *explicit* judgement. An explicit judgement is also called a *well-formedness goal*. The computational content of well-formedness goals is trivial.

The conditions under which a Nuprl sequent is deemed true are rather technical because of the so-called functionality constraints insuring equal elements of hypotheses can be freely substituted into the consequent and extract terms; the reader is referred to the Nuprl book [7, pg.141] for a full account. Informally, the implicit judgement asserts that, assuming the hypotheses are well-formed types, and the conclusion and extract terms are functional in those types, then the term S is an inhabited type and the extract s is an inhabitant. The fact that the extract term s inhabits S is an artifact of the proof that S is inhabited. If S is inhabited there may be more than one inhabitant and different proofs may yield different inhabitants. The

[3]The first form subsumes the second; there is really only one form but it is useful to make the distinction as if there were two forms so we do.

explicit judgement form similarly asserts that `S` is inhabited but in addition proves `s` to be the(an) inhabitant. Since `s∈S` is simply shorthand for `s=s∈S` by the propositions-as-types interpretation for type equality, the extract of the explicit form is just `Axiom`.

A Nuprl goal is a judgement having no hypotheses.

A Nuprl proof is a decorated tree structure in which the children of each node are instances of sequents justified by the rules of the type theory. A proof of a sequent shows that the goal, viewed as a type, is both well-formed and inhabited. Given terms inhabiting the hypotheses of a rule, it specifies how to construct a term inhabiting the type in the conclusion of the rule; thus, proofs contain instructions for the construction of witnesses. *Extraction* is the process of constructing a witness term as specified by a proof. Although extracts are not displayed by the system, the extract term can be retrieved by applying the function `extract_of_thm_object` to a token containing the name of the theorem to be extracted. The extract of a completed proof of a sequent is a closed term; the extract of an incomplete proof is a term possibly containing free variables.

2.5 The Nuprl system

The Nuprl system supports construction of top-down proofs by refinement. The prover is implemented as a tactic based prover in the style of LCF [13] and built on a base of ML. In Nuprl and related constructive systems [18, 20, 8], the so-called proposition-as-types interpretation allows for presentations to be cloaked in either logical or more purely type-theoretic terms. Paul Jackson's huge effort to rationally reconstruct the Nuprl V3 tactics and display forms serves as the basis of the Nuprl V4 system; it gives the system a distinctly logical appearance (in contrast to the more type-theoretic appearance of other systems.)

Additionally, the system supports a library mechanism which provides for grouping of Nuprl objects. The status and class of an object is indicated in the library by a two character sequence preceding the name of the entry in the library, the first character indicates the status and the second indicates the type of object. The seven kinds of objects (and their single character labels) are the following: an *abstraction* object (`A`); a *comment* object (`C`), a *display form* object (`D`), an *ml* object (`M`), a *precedence* object (`P`), a *rule* object (`R`), and *theorem* objects which are labeled by the lower case character (`t`) if the theorem is unexpanded and are labeled by the upper case character (`T`) if expanded. Every object has associated with it a status, also labeled by a single character, which is either *raw* (?), *bad* (-), *incomplete* (#), or *complete* (*). A *raw* status means an object has been changed but not yet checked. A *bad* status means an object has been checked and found to contain errors. An *incomplete* status is meaningful only for theorem objects and signifies that its proof contains no errors but has not been finished. A *complete* status indicates that the object is correct and complete.

Typically, when a new operator is being defined a display form, abstraction, and well-formedness theorem are provided. If the operator has associated computational behavior an ML object supporting the selective rewriting is usually also added.

2.6 Logic via propositions-as-types

A constructive logic is encoded within the Nuprl type theory which formalizes the Heyting interpretation presented above. The following abstractions defined in the Nuprl V4 `core_1` system library encode the logic.

```
*A true    True ==  0 ∈ ℤ
*A false   False ==  Void
*A and     P ∧ Q ==  P × Q
*A or      P ∨ Q ==  P | Q
*A implies P ⇒ Q ==  P → Q
*A not     ¬A ==  A ⇒ False
*A exists  ∃x:A. B[x] ==  x:A × B[x]
*A all     ∀x:A. B[x] ==  x:A → B[x]
```

Thus, `True`, which could be modeled by any inhabited type, is modeled here as the type asserting $0 \in \mathbb{Z}$ (which happens to be true and has the single inhabitant `Axiom`.) `False` is encoded as the empty type, conjunction is just a simple product, disjunction is disjoint union, implication is the function type, negation of a proposition `P` is defined to be the function that, applied to an element of `P` returns an element of the empty type, existential quantification is encoded as the dependent product type, and universal quantification is the dependent function type. The correspondence between the propositions encoded as above and the type theory is elaborated on in [7, Section 3.6,]. The Nuprl tactics have been built to deal with either the propositions or types formulation uniformly.

2.7 Set Types, Decidability, Stability, and Squash Stability

The principle of constructive evidence can sometimes result in overly complex types. The set type provides a mechanism for defining new types with possibly complex properties but which have the computational content associated with the defining property discarded. Thus, the declaration `a:{x:T | P[x]}`, asserts an element `a` such that `a∈T` and `P[a/x]` is inhabited, but without carrying along the proofs that it does. Since the computational content has been discarded, the fact that the `a` has the property `P[a/x]` is not freely obtainable in the proof; it is a so-called *hidden hypothesis*. Hidden hypotheses cannot be used (without first unhiding them) in proof steps that contribute to the computational content of the proof; however, not all proof steps do contributes. Well-formedness goals generated in the course of a proof, as well as equality reasoning, do not contribute.

This restriction on the use of hidden hypotheses, would appear to the limit the usefulness of the set type; however, if `P` is *stable*, *squash stable* or *decidable* then membership of an element can be effectively determined and the missing computational content is not needed; it resides in the justification of the stability or decidability of `P`.

Being constructive, Nuprl cannot in general prove the so-called law of excluded middle; that is, `∀P:ℙ.P∨¬P` is not a theorem of Nuprl. Even though excluded middle is not provable for arbitrary propositions P, for many P it is uniformly decidable (*i.e.* there is an algorithm to decide) which of P or $\neg P$ holds. That is precisely the definition of the abstraction `Dec{P}`.

```
*A decidable     Dec{P} == P ∨¬P
*T decidable_wf  ∀P:ℙ{i}. (Dec{P} ∈ ℙ{i})
```

Note that the well-formedness theorem asserts the decidability proposition is a type but does not prove it is inhabited for arbitrary propositions `P`.

A related notion is that of *stability* which is weaker than decidability but is still not constructively valid.

```
*A stable        Stable{P} == ¬¬P→P
*T stable_wf     ∀P:ℙ{i}. (Stable{P}∈ℙ{i})
```

Stability is weaker in the sense that it is (constructively) implied by decidability but does not imply decidability. It is an interesting exercise for the reader to try to prove ∀P:ℙ{i}.Stable{P} →Dec{P} to see why the proof fails.

Squash stability is weaker even that stability and is related to stability in that they are equivalent under a presumption of decidability. That is, for all P, if P is decidable, then P is squash stable if and only if it is also stable.

```
*A sq_stable     SqStable{P} == ↓{P}  → P
*T sq_stable_wf  ∀P:ℙ{i}. (SqStable{P} ∈ ℙ{i})
```

The tactics for manipulating set types search for theorems to justify these properties via lemmas named by the convention

> `stable_`⟨*abs_name*⟩⟨*_optional_tag*⟩,
> `sq_stable_`⟨*abs_name*⟩⟨*_optional_tag*⟩, or
> `decidable_`⟨*abs_name*⟩⟨*_optional_tag*⟩.

2.8 An example recursive definition

In this section an operator for list quantification is developed. It exhibits a non-trivial use of the display-form mechanism, gives a definition of an abstraction employing a recursive function defined via general recursion, defines an ML object showing how the computational behavior is reflected into the rewrite system, and presents a well-formedness theorem and its extract term.

The following four entries appear in the `list_3` library supporting the propositional decidability theorems. The status of all the objects is complete.

```
*D list_all_df
   ∀<x:element>∈<L:List>.<P:T→ℙi:L>== list_all{}(<L>; <x>.<P>)

*A list_all
   ∀x∈L.P[x] ==
      (letrec list_all L =
                list_ind L of [] => True | h::t => P[h] ∧ list_all t
      ) L

*M list_all_unroll
   let list_all_conv T =
       FwdMacroC  `list_all_unrollC`
         (AllC [UnfoldC `list_all`;
```

```
                letrec_unrollC;
                ReduceC;
                TryC (FoldC `list_all`)])
      T
   ;;
   let list_all_unrollC =
    SomeC [list_all_conv ⌜∀x∈[].P[x]⌝;
           list_all_conv ⌜∀x∈(h::t).P[x]⌝]
   ;;
   add_AbReduce_conv `list_all` list_all_unrollC;;

*T list_all_wf
   ∀T:𝕌. ∀P:T → ℙ. ∀L:T List.  ∀x∈L.P[x] ∈ 𝕌
   Extraction:
      λT,P,L.list-case(L) of [] => Axiom | u::v => %.Ax
```

The first entry is a display form object named `list_all_df`. This object provides a template for display of instances of the `list_all` operator. The naming convention in Nuprl is to give display-form objects names of the form *op_id_name*`_df` where *op_id_name* is the operator identifier of the abstraction being displayed. The display-form itself has two components separated by a double equality; "*[lhs]*`==`*[rhs]*". The *lhs* is the display template with slots. In the display-form for `list_all` there are three: one slot for the variable `x` which is bound by the quantifier; one slot for the list `L` which is the domain of the quantification; and one slot for the predicate `P[x]` possibly containing free occurrences of the variable `x`. In a complete display-form, *rhs* is an instance of the operator being displayed. In this case the operator is `list_all{}(<L>; <x>.<P>)` having the operator identifier `list_all` and having no parameters (which would be enclosed within the set brackets) and having three meta-variables `L`, `x`, and `P` with the notation `<x>.<P>` indicating that `x` is bound in `P`. The Nuprl display form mechanism supports additional features not used here.

The second library entry is an abstraction with the name `list_all`. The naming convention for abstractions is to give them the same name as the operator identifier of the operator begin defined. In the remainder of this paper, unless there is some interesting characteristic of a display-form itself, display definitions will not be shown, abstractions will be displayed as above; *e.g.* with the instantiated display-form to the left of the "`==`" and with the definition to the right. In this case the definition consists of a recursive function, defined via a `letrec` form, applied to the argument `L`.

The letrec form used to define the list_all operator is one of three abstractions that hide the underlying general recursive definitions implemented by the fixed-point combinator Y. The Y combinator is defined in the Nuprl system library core_2 as follows:

```
*A ycomb          Y == λf.(λx.f (x x))(λx.f (x x))
```

The methodology developed by Howe and Jackson for effective use of definitions incorporating Y depend on the rewriting system. The rewrite conversion YUnrollC encapsulates the fixed-point property of the Y combinator as follows:

Y F $\triangleright_R$ F (Y F)

where F is any term. Although Y is not typeable in Nuprl, its use in definitions is possible because its behavior as a fixed-point combinator is justified via the direct computation rules which implicitly preserve typing; because of this, no well-formedness goals are generated for Y when the conversion is applied.

The letrec form is implemented through three display forms and abstractions that hide the underlying use of the fixed point combinator.

```
*A letrec              (letrec f b[f])  == Y (λf.b[f])
*A letrec_body         = b  == b
*A letrec_arg          x b[x]  == λx.b[x]
```

Like YUnrollC, a conversion supports the unfolding of letrec, letrec_arg and letrec_body terms; the conversion letrec_unrollC captures the following computational behavior of applications of letrec terms.

(letrec f t = b[f;t])(T) $\triangleright_R$ b[(letrec f t = b[f;t]),T/f,t]

i.e. the recursive call is substituted for f in the term b[f;t] and the argument T is substituted for t.

Based on these definitions, the behavior of the abstraction for the list_all operator should be transparent. The computational behavior can be explained by considering the rewrite conversions defined in the ML object associated with its definition.

The ML object named list_all_unroll contains the code used to selectively unroll occurrences of the list_all operator. The details of the ML code are unimportant, but it is worth pointing out how selective rewrite support is provided for recursively defined functions. Upon evaluation of the

ML object list_all_unroll by the system, the two objects list_all_conv and list_all_unrollC result in ML objects of type conversion. When applied by the rewrite system, the conversions exhibit the following behavior.

$$\forall \mathtt{x}{\in}\mathtt{[].P[x]} \;\; \triangleright_R \;\; \mathtt{True}$$
$$\forall \mathtt{x}{\in}\mathtt{h::t.P[x]} \;\; \triangleright_R \;\; \mathtt{P[h]} \wedge \forall \mathtt{x}{\in}\mathtt{h::t.P[x]}$$

The last line of the ML object adds the list_all_unrollC conversion to the list of conversions applied by the tactic AbReduce when an operator with operator id 'list_all' is encountered having either the empty list or a cons as its primary argument. For a complete account of the rewrite system and the Nuprl ML system the documentation provided with the system should be consulted.

The third library entry above is a well-formedness theorem object for the list_all abstraction. The theorem is named list_all_wf following the convention used by the well-formedness tactics which will search for it by name and automatically apply it when well-formedness goals are induced during the proving process. The theorem says that for appropriately typed arguments, the list_all operator denotes a type. More precisely, for every type T, and for every T List, and for every proposition (function from T onto $\mathbb{P}$) the list_all operator is a member of all levels of the type universe hierarchy. Typically we will omit the presentation of well-formedness goals that simply state an abstraction is an element of some type universe.

The proof of list_all_wf provides the extract

```
λT,P,L.list-ind(L) of [] => Axiom | u::v => %.Axiom
```

This term is one function inhabiting the proposition

$$\forall \mathtt{T}:\mathbb{U}.\ \forall \mathtt{P}:\mathtt{T} \rightarrow \mathbb{P}.\ \forall \mathtt{L}:\mathtt{T\ List}.\ \ \forall \mathtt{x}{\in}\mathtt{L.P[x]} \in \mathbb{U}$$

The theorem was proved by induction on L and so extraction includes the the list-ind form which is the computational content of list induction. In the case the list is empty the extract evaluates to Axiom, this is because the element inhabiting the proposition True∈$\mathbb{U}$ is Axiom. The inductive hypothesis of the list induction establishes that the recursive call is well-formed, thus in the case of a cons u::v the extract evaluates to %.Axiom [4]

[4] The term ⌜%.Axiom⌝ denotes the binding of the hidden variable "%" in the constant Axiom. The binding is an artifact of the listElimination rule and the term can be considered identical to the term ⌜Axiom⌝.

3 Formulas, Semantics, and Sequents

In this section the Nuprl definitions supporting the statement and proof of the decidability theorem are presented. The syntax of propositional formulas are formalized using the recursive type mechanism. The semantic notion of a Kleene (partial) valuation on a formula is defined. This requires the development of a theory of three valued types and the definition of the logical operators of Kleene's strong three valued logic. Kleene valuations are defined over three valued assignments using the operators. The semantics for formulas is then defined in terms of Kleene valuations by giving definitions for formula satisfiability, formula falsifiability and a unique definition of formula validity. Subsequently, a sequent type is defined as the Cartesian product of lists of formulas. The semantics of sequents is given by lifting the formula semantics in the natural way to define sequent satisfiability, sequent falsifiability, and sequent validity.

The decidability theorem is stated and is proved in the next section using these definitions. The proof is by induction on the complexity of sequents where the measure is the number of propositional operators occurring in the sequent. Thus, the definition of sequent rank (defined in terms of formula rank) is also given here.

3.1 Formula

In the Nuprl formalization of the logic formulas are characterized by a recursive type.

```
*A Formula
Formula == rec(F.Var | F | (F × F) | (F × F) | (F × F))
```

The `Formula` type abstraction is defined to be the recursive type whose members are a disjoint union of five elements. The first element of the disjoint union is the uninterpreted type `Var` or propositional variables. Since the variable `F`, bound by the rec-type operator, doesn't occur in the first component, terms of the form `inl(x)`, where `x` is an element of the type `Var`, form the basis elements of the recursive type. The second component of the disjoint union is an instance of the bound variable `F` denoting a recursively smaller element of the formula type. The third, fourth, and fifth elements of the disjoint union are the Cartesian products (pairing) of two recursively smaller formula. When the semantics of the propositional formulas are defined below it will become clear that the second disjunct of the formula type denotes the negation of the formula F and the pairs of formula in the third, fourth, and fifth disjuncts denote the operators for conjunction, disjunction, and material implication.

3.1.1 Constructors

To facilitate manipulation of the type `Formula` a collection of constructors and a destructor are defined. The constructors and their well-formedness theorems are as follows.

```
*A fvar        ⌈x⌉ == inl x
*T fvar_wf     ∀x:Var. (⌈x⌉ ∈ Formula)

*A fnot        (⌈∼⌉p) == inr inl p
*T fnot_wf     ∀p:Formula. (⌈∼⌉p ∈ Formula)

*A fand        (p⌈∧⌉q) == inr inr inl <p,q>
*T fand_wf     ∀p,q:Formula. (p⌈∧⌉q) ∈ Formula

*A for         (p⌈∨⌉q) == inr inr inr inl <p,q>
*T for_wf      ∀p,q:Formula. (p⌈∨⌉q) ∈ Formula

*A fimp        (p⌈⇒⌉q) == inr inr inr inr <p,q>
*T fimp_wf     ∀p,q:Formula. (p⌈⇒⌉q) ∈ Formula
```

Thus, `(((⌈x⌉⌈⇒⌉⌈y⌉)⌈⇒⌉⌈x⌉)⌈⇒⌉⌈x⌉)` is a formula.

A formula of the form `⌈x⌉`, where `x` denotes an element of type `Var`, will be called an *atomic formula* (or simply *atomic*) and all others are called *non-atomic formula*.

3.1.2 Case analysis

The `formula_case` operator defined below is the destructor for the `Formula` type. It is defined using nested case analysis on the disjoint union type. A nested series of `decide` operators gives the case analysis for the type `Formula`. The definition is as follows:

```
*A formula_case
case F:
      ⌈x⌉ → varC[x];
```

```
   ⌈∼⌉p1 → notC[p1];
   p2⌈∧⌉p3 → andC[p2; p3];
   p4⌈∨⌉p5 → orC[p4; p5];
   p6⌈⇒⌉p7→impC[p6; p7];
==
decide F of
  inl(x) => varC[x]
  | inr(F) => decide F of
  inl(p1) => notC[p1]
  | inr(F) => decide F of
  inl(x) => let p2,p3 = x in andC[p2; p3]
  | inr(F) => decide F of
  inl(x) => let p4,p5 = x in orC[p4; p5]
  | inr(x) => let p6,p7 = x in impC[p6; p7]
```

In the abstraction, the display form slots contain occurrences of the second-order variables `varC[x]`, `notC[p1]`, `andC[p2;p3]`, `orC[p4;p5]` and `impC[p6,p7]`. In an instantiation of the `formula_case` operator, terms, possibly containing free occurrences of the bracketed variables, are substituted for the variables. The bracketed variables are are bound by the names to the left of the $\rightarrow$.

As an example usage of the operator we define a function which collects the principal subformula of a formula into a list.

```
*A principal_subformula
  principal_subformula(F) == case F:
                               ⌈y⌉ → [];
                               ⌈∼⌉p → (p::[]);
                               p⌈∧⌉q → (p::q::[]);
                               p⌈∨⌉q → (p::q::[]);
                               p⌈⇒⌉q → (p::q::[]);
```

In this instantiation of the `formula_case` operator, the display-form variable `x` is bound to `y` and the term denoting the empty list (`[]`) is bound to the second-order variable `varC[y]`. In the second case, the variable `p1` is bound to `p` and the term `(p::[])` is bound to the second-order variable `notC[p]`. The other cases are similarly explained. Thus, for a formula `F`,

$$\texttt{principal\_subformula(}\ulcorner\!\sim\!\urcorner\texttt{F)} \quad \rhd_R \quad \texttt{F::[]}$$

e.g. the computation system extended to include computation over the `principal_subformula` abstraction evaluates the term `principal_subformula(⌈∼⌉F)` to the list containing the single formula `F`.

3.1.3 Formula rank

Having defined the type `Formula` and the supporting constructors and destructor we define a well-founded measure on formulas. The following measure function, a count of the number of operators in a formula is, arguably, the simplest and most natural ordering to consider.

```
*A formula_rank  ρ ==
    letrec measure f =
      case f:
        ⌈x⌉ → 0;
        ⌈∼⌉p → ((measure p) + 1);
        p⌈∧⌉q → (((measure p) + (measure q)) + 1);
        p⌈∨⌉q → (((measure p) + (measure q)) + 1);
        p⌈⇒⌉q → (((measure p) + (measure q)) + 1);
```

Under `formula_rank`, formulas that are propositional variables are assigned rank 0, while negations, conjunctions, disjunctions, and implications are all assigned the sum of the recursively computed ranks of their principal subformulas plus 1.

The well-formedness theorem for the `formula_rank` function certifies it is a function from formulas to natural numbers.

```
*T formula_rank_wf   ρ ∈ Formula → ℕ
```

3.2 Three valued Semantics of propositional logic

In standard treatments of decidability (and completeness) for classical propositional logic, for example as found in Mendelson [19], *truth assignments* are total functions mapping the (countably infinite) propositional variables onto the Booleans. More constructively, Smullyan [23] considers finite functions mapping the set of variables occurring in a formula (or set of formulas) onto the Booleans. In both presentations it is shown how assignments uniquely determine *Boolean valuations*. The decision procedure developed here not only determines whether a propositional sequent (to be defined below) is valid, but in the case the sequent is not valid, it returns a falsifying assignment. In the natural development, falsifying assignments generated by the decision procedure are finite functions, but furthermore, their domains may not even include all variables occurring in the sequent. Thus, the formal development must either account for an arbitrary extension of the domain of a partial falsifying assignment to include all variables occurring in the sequent or, the notion of partial assignments must be accounted for in the semantics. The second approach is developed here.

We define the semantics of propositional logic in terms of Kleene's strong three-valued logic [15]. A Kleene valuation reflects the classical interpretations of the standard propositional connectives under fully determined assignments (those assigning true or false to every variable in the formula); but also, when a partial assignment has "enough information" to determine the truth or falsity of a formula the Kleene valuation induced by it does too. For example, if either conjunct of the formula $\mathtt{p}\wedge_K\mathtt{q}$ is *false* under the Kleene valuation induced by a partial assignment a, then $\mathtt{p}\wedge_K\mathtt{q}$ is *false* under the valuation too. It does not matter what value the other conjunct has, or even if it is defined. Under all extensions of a, the valuation of $\mathtt{p}\wedge_K\mathtt{q}$ is false. Thus, once determined, an assignment remains fixed. Similar rules apply for the other operators which are formally defined below.

To proceed with the formalization in Nuprl we first define a three valued type.

3.2.1 A three valued type

There are a number of ways to formalize a three element type. Here we choose to use a three-way disjoint union.

```
*A Three        3 == Unit | Unit | Unit
```

It does not matter what types we choose as component types of the disjoint union since we are only distinguishing which component an element is in and nothing else. For simplicity we choose the type `Unit` having only one member. The single element of `Unit`, called "dot" is displayed as '·'. Thus, **3** is the type containing the injections of · into the first, second, or third components of the three part disjoint union. We provide names and display forms for each of the three elements below.

```
*A Three_0      0₃ == inl ·
*T Three_0_wf    0₃ ∈ 3
```

```
*A Three_1      1₃ == inr inl ·
*T Three_1_wf    1₃ ∈ 3

*A Three_2      2₃ == inr inr ·
*T Three_2_wf    2₃ ∈ 3
```

A case discriminator for the three-valued type is also defined.

```
*A Three_case
case x: 0₃ → case0; 1₃ → case1; 2₃ → case2; ==
    decide x of
      inl(zero) => case0
    | inr(one_or_two) =>
         decide one_or_two of
           inl(one) => case1
         | inr(two) => case2
```

Continuing with the development of the theory we define two tactics: `ThreeInd` and `ThreeNEQ`. Following the convention used in the Nuprl V4 tactics we name the tactic that does the case analysis on the type **3**, `ThreeInd`.[5]

```
H₀, (i) x:3, H₁ >> C[x] by ThreeInd i
   H₀, H₁[0₃/x] >> C[0₃/x]
   H₀, H₁[1₃/x] >> C[1₃/x]
   H₀, H₁[2₃/x] >> C[2₃/x]
```

If a variable `x` is declared to be of type **3** in hypothesis `i` of a Nuprl sequent, the application of the tactic `ThreeInd i` generates three subgoals, one for each of the three elements of the type. The subgoals are formed by substituting one of 0_3, 1_3, or 2_3 for occurrences of the variable `x` in the goal sequent. The raw extract generated by application of the ThreeInd tactic is of the following form.

```
decide x of
  inl(x1) => Ext0
| inr(y)  => decide y of
               inl(x) => Ext1
             | inr(y1) => Ext2
```

[5]To read the rule-like characterization of the tactic: the top line is the goal sequent and the indented lines below it the three subgoals generated by application of `ThreeInd` to hypothesis number `i` in the goal.

Here, Ext_0, Ext_1, and Ext_2 denote the extracts of the proofs of the three subgoals generated by the tactic. Observing that this term schema is an instance of the Three_case abstraction defined above, we will use the following tidied version when displaying the extract:

case x: $0_3 \longrightarrow Ext_0$; $1_3 \longrightarrow Ext_1$; $2_3 \longrightarrow Ext_2$;

The tactic ThreeNEQ solves goals (generating no subgoals) of the following form,

H_0, (i) $a = b \in \mathbf{3}$, H_1 >> C by ThreeNEQ i

where a and b are different constants of type $\mathbf{3}$ and hypothesis (i) falsely asserts their equality. The extract generated by the application of the tactic is a term which, applied to any argument, returns the constant Axiom. The raw extract and its reduction are shown here:

λ%.(λ%1.Axiom) Axiom $\rhd_R$ λ%.Axiom

The following theorem asserts $\mathbf{3}$ is a discrete type, that is, that the equality on $\mathbf{3}$ is decidable. This theorem is the first having a proof with interesting computational content. Since it is the first such proof we examine it and its extracted term in some detail.

*T decidable_equal_Three
$\forall$x,y:$\mathbf{3}$. Dec{x = y $\in$ $\mathbf{3}$}

The first step of the proof is the elimination of the outermost universal quantifiers by the tactic UnivCD THENA Auto. This yields the following Nuprl sequent.

1. x: $\mathbf{3}$
2. y: $\mathbf{3}$
$\vdash$ Dec{x = y $\in$ $\mathbf{3}$}

The extract resulting from this proof step has the form λx,y.Ext where Ext is the extract of the proof of resulting subgoal.

The next step in the proof is case analysis on x and then on y. Two applications of the ThreeInd tactic accomplish this. This results in nine subgoals.

```
1* ⊢ Dec{0₃ = 0₃ ∈ 3}
2* ⊢ Dec{0₃ = 1₃ ∈ 3}
3* ⊢ Dec{0₃ = 2₃ ∈ 3}
4* ⊢ Dec{1₃ = 0₃ ∈ 3}
5* ⊢ Dec{1₃ = 1₃ ∈ 3}
6* ⊢ Dec{1₃ = 2₃ ∈ 3}
7* ⊢ Dec{2₃ = 0₃ ∈ 3}
8* ⊢ Dec{2₃ = 1₃ ∈ 3}
9* ⊢ Dec{2₃ = 2₃ ∈ 3}
```

The extract resulting from this step will be nested occurrences of the Three_case operator with the first splitting on x and the second on y.

Each of the nine cases is easily proved. Recall that Dec{P} is the constructive disjunction $P \vee \neg P$. To prove cases 1, 5, and 9, the first disjunct is selected which in turn is discharged by the Auto tactic. Equality terms viewed as types, when true, have as their inhabitants the single element denoted by the constant Axiom. The proofs of each of these three cases contribute the extract inl(Axiom) to their respective case splits.

In the six other cases, the equality is false; to prove the theorem the second disjunct is chosen resulting in a subgoal having the negated form of the equality as its consequent. Eliminating the negation results in a false hypothesis which is then discharged by the ThreeNEQ tactic. The proofs of these six cases contribute the extract inr(λ%.Axiom) to the corresponding case.

The extract of the entire proof is a term deciding if two elements of the type **3** are in fact equal.

```
λx,y. case x: 0₃ → case y: 0₃ → inl(Axiom) ;
                           1₃ → inr(λ%.Axiom) ;
                           2₃ → inr(λ%.Axiom) ;;
              1₃ → case y: 0₃ → inr(λ%.Axiom) ;
                           1₃ → inl(Axiom) ;
                           2₃ → inr(λ%.Axiom) ;;
              2₃ → case y: 0₃ → inr(λ%.Axiom) ;
                           1₃ → inr(λ%.Axiom) ;
                           2₃ → inl(Axiom) ;;
```

Thus, to decide $x = y \in \mathbf{3}$ it is enough to apply the function to x and y and then to observe whether it returns a left or right injection; the content under the `inl` or `inr` is not used. This function is evidence for the proposition that the type is discrete.

3.2.2 Kleene's Strong Three Valued Logic

In this section the operators of Kleene's three valued logic are defined over the type **3**. Inspection of the definitions below reveals that on inputs restricted to 0_3 and 2_3 (which are to be interpreted as false and true respectively and will often be referred to as such below) the operators behave exactly as the familiar boolean operators of the same names. More technically, following Kleene [15], we may say, these operators are uniquely determined as the strongest possible regular extensions of the classical 2-valued operators.

```
*A K_not     ~K p ==   case p:  0₃→ 2₃;
                                1₃→ 1₃;
                                2₃→ 0₃;
```

Thus for negation the undefined value 1_3 is a fixpoint and, as is the case for the other Kleene operators, on the values 0_3 and 2_3 the Kleene operator reflects the behavior of its two-valued counterpart.

For a conjunction, in the case one of `p` or `q` is false then the conjunct $p \wedge_K q$ is false too. A conjunction is undefined either if one of the conjuncts is true and the other is undefined or if they're both undefined. It is true otherwise.

```
*A K_and     p ∧K q ==case p:  0₃→ 0₃;
                               1₃→ case q: 0₃→ 0₃;
                                           1₃→ 1₃;
                                           2₃→ 1₃;;
                               2₃→q;
```

For disjunctions, if one of `p` or `q` is true then the disjunction $p \vee_K q$ is too. It is undefined either if one disjunct is false and the other is undefined or if both disjuncts are undefined. It is false otherwise.

```
*A K_or      p ∨K q ==case p:  0₃→ q;
                               1₃→ case q:   0₃→ 1₃;
                                             1₃→ 1₃;
                                             2₃→ 2₃;;
                               2₃→ 2₃;
```

For an implication $p \Rightarrow_K q$, if either p is false or q is true then the implication is true as well. An implication is undefined if p is true and q is undefined or if both p and q are undefined. A Kleene implication is false otherwise.

```
*A K_imp     p ⇒K q == case p:  0₃→ 2₃;
                                1₃→ case q:  0₃→ 1₃;
                                             1₃→ 1₃;
                                             2₃→ 2₃;;
                                2₃→ q;
```

The well-formedness theorems for the Kleene operators exhibit their closure over the type **3**.

```
*T K_not_wf  ∀p:3.(~K p ∈ 3)
*T K_and_wf  ∀p,q:3.(p ∧K q ∈ 3)
*T K_or_wf   ∀p,q:3.(p ∨K q ∈ 3)
*T K_imp_wf  ∀p,q:3.(p ⇒K q ∈ 3)
```

3.2.3 Assignments and Kleene Valuation

The type of three valued assignments is defined as follows.

```
*A Assignment:          Assignment == Var → 3
```

The Kleene valuation of a formula F under the partial assignment a (displayed as (F under a)) is defined as follows.

```
*A valuation
(F under a)  == (letrec val f =
                   case f:
                     ⌈x⌉ → (a x);
                     ⌈~⌉p → ~K val p;
                     p⌈∧⌉q → val p ∧K val q;
                     p⌈∨⌉q → val p ∨K val q;
                     p⌈⇒⌉q → val p ⇒K val q;
                ) F
```

The `valuation` function recursively computes the Kleene valuation of the formula F under the assignment a. The abstraction is defined by the application of the recursive procedure `val` to the formula F. The body of the recursive procedure is defined via case analysis on the parameter f. In the base case, the formula `f` is a formula of the form ⌜`x`⌝, then the result is the value returned by the application of assignment `a` to the variable `x`. If `f` is a non-atomic formula the valuation is computed by applying the corresponding Kleene operator to the recursively computed values of the subformula of `f`.

As expected, the well-formedness theorem for the valuation operator says it's an element of the type **3**.

```
*T valuation_wf  ∀a:Assignment.∀F:Formula.((F under a)  ∈ 3)
```

3.2.4 Satisfaction and Falsification of Formulas

Using the Kleene valuation we define the semantic notion of a formula being satisfied (falsified) by an assignment `a`.

```
*A formula_sat            a |= F  == (F under a)  = 2₃ ∈ 3
*A formula_falsifiable    a |≠ F == (F under a)  = 0₃ ∈ 3
```

Thus, a formula `F` is satisfied by assignment `a` (written `a |= F`) when `(F under a)` evaluates to 2_3. Similarly, a formula `F` is falsified by assignment `a` (written `a |≠ F` when `(F under a)` evaluates to 0_3.

The satisfiability (or falsifiability) of a formula under an assignment is clearly a decidable property; to decide if a formula is satisfied (falsified) by `a`, evaluate `(F under a)` and check whether the result is equal to 2_3 (0_3). This property is captured by the following theorems.

```
*T decidable_formula_sat:
 ∀a:Assignment. ∀F:Formula. Dec{a |= F}
 Extraction:
  λa,F.((λ%1.%1 (F under a)  2₃) ext{decidable_equal_Three})

*T decidable_formula_falsifiable:
 ∀a:Assignment. ∀F:Formula. Dec{a |≠ F}
 Extraction:
  λa,F.((λ%1.%1 (F under a)  0₃) ext{decidable_equal_Three})
```

The functions extracted from the formal proofs reflect the informal argument just given, *i.e.* they accept as arguments an assignment `a` and a formula `F` and then apply the decision procedure for equality over **3** to the terms (F under a) and 2_3 (0_3.). The function deciding equality over **3** is referred to by the term ext{decidable_equal_Three} which denotes the extract of the theorem decidable_equal_Three. Reference to the extract of a previously proved lemma arises by reference to the previously proved lemma in the proof.

Some useful lemmas follow immediately from the definitions of satisfiability and falsifiability. These lemmas relate semantic notions with syntactic structure by characterizing the satisfiability (or falsifiability) of a formula in terms of its subformulas.

```
*T formula_not_sat_lemma
   ∀F:Formula. ∀a:Assignment.
       a |= ⌈∼⌉F ⟺ a |≠ F
*T formula_not_falsifiable_lemma
   ∀F:Formula. ∀a:Assignment.
       a |≠ ⌈∼⌉F⟺ a |= F
*T formula_and_sat_lemma
   ∀a:Assignment. ∀q:Formula. ∀r:Formula.
       a |= q⌈∧⌉r ⟺ a |= q  ∧ a |= r
*T formula_and_falsifiable_lemma
   ∀a:Assignment. ∀q,r:Formula.
       a |≠ q⌈∧⌉r⟺ a |≠ q ∨ a |≠ r
*T formula_or_sat_lemma
   ∀a:Assignment. ∀q,r:Formula.
       a |= q⌈∨⌉r ⟺ a |= q  ∨ a |= r
*T formula_or_falsifiable_lemma
   ∀a:Assignment. ∀q,r:Formula.
       a |≠ q⌈∨⌉r⟺ a |≠ q ∧ a |≠ r
*T formula_imp_sat_lemma
   ∀a:Assignment. ∀q,r:Formula.
       a |= q⌈⇒⌉r ⟺ a |≠ q ∨ a |= r
*T formula_imp_falsifiable_lemma
   ∀a:Assignment. ∀q,r:Formula.
       a |≠ q⌈⇒⌉r⟺ a |= q  ∧ a |≠ r
```

These lemmas are proved by unfolding the definitions of formula_sat and formula_falsifiable and then case analysis. These characterizations have been shown to hold in both directions ($\Longleftrightarrow$) but are typically applied as rewrites in a left to right form ($\Rightarrow$).

3.2.5 A Lemma concerning Kleene Valuations

In this section we show that the use of the semantics based on the three valued Kleene valuation coincides with the standard two valued semantics.

Given assignments `a'` and `a` we define the *restriction* of `a'` to `a` (denoted (`a'↓a`) to be the assignment that is undefined (*i.e.* equal to $\mathtt{1}_3$) whenever `a` is undefined and that agrees with `a'` everywhere else. The formal definition is as follows.

```
*A restriction
a'↓a == λx.case (a x): 3₀ → (a' x); 3₁ → 3₁; 3₂ → (a' x);
```

The well-formedness theorem establishes that `a'↓a` is in fact an assignment.

```
*T restriction_wf
∀a,a':Assignment. (a↓a' ∈ Assignment)
```

If the restriction of an assignment `a'` to an assignment `a` is identical with `a` we say `a'` is an *extension* of `a` or `a'` *extends* `a`. We formalize this notion in the following abstraction.

```
*A extension
a' extends a == a'↓a = a ∈ Assignment
```

The well-formedness goal for the the definition asserts that it is indeed a proposition.

```
*T extension_wf
  ∀a,a':Assignment. (a' extends a ∈ ℙ{i})
```

The following lemma characterizes the notion of extension and verifies the main property of interest; specifically, if `a'` extends `a` then `a'` and `a` agree on every variable for which `a` is defined.

```
*T extension_lemma
∀a,a':Assignment
   a' extends a ⇒
     (∀x:Var. ¬((a x) = 3₁ ∈ ℕ₃) ⇒ ((a x) = (a' x) ∈ ℕ₃))
```

Having formally defined the notion of one assignment being an extension of another we can state a theorem justifying the Kleene valuation semantics with respect to the standard two valued semantics.

```
*T assignment_monotone
∀a,a':Assignment
   ∀F:Formula.
      a' extends a ⇒
        (a |= F  ⇒ a' |= F ) ∧ (a |≠ F ⇒ a' |≠ F)
```

The lemma is proved by induction on the structure of the formula `F`. It tells us that any assignment extending a satisfying (falsifying) assignment for a formula `F` is also a satisfying (falsifying) assignment of `F`. Extensions of partial assignments not having the value 1_3 in their range comprise the standard two valued assignments justifying our use of three valued semantics based on the Kleene operators.

3.2.6 Fullness and Validity

Although we are ultimately interested in determining the validity of sequents, introducing the concepts of fullness and validity in relation to formulas first is worthwhile. The definitions presented in this section are not used in the decidability proof itself but do serve to illustrate fullness and validity in the simpler context of formulas. These definitions *are* used to prove a theorem characterizing the relationship between validity of formulas and validity of sequents.

In two-valued semantic presentations, a formula F is said to be *valid* when

$$\forall a : \texttt{Assignment}.\, a \models F.$$

However, under this definition there are no valid sentences of Kleene's three valued logic. To see why consider the constant assignment $(\boldsymbol{\lambda}\texttt{x}.\,1_3)$; examination of the matrices for the Kleene operators shows that *no* formula is *true* under this assignment. Thus, if validity requires a formula to be true under every Kleene valuation, there are no valid formulas. Validity was not at issue for Kleene who used the logic for reasoning about partial recursive predicates; for us, an acceptable notion of validity is crucial.

Toward this end we will say an assignment is *full* for a formula `F` if the assignment either satisfies `F` or falsifies `F`. For example, let `a` be the assignment that maps variable `x` to the value 2_3 and maps all other variables to the undefined value, 1_3. Then (⌈x⌉ under a) evaluates to 2_3 and so satisfies the formula ⌈x⌉; `a` is full for the formula ⌈x⌉. On the other hand, (⌈x⌉⌈⇒⌉⌈y⌉ under a) evaluates to (⌈y⌉ under a) which in turn valuates to 1_3 and thus `a` is not full for the formula ⌈x⌉⌈⇒⌉⌈y⌉. This notion of fullness allows for consideration of only those assignments that contain "enough information" to completely determine the value of a formula. In the formalization, full assignments are defined as a subtype of `Assignment`.

```
*A full_formula_assignment
    Full(F) == {a:Assignment| (a |= F  ∨ a |≠ F)}
```

Thus, `Full(F)` is the type of assignments satisfying the fullness predicate for formula `F`. In Nuprl, the computational content associated with the proof that an element satisfies the defining predicate of a set type, proving it is in fact an inhabitant, is not available unless explicitly provided. The fact that an element of the set type satisfies the defining predicate is "hidden". In this case it would mean that the fact that an assignment inhabiting the type `Full(F)` satisfies `F` or falsifies `F` could not be used unless evidence (proofs) were provided for every new element of the type. Technically, the information associated with the defining predicate is available in some contexts when discharging well-formedness goals and in some kinds of equality reasoning. This exception is possible because equality reasoning and well-formedness reasoning do not contribute to computational content, *i.e.* proofs of well-formedness and equality reasoning are not used in the construction of the extracted program. To be unable to use the fact that a variable declared to be in the set type satisfies the defining predicate limits the usefulness of the set type to those having defining predicates which are uniformly decidable.

It was shown above that `formula_sat` and `formula_falsifiable` are decidable properties; hence, for any formula `F` and any assignment `a`, it can be uniformly decided whether `a` is in the type `Full(F)` or not. In general, if the defining predicate of a set type is decidable we may disregard the restrictions on the use of the properties specified by the defining predicate; this is because the corresponding computational content can be reconstructed at will. The tactics for manipulating set types take advantage of this fact by searching for so-called property lemmas.

For abstractions defined by set types, the decomposition tactics will automatically unfold the set type definitions and unhide hidden hypotheses if a lemma is found in the library with name ⟨*opid_name*⟩`_properties` which asserts that the defining predicate holds for arbitrary elements of the type.

The following property lemma is used by the Nuprl decomposition tactics to unfold abstractions defined by set types and to unhide the resulting hypotheses.

```
*T full_formula_assignment_properties
```

```
∀F:Formula. ∀a:Full(F). a |= F  ∨ a |≠ F
```

This lemma is proved by eliminating the outermost quantifiers and then decomposing the type `Full(F)`. This results in the following sequent.

```
 1.  F:Formula
 2.  a: Assignment
 [3]. a |= F  ∨ a |≠ F
 ⊢ a |= F  ∨ a |≠ F
```

Hidden hypotheses are labeled hidden by the square brackets surrounding their hypothesis numbers.

The proof is trivial if hypothesis 3 can be unhidden. To do this we assert its decidability which results in two subgoals: one to show that the disjunction is in fact decidable; and the second to show the original goal under the additional hypothesis of decidability of the disjunct. The first subgoal is reduced to trivial subgoals by the `ProveDecidable` tactic which establishes the decidability of the disjunct using a lemma in the library characterizing when disjunctive formulas are decidable (*i.e.* whenever the principal sub-terms are too.) The second subgoal generated by asserting `dec{a |= F ∨ a |≠ F}` is discharged by applying the `UnhideHyp` tactic to the hidden hypothesis. This results in a subgoal requiring us to show that the decidable predicate is squash stable which in turn is discharged by applying the `ProveSqStable` tactic completing the proof of the properties lemma.

Using the definition of fullness just given we formalize the notion of validity as follows.

```
*A formula_valid         |= F == ∀a:Full(F). a |= F
```

Thus, a formula is valid when it is satisfied by every full assignment. If an assignment contains enough information to determine the truth or falsity of a formula and every such assignment corroborates the truth of the formula then the formula is valid.

3.3 Sequents

Sequents are formalized as pairs of lists of formulas; of course, other options are possible, pairs of bags (multi-sets) of formulas chief among them.

```
*A Sequent:  Sequent == Formula list × Formula list
```

Another trivial inclusion lemma is provided for the type checking tactics.

```
*T Sequent_inc:  Sequent ⊆ (Formula list × Formula list)
```

A functional interface is provided for decomposing sequents into the hypothesis and conclusion lists by the `H` and `C` abstractions.

```
*A H:   s.H == let h,c = s in h
*A C:   s.C == let h,c = s in c
```

3.3.1 Sequent Rank

Before defining a measure on sequents we define the rank of a list of formulas; it is simply the sum of the ranks of the formulas occurring in the list.

```
*A list_rank     ρ == λL.reduce((λx,y.((ρ x) + y));0;L)
*T list_rank_wf  ρ ∈ (Formula list → ℕ)
```

This definition uses the `reduce` operator on lists which accepts three arguments: a two argument function which is right associative; an identity for the operator; and a list. Note that we have not distinguished the display forms for the `formula_rank` function (which is used within the definition of the right associative operator) and the `list_rank` function being defined here; they share the same display in the system as well. The same display is used for the `sequent_rank` function defined below. It is clear from the context which operator is being used, and, in the system, should it be confusing at any point which operator a display denotes a click of a mouse button distinguishes them.

A useful property of `list_rank` is that it, in some sense, "distributes" over list append (denoted in infix notation here by @.) More formally we say `list_rank` is homomorphic with respect to append and addition. Of course this property can be formulated more abstractly in that any right associative operation (addition in this case) iteratively applied to list, is homomorphic with the append operator; the following theorem is a special case of that fact and is useful in the decomposition of ranked lists.

```
*T list_rank_append_homomorphism
   ∀M,N:Formula list. ((ρ M @ N) = ((ρ M) + (ρ N)) ∈ ℕ)
```

Using the `list_rank` just defined, the rank of a sequent is simply defined to be the sum of the ranks of the hypothesis and conclusion lists.

```
*A sequent_rank     ρ == λS.((ρ S.H) + (ρ S.C))
*T sequent_rank_wf  ρ ∈ (Sequent → ℕ)
```

We call sequents having rank 0 *atomic* sequents.

3.3.2 Sequent satisfiability and falsifiability

In this section the semantics of sequents is given. First the meaning of a sequent is given in informal mathematical terms and then this definition is translated into the three-valued model being developed here.

A sequent S is of the form $\langle [H_1, H_2, \ldots, H_n], [C_1, C_2, \ldots, C_m]\rangle$, where $[H_1, \ldots, H_n]$ and $[C_1, \ldots, C_m]$ are lists of formula corresponding to the hypothesis and conclusion respectively. S is interpreted to be true precisely when the conjunction of the hypotheses implies the disjunction of the conclusions.

$$(H_1 \wedge \cdots \wedge H_n) \Rightarrow (C_1 \vee \cdots \vee C_m)$$

Adopting the convention that an empty conjunction denotes truth and the empty disjunction denotes falsity, $\langle [H_1, \ldots, H_n], [\,]\rangle$ means $\neg H_1 \vee \cdots \vee \neg H_n$, $\langle [\,], [C_1, \ldots, C_m]\rangle$ means $C_1 \vee \cdots \vee C_m$, and the empty sequent, $\langle [\,], [\,]\rangle$, denotes an unsatisfiable sequent.

The discussion above follows the standard presentation for a two valued semantics of sequents; here however, as for formula above, we are interested in the satisfaction of sequents under Kleene valuations induced by partial assignments. Adapting the discussion above to the analogous definition under Kleene interpretation, which we've already defined for formulas, we find the following.

A partial assignment $\mathtt{a}$ *satisfies* a sequent $\langle [H_1, \ldots, H_n], [C_1, \ldots, C_m]\rangle$ if and only if

$$\mathtt{a} \models (H_1 \lceil\wedge\rceil \cdots \lceil\wedge\rceil\, H_n) \lceil\Rightarrow\rceil (C_1 \lceil\vee\rceil \cdots \lceil\vee\rceil\, C_m)$$

where $\models$ is the formula satisfaction relation defined above using the Kleene interpretation induced by $\mathtt{a}$. Similarly, an assignment *falsifies* a sequent $\langle [H_1, \ldots, H_n], [C_1, \ldots, C_m]\rangle$ if and only if

$$\mathtt{a} \not\models (H_1 \lceil\wedge\rceil \cdots \lceil\wedge\rceil\, H_n) \lceil\Rightarrow\rceil (C_1 \lceil\vee\rceil \cdots \lceil\vee\rceil\, C_m)$$

These definitions could be formalized as presented and would serve to define the semantic notions of sequent satisfiability, sequent falsifiability and sequent validity; however, the properties of the operators for conjunction, disjunction and implication under the Kleene valuation suggest a computationally simpler characterization. Under the definition above, a sequent is satisfiable under an assignment a either when there is some hypothesis that is falsified by a or there is some formula in the conclusion that is satisfied by a. This suggests the following definition.

```
*A sequent_satisfiable
a |= S == ∃F∈S.H.a |≠ F ∨ ∃F∈S.C.a |= F
```

Similarly, a sequent S is falsifiable under an assignment a if every hypotheses of S is satisfied by a and every conclusion of S is falsified by a. Again, this is the formal definition adopted here.

```
*A sequent_falsifiable
a |≠ S == ∀F∈S.H.a |= F  ∧ ∀F∈S.C.a |≠ F
```

These definitions exhibit the first use of the bounded list quantification operators. The list existence quantifier is non-void (true) if, for any member x of the list L, the predicate P[x] is non-void. Thus, for empty lists it is false. Similarly, the list forall quantifier is true if every x in L satisfies P[x]. For the empty list, the quantifier is vacuously true.

It can effectively be decided whether a sequent is satisfied or falsified by an assignment; this follows from the decidability of the same properties for formulas. These facts are captured in the following two decidability theorems.

```
*T decidable__sequent_satisfiable:
  ∀S:Sequent. ∀a:Assignment. Dec{a |= S}
*T decidable__sequent_falsifiable:
 ∀S:Sequent. ∀a:Assignment. Dec{a |≠ S}
```

3.3.3 Full Sequent Assignments and Sequent Validity

The analogue of fullness of an assignment for a formula is now defined for sequents.

```
*A full_sequent_assignment
   Full(S) == {a:Assignment| (a |= S ∨ a |≠ S)}
```

Again, we provide a trivial sub-typing lemma and a properties lemma for use by the decomposition tactics.

```
*T full_sequent_assignment_inc
   ∀S:Sequent. Full(S) ⊆ Assignment
*T full_sequent_assignment_properties
   ∀S:Sequent. ∀a:Full(S). a |= S ∨ a |≠ S
```

Validity can now be defined with respect to fullness.

```
*A sequent_valid   |= S  == ∀a:Full(S). a |= S
```

If sequent validity has the relationship to formula validity we expect, then for every formula `F`, the sequent ⟨`[],F::[]`⟩ should be valid precisely when `F` itself is. This is captured by the following theorem which is easily proved by unfolding the definitions for sequent validity and formula validity followed by some steps of computation.

```
*T formula_valid_iff_sequent_valid
   ∀F:Formula. |= F ⟺ |= <[],F::[]>
```

4 Decidability

Our goal is to prove decidability of propositional logic, *i.e.* to show that we can decide if a propositional sequent is valid. The most natural formalization of the theorem would simply say

```
∀S:Sequent. |= S ∨ ¬(|= S)
```

A proof of this theorem would yield a function accepting a sequent as an argument and then returning an `inl` term or an `inr` term, depending on whether the sequent was valid or not. But we know a proposition is not valid if and only if there is some full assignment which falsifies it.

```
∀S:Sequent.¬(|= S) ⟺ (∃a:Assignment. a|≠ S).
```

Using this logically equivalent form of unsatisfiability we state a computationally stronger version of the decidability theorem.

```
*T propositional_decidability
   ∀S:Sequent. |= S  ∨ (∃a:Assignment. a |≠ S)
```

That is, every sequent is either valid or there exists an assignment which falsifies it. The revised version of the theorem is stronger in the sense that we can extract more interesting computational content from its proof. A witness for the theorem is a function of type

```
S:Sequent → (|= S  |  a:Assignment × a |≠ S)
```

Thus, a constructive proof of the theorem results in a function accepting a sequent `S` as its argument and returning one of `inl(t)` or `inr(⟨a,e⟩)`. The term `t` under the injection `inl` has little interest for us; however, the pair `⟨a,e⟩` under the `inl` injection is most interesting. The first element of the pair is an assignment falsifying the proposition. This assignment provides important diagnostic information telling exactly when the proposition is false. We have formalized the semantics in terms of partial assignments to be able to refine the information content in the falsifying assignment further: depending on the form of the proof, the falsifying assignment returned by the procedure can be minimal in the sense that only those variables contributing to the falsification of the formula are assigned one of `true` or `false`, all others are left `undefined`. Of course, this depends on the form of the proof; it is shown below where the partiality plays a part.

4.1 A strategy for the proof

We say a collection of sequents has the *collective validity property* with respect to a sequent `S` if the validity of the collected sequents implies the validity of `S`. A collection of sequents has the *individual falsifiability property* with respect to a sequent `S` if any assignment falsifying any element of the collection also falsifies the `S`.

The proof of the decidability theorem is based on the following two observations:

1.) atomic sequents are easily decided, and

2.) given an occurrence of non-atomic formula f in a sequent S, there is a method of eliminating the principal operator of f which induces one or two sequents such that:

- **i.)** The induced sequents contain fewer operators than the original sequent (and thus are of smaller rank),
- **ii.)** the induced sequents collectively validate `S` (if the induced sequents are valid then so is the `S`), and
- **iii.)** the induced sequents enjoy individual falsifiability with respect to `S` (if any of the induced sequents are falsified by an assignment then that assignment falsifies the original sequent too.)

These observations suggest an approach to deciding sequents.

Observation (2) suggests the existence of a one-step normalization procedure for sequents that results in a collection of sequents having smaller rank and, if they are all valid, which collectively preserve validity and, if any of them are falsifiable, they individually entail the falsifiability of the original. By transitivity of implication, the repeated application of the one-step normalization procedure would result in a collection of atomic sequents whose collective validity implies the validity of the goal and if any are individually falsifiable, then the falsifying assignment falsifies the original sequent too. The existence of a collection is formalized by the following lemma.

```
* THM normalization_lemma
∀G:Sequent
   ∃L:Sequent list
    ∀s∈L.ρ s = 0 ∧
   (∀s∈L.|= s)  ⇒ |= G ∧
   (∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ G)
```

A slightly generalized form of observation (1) is formalized in the following lemma which says that for every list of atomic (zero rank) sequents, either they are all valid or there is some assignment falsifying some sequent in the list.

```
* THM zero_rank_valid_or_falsifiable
   ∀L:Sequent list.
      (∀s∈L. ρ s = 0)  ⇒
        ∀s∈L.|= s  ∨ (∃a:Assignment. ∃s∈L.a |≠ s)
```

The lemmas `zero_rank_valid_or_falsifiable` and `normalization_lemma` are proved below. First we give the proof of propositional decidability using these two lemmas.

4.2 Decidability proof

Recall the statement of the theorem.

```
⊢ ∀S:Sequent. |= S  ∨ (∃a:Assignment. a |≠ S)
```

The first step of the proof is to decompose the outermost universally quantified variable `S` resulting in the declaration of a new variable `S` of type `Sequent`. The second step of the proof is the instantiation of the normalization lemma with the sequent `S` as the goal. In the same step, the instantiated lemma is repeatedly decomposed resulting in four additional hypotheses and the following Nuprl sequent.

```
1. S: Sequent
2. L: Sequent list
3. ∀s∈L.ρ s = 0
4. ∀s∈L.|= s  ⇒  |= S
5. ∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ S
⊢ |= S  ∨ (∃a:Assignment. a |≠ S)
```

Hypothesis 2 asserts the existence of a list L of type Sequent. Hypotheses 3 through 4 assert that L contains only rank zero sequents which collectively validate S and individually falsify S. The next step in the proof is to forward chain through the zero_rank_valid_or_falsifiable lemma with hypothesis 3 asserting L consists of only zero rank sequents. This instantiates the universally quantified list variable in the lemma with L leaving a disjunction asserting that either all elements of L are valid or some element of L is falsifiable.

```
6. ∀s∈L.|= s  ∨ (∃a:Assignment. ∃s∈L.a |≠ s)
⊢ |= S  ∨ (∃a:Assignment. a |≠ S)
```

A case split on the disjunction in hypothesis 6 yields two cases.

```
6. ∀s∈L.|= s
⊢ |= S  ∨ (∃a:Assignment. a |≠ S)
```

If every sequent in L is valid, then by hypothesis 4 (collective validity) the goal sequent S is valid. This is established by forward chaining through hypothesis 4 with the assumption that every sequent in L is valid (hypothesis 6). Choosing the left disjunct of the conclusion completes this branch of the proof.

In the second case there is an assignment which falsifies some sequent in L. Decomposing hypothesis 6 for the second disjunct yields the following Nuprl goal.

```
5. ∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ S
6. a: Assignment
7. ∃s∈L.a |≠ s
⊢ |= S  ∨ (∃a:Assignment. a |≠ S)
```

Forward chaining through hypothesis 5 with hypothesis 7 which asserts that a falsifies some element of L yields the fact that a falsifies the sequent S.

```
8. a |≠ S
⊢ |= S  ∨ (∃a:Assignment. a |≠ S)
```

Choosing the right disjunct of the conclusion and discharging the existentially quantified variable with the witness `a` completes the proof of propositional decidability.

4.3 Deciding zero rank sequents

Recall the statement of the lemma asserting that, for all lists of zero-rank sequents either all sequents in the list are valid or there is an assignment which falsifies at least one of them.

```
* THM zero_rank_valid_or_falsifiable
   ∀L:Sequent list.
      (∀s∈L. ρ s = 0)  ⇒
        ∀s∈L.|= s  ∨ (∃a:Assignment. ∃s∈L.a |≠ s)
```

The proof rests on the observation that a sequent ⟨`hyp,concl`⟩ containing only atomic formulas is falsifiable if and only if the hypothesis list `hyp` and the conclusion list `concl` are disjoint lists (and is valid otherwise.) The property of lists being disjoint is decidable if the type of the lists is a discrete type. Thus the proof of the lemma is split into two cases, *i.e.* depending on whether every sequent in `L` is disjoint or not. Before the case split we introduce discrete equalities for the types `Sequent` and `Formula`, these equalities are used when searching for a sequent in a list and for determining if the sequents in the list are disjoint. After the introduction of the equalities and the case split, the Nuprl goal after the first case split appears as follows.

```
1. L: Sequent list
2. eqS: {Sequent=₂}
3. eqF: {Formula=₂}
4. ∀s∈L.ρ s = 0
5. ∃s∈L.disjoint(eqF;s.H;s.C)
⊢ ∀s∈L.|= s  ∨ (∃a:Assignment. ∃s∈L.a |≠ s)
```

In this case, by hypothesis 5, there is a sequent `s` in `L` such that its hypothesis list `s.H` and its conclusion list `s.C` are disjoint. This is rewritten using `list_exists_is_member_lemma` to give the following more explicit Nuprl goal.

```
 6. hyp: Formula list
 7. concl: Formula list
 8. ↑(<hyp,concl>(∈eqS) L)
 9. disjoint(eqF;hyp;concl)
 ⊢ ∀s∈L.|= s  ∨ (∃a:Assignment. ∃s∈L.a |≠ s)
```

We proceed with the proof by choosing the second disjunct of the conclusion as the new goal. To prove it we must construct a falsifying assignment for some sequent in `L`. We falsify the sequent `<hyp,concl>` by: mapping variables `x` such that the formula ⌈`x`⌉ occurs in `hyp` to 2_3 (true); mapping variables `y` such that ⌈`y`⌉ occurs in `concl` to 0_3 (false); and mapping all other variables to 1_3 (undefined). In the proof, we assert that this function is in fact an assignment by the following tactic invocation.

```
Assert '∃a:Assignment. a = (λv.if ⌈v⌉(∈eqF) x.H then 3₂
                               else if  ⌈v⌉(∈eqF) x.C then 3₀
                               else 3₁ fi
                               fi) ∈ Assignment'
```

After discharging the obligation showing the assertion is in fact true, *i.e.* that the lambda term does indeed inhabit type `Assignment`, we use the assignment just declared to discharge the existential quantified variable in the conclusion. This results in the following Nuprl proof node.

```
 6. hyp: Formula list
 7. concl: Formula list
 8. ↑(<hyp,concl>(∈eqS) L)
 9. disjoint(eqF;hyp;concl)
 10. a: Assignment
 11. a = (λv.if ⌈v⌉(∈eqF) hyp then 3₂ else
              if ⌈v⌉(∈eqF) concl then 3₀ else 3₁ fi
            fi ) ∈ Assignment
 ⊢ ∃s∈L. a |≠ s
```

Discharging the existential list quantifier in the conclusion with the sequent `<hyp,concl>`, unfolding the definition of `sequent_falsifiable`, and computing with the falsifying assignment completes this branch of the proof. This proves that if there is any sequent in `L` having disjoint hypothesis list and conclusion lists, there is a falsifiable sequent in `L`.

In the other case, no sequent in `L` has disjoint hypothesis and conclusions lists. We take up this case with the following Nuprl goal.

```
5. ¬∃s∈L.disjoint(eqF;s.H;s.C)
⊢ ∀s∈L.|= s  ∨ (∃a:Assignment. ∃s∈L.a |≠ s)
```

Rewriting the negated existential list quantifier to a positive form, choosing the first disjunct in the conclusion and then eliminating the leading list_all quantifier, and reasoning about the non-disjointedness of the sequent <hyp,concl> yields the following Nuprl goal.

```
5. ∀s∈L.(¬disjoint(eqF;s.H;s.C))
6. hyp: Formula list
7. concl : Formula list
8. x: Var
9. ↑(⌜x⌝(∈eqF) hyp)
10. ↑(⌜x⌝(∈eqF) concl)
⊢ |= <hyp,conc>
```

Unfolding the definition of sequent validity and then performing some reduction steps we derive the following Nuprl proof node.

```
11. a: Assignment
12. a |= <hyp,concl> ∨ a |≠ <hyp,concl>
⊢ a |= <hyp,concl>
```

Doing a case split on hypothesis 12, the first subgoal is trivially true. In the second case, hypothesis 12 is as follows.

```
12. a |≠ <hyp,concl>
⊢ a |= <hyp,concl>
```

A contradiction is induced by the hypotheses. Hypothesis 12, asserts that assignment a falsifies the sequent <hyp,concl>. Hypotheses 9 and 10 which assert that ⌜x⌝ occurs in both hyp and concl. Unfolding the definition of a |≠ <hyp,concl> we obtain the following goal.

```
12. ∀f∈hyp. a |= f  ∧ ∀f∈concl. a |≠ f
⊢ a |= <hyp,concl>
```

Thus, any assignment falsifying the sequent <hyp,concl> must satisfy all formulas in hyp and must falsify all formulas in concl. Since ⌜x⌝ is in both lists, this induces the contradiction; the assignment a cannot both satisfy and falsify the same formula.

This completes the proof of the lemma.

Before moving to the proof of the normalization lemma some remarks about this proof are in order. The first thing to notice is that this is the only place in the decidability proof where a computable equality for sequents is required. The equality is used in the step eliminating an existential list operator in the branch of the proof that considers the case that there is some non-disjoint sequent in the list `L`. The discrete equality on formulas is needed to search for the atomic formula $\ulcorner$`x`$\urcorner$ that is a member of both the hypothesis and conclusion lists.

The assignment chosen to falsify the sequent, having non-disjoint hypotheses and conclusions, is a partial assignment. Indeed, it only assigns values to variables occurring in the non-disjoint sequent. Since not every variable occurring in a sequent contributes content to a falsifying assignment the partial assignment used here provides more information, both in which variables are assigned false and true but also by showing which variables do not affect the falsification of the sequent by the particular assignment returned. In fact, the proof goes through essentially unchanged when the following total assignment is used.

λ`v.if` $\ulcorner$`v`$\urcorner$`(`$\in$`eqF) x.H then` 3_2 `else` 3_1 `fi`

The development of a three valued type and the Kleene operators presented here with the Kleene valuation is more complex, but the end result is tighter information on the falsifying assignment returned by the procedure since we can identify "don't care" variables in the counter-example.

4.4 The Normalization Proof

Decidability rests on the properties of the list `L` whose existence is established by the normalization lemma. The proof of this lemma provides the core of the computational procedure. The proof is by well-founded induction and so yields a recursive procedure that maps a single sequent `G` to a list of atomic sequents which collectively validate `G` or at least one of which falsifies `G`. After presenting the detailed proof here we will show how the inductive cases give rise to a natural set of sequent proof rules.

Recall the statement of the lemma.

```
* THM normalization_lemma
∀G:Sequent
```

```
∃L:Sequent list
 ∀s∈L.ρ s = 0 ∧
(∀s∈L.|= s)  ⇒ |= G ∧
(∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ G)
```

The proof proceeds as follows. After stripping off the universally quantified variable `G`, the inverse image induction tactic is invoked with `sequent_rank` as the measure, mapping sequents to natural numbers and with complete induction on the type ℕ(the natural numbers) as the induction principle. The inverse image induction tactic is invoked as follows.

```
New [‘S’] (InvImageInd ’ρ’ ’ℕ’ CompNatInd 1 THENA Auto)
```

Applying this tactic and then decomposing the goal sequent `G` into component formula lists, `hyp` and `concl`, results in the following Nuprl proof node.

```
1. hyp: Formula list
2. concl: Formula list
3. ∀S:Sequent
    ρ S < ρ <hyp,concl>
    ⇒ (∃L:Sequent list
         ∀s∈L.ρ s = 0 ∧
        (∀s∈L.|= s  ⇒ |= S ) ∧
        (∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ S))
⊢ ∃L:Sequent list
|   ∀s∈L.ρ s = 0
|    ∧ (∀s∈L.|= s  ⇒ |= <hyp,concl> )
|    ∧ (∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ <hyp,concl>)
```

Hypothesis 3 is the induction hypothesis we expect to see. The sequent to the right of the less-than in the induction hypothesis will be referred to as the *goal sequent* or simply as the *goal*. In this case the goal is the sequent `<M @ (⌈∼⌉x::N),concl>`. The proof of the normalization lemma proceeds by inductively decomposing non-zero rank elements of the sequent `<hyp,concl>`, if there are any; if not we directly argue the theorem holds. In the inductive cases (cases where there is a non-atomic formula in the sequent) we construct one or, in some cases, two “smaller” sequents each of whose validity implies the validity of the goal sequent and either of whose falsifiability implies the falsifiability of the goal. We refer to these smaller sequents as *eliminants*.

Thus, to proceed with the proof we case split on whether the list `hyp` contains any non-zero rank formula. This split is accomplished by the following tactic invocation.

```
Decide '∃f∈hyp.ρ f > 0' THENA Auto
```

Since the `list_exists` operator is decidable whenever the predicate is, and since `ρ f > 0` is decidable, the application of the tactic results in only two proof branches. Had the system been unable to infer the decidability of `∃f∈hyp.ρ f > 0` from the library, it would have included its decidability as a third subgoal.

The two new branches share the same proof goal as the previous proof goal but each has an additional hypothesis: one declaring that there is a non-atomic formula in `hyp`

```
4. ∃f∈hyp.ρ f > 0
```

and the other declaring that all formulas in the list `hyp` are atomic.

```
4. ¬∃f∈hyp.ρ f > 0
```

In the second case, we must further consider whether `concl` contains any non-atomic formula. Again this results in two cases, one having the hypotheses that `hyp` is atomic (or empty) and that the conclusion contains a non-atomic formula;

```
4. ¬∃f∈hyp.ρ f > 0
5. ∃f∈concl.ρ f > 0
```

The remaining case asserts that neither `hyp` nor `concl` contain any non-atomic formula.

```
4. ¬∃f∈hyp.ρ f > 0
5. ¬∃f∈concl.ρ f > 0
```

We consider this last case first.

4.4.1 The sequent is atomic

In this section we discharge the case that both the hypothesis list `hyp` and the conclusion list `concl` contain only zero-rank formulas, *i.e.* that the original sequent `G` was atomic.

First we assert the fact that the sequent `<hyp,concl>` is atomic, *i.e.* that ρ `<hyp,concl> = 0`. The subgoal requiring the proof of the assertion itself is discharged by applying the facts that neither `hyp` nor `concl` contain any formula having rank greater than zero. Next, the existential quantifier in the goal of the Nuprl sequent is instantiated by the list containing the single sequent `<hyp,concl>`. This step is accomplished by the following tactic invocation:

```
With '<hyp,concl>::[]' (D 0) THENA Auto
```

and results in the following proof node.

```
4. ¬∃f∈hyp.ρ f > 0
5. ¬∃f∈concl.ρ f > 0
6. ρ <hyp,concl> = 0
⊢ ∀s∈(<hyp,concl>::[]).ρ s = 0
|   ∧ (∀s∈(<hyp,concl>::[]).|= s  ⇒ |= <hyp,concl> )
|   ∧ (∀a:Assignment.
|      ∃s∈(<hyp,concl>::[]).a |≠ s ⇒ a |≠ <hyp,concl>)
```

Performing some steps of computation on hypothesis `6` and in the goal, followed by an application of the Nuprl auto-tactic yield the following proof node.

```
6. (ρ hyp) + (ρ concl) = 0
7. a: Assignment
8. a |≠ <hyp,concl> ∨ False
⊢ a |≠ <hyp,concl>
```

This is goal trivially proved by applying the tactic `D (-1) THEN Trivial`. Thus, we have finished the case where the sequent `G = <hyp,concl>` is atomic.

4.4.2 The hypothesis contains non-atomic formula

Now we backup to the case where the formula list `hyp` contains a non-zero rank formula.

Throughout the proof presented here, whenever an existential property (P) is asserted to hold for a list L, *i.e.* P[x] holds for some element x in L, we use the following lemma to decompose the list, explicitly naming an element of the list having the property.

```
*T list_exists_is_member_append_lemma
   ∀T:𝕌. ∀P:T → ℙ. ∀L:T List.
     ∃x∈L.P[x] ⟺ (∃M,N:T List. ∃x:T. P[x] ∧ L = M @ (x::N))
```

Forward chaining through hypothesis 4 with this lemma and decomposing the resulting existential hypotheses yields the following proof state.

```
 4. ∃f∈hyp.ρ f > 0
 5. M: Formula list
 6. N: Formula list
 7. f: Formula
 8. ρ f > 0
 9. hyp = M @ (f::N) ∈ (Formula list)
 ⊢ ∃L:Sequent list
 |   ∀s∈L.ρ s = 0
 |   ∧ (∀s∈L.|= s  ⇒ |= <hyp,concl> )
 |   ∧ (∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ <hyp,concl>)
```

From this point, the proof proceeds by case analysis on the variable f declared in hypothesis 7. The FormulaCase tactic does case analysis on variables of type Formula resulting in five subgoals, one for each formula class, *i.e.* whether the formula is a variable, a negation, a conjunction, a disjunction or an implication. The five subgoals and their proofs are presented below. We do not present the case split and intervening clean-up steps but take up the proofs after these proof steps have been performed.

4.4.2.1 A variable on the left In the first case the non-zero rank formula is of the form ⌈x⌉ where x is an element of type Var.

```
 8. x: Var
 9. ρ ⌈x⌉ > 0
 ⊢ ∃L:Sequent list
 |   ∀s∈L.ρ s = 0
 |   ∧ (∀s∈L.|= s  ⇒ |= <hyp,concl> )
 |   ∧ (∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ <hyp,concl>)
```

Reducing the application ρ ⌈x⌉ yields the contradictory hypothesis `0 > 0` and this branch of the proof is discharged by the auto-tactic.

4.4.2.2 Negation on the left In this case the non-zero rank formula occurring in the list `hyp` is a negation. This is the first case that requires the application of the induction hypothesis. The form of the Nuprl sequent to be discharged is as follows.

```
8. x: Formula
9. ∀S:Sequent
     ρ S < ρ <M @ (⌈∼⌉x::N),concl>
     ⇒ (∃L:Sequent list
          ∀s∈L.ρ s = 0
          ∧ (∀s∈L.|= s  ⇒ |= S )
          ∧ (∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ S))
⊢ ∃L:Sequent list
|    ∀s∈L.ρ s = 0
|    ∧ (∀s∈L.|= s  ⇒ |= <M @ (⌈∼⌉x::N),concl> )
|    ∧ (∀a:Assignment.
|          ∃s∈L.a |≠ s ⇒ a |≠ <M @ (⌈∼⌉x::N),concl>)
```

To prove this we must find a sequent `S` such that:

```
(i.)    ρ S < ρ <M @ (⌈∼⌉x::N),concl>
(ii.)   |= S ⇒ |=  <M @ (⌈∼⌉x::N),concl>
```
and
```
(iii.) ∀a:Assignment. a |≠ S ⇒ a |≠ <M @ (⌈∼⌉x::N),concl>
```

The eliminate `S` is constructed from `<M @ (⌈∼⌉x::N),concl>` by removing the negation ⌈∼⌉`x` from the hypothesis and prepending the negated formula `x` to the conclusion. The following tactic invocation instantiates the induction hypothesis with this sequent.

```
With '<M @ N,x::concl>' (D (-1) THENA Auto) THEN
  Repeat (D (-1))
```

Two subgoals result.

The first is to show the antecedent of the induction hypothesis holds.

```
⊢ ρ <M @ N,x::concl> < ρ <M @ (⌈∼⌉x::N),concl>
```

The tactic `SequentRankReduce` encapsulates the computational behavior of the `sequent_rank`, `list_rank`, and `formula_rank` functions, and appropriately rewrites the instances of `append` with respect to the `list_rank` operator. Application of `SequentRankReduce` and then the Sup-Inf auto tactic `SIAuto` discharges this proof obligation. In every inductive case a subgoal corresponding to this one is generated and discharged in the same way.

The second subgoal generated by the instantiation of the induction hypothesis is the following Nuprl goal.

```
 9. L: Sequent list
 10. ∀s∈L.ρ s = 0
 11. ∀s∈L.|= s  ⇒ |= <M @ N,x::concl>
 12. ∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ <M @ N,x::concl>
 ⊢ ∃L:Sequent list
 |    ∀s∈L.ρ s = 0
 |    ∧ (∀s∈L.|= s  ⇒ |= <M @ (⌈∼⌉x::N),concl> )
 |    ∧ (∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ <M @ (⌈∼⌉x::N),concl>)
```

The existential quantifier in the conclusion is eliminated by providing the list `L` as witness. After decomposing the conjunctions this yields three subgoals, one for each conjunct in the conclusion of the Nuprl goal.

The first conjunct is trivially discharged as it occurs as hypothesis `10`.

Decomposing the second conjunct results in a Nuprl goal requiring us to show the goal sequent is valid under the assumption that all sequents in the list `L` are valid.

```
 13. ∀s∈L.|= s
 ⊢ |= <M @ (⌈∼⌉x::N),concl>
```

Consider the following lemma characterizing the relationship of validity between the eliminant and the goal sequent.

```
*T formula_not_left_sound
   ∀concl,M,N:Formula list. ∀f:Formula.
     |= <M @ N,f::concl>  ⇒ |= <M @ (⌈∼⌉f::N),concl>
```

Backchaining through this lemma yields the following Nuprl subgoal.

```
 ⊢ |= <M @ N,x::concl>
```

This simpler goal is trivially proved by backchaining through the hypotheses. This completes the proof of the obligation induced by the second conjunct.

The proof obligation generated by the third conjunct relies on another lemma, this one justifying that any assignment falsifying the eliminant also falsifies the goal sequent.

```
*T formula_not_left_falsifiable
   ∀concl,M,N:Formula list. ∀f:Formula. ∀a:Assignment.
      a |≠ <M @ N,f::concl>⟺ a |≠ <M @ (⌈~⌉f::N),concl>
```

After decomposing the conclusion containing the third conjunct we have the following proof node.

```
 13. a: Assignment
 14. ∃s∈L.a |≠ s
 ⊢ a |≠ <M @ (⌈~⌉x::N),concl>
```

Backchaining through the lemma `formula_not_left_falsifiable` and then backchaining through the hypotheses discharge this goal completing the proof of the case where a negation occurs in the hypothesis.

The proofs of the other inductive cases, *i.e.* a non-zero rank formula occurs in either the `hyp` or the `concl` lists, proceed similarly. One (or two) sequent(s) of smaller rank that behave appropriately under the sequent validity and sequent falsifiability relations are used to instantiate the induction hypothesis. In the case only one instance of the induction hypothesis is needed because the resulting list is provided as witness to eliminate the existential quantifier in the conclusion. If two occurrences of the induction hypothesis are needed, two sequent lists are generated by the two applications of the induction hypothesis. The existential in the conclusion is eliminated by the list created by conjoining them. In every case, the second and third conjuncts of the conclusion are justified by backchaining through lemmas which characterize the elimination of an operator from the non-zero rank formula. The cases of an `or` or an `implication` in the hypothesis list `hyp` induce two hypotheses as does an occurrence of `and` in the conclusion list `concl`.

4.4.2.3 Conjunction on the left In this case a non-zero rank formula occurring on the left is a conjunction of the form `x1⌈∧⌉x2`. This yields the following Nuprl goal.

```
 8. x1: Formula
```

```
 9. x2: Formula
 10. ∀S:Sequent
       ρ S < ρ <M @ (x1⌈∧⌉x2::N),concl>
       ⇒ (∃L:Sequent list
            ∀s∈L.ρ s = 0
            ∧ (∀s∈L.|= s  ⇒ |= S )
            ∧ (∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ S))
 ⊢ ∃L:Sequent list
 |   ∀s∈L.ρ s = 0
 |   ∧ (∀s∈L.|= s  ⇒ |= <M @ (x1⌈∧⌉x2::N),concl> )
 |   ∧ (∀a:Assignment.
         ∃s∈L.a |≠ s ⇒ a |≠ <M @ (x1⌈∧⌉x2::N),concl>)
```

In this case the induction hypothesis is instantiated with the sequent <x1::x2::M @ N,concl>. Note, the conjunction x1⌈∧⌉x2 has been removed from the hypothesis list of the goal sequent and the two conjuncts x1 and x2 have been prepended to it.

The following lemma justifies the claim that validity of this simpler sequent implies the validity of the more complex one.

```
*T formula_and_left_sound
    ∀concl,M,N:Formula list. ∀q,r:Formula.
        |= <q::r::M @ N,concl>  ⇒ |= <M @ (q⌈∧⌉r::N),concl>
```

That any assignment falsifying the eliminant also falsifies the goal sequent is justified by the following lemma.

```
*T formula_and_left_falsifiable
   ∀concl,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment.
      a |≠ <q::r::M @ N,concl>⟺ a |≠ <M @ (q⌈∧⌉r::N),concl>
```

4.4.2.4 Disjunction on the left In this case a disjunction of the form x1⌈∨⌉x2 occurs on the left. Elimination of a disjunction on the left is the first case seen here requiring two instances of the induction hypothesis. After copying the induction hypothesis and making appropriate substitutions we are presented with the following Nuprl goal.

```
 8. x1: Formula
 9. x2: Formula
 10. ∀S:Sequent
```

```
       ρ S < ρ <M @ (x1⌈∨⌉x2::N),concl>
       ⇒ (∃L:Sequent list
            ∀s∈L.ρ s = 0
            ∧ (∀s∈L.|= s  ⇒ |= S )
            ∧ (∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ S))
11. ∀S:Sequent
        ρ S < ρ <M @ (x1⌈∨⌉x2::N),concl>
        ⇒ (∃L:Sequent list
             ∀s∈L.ρ s = 0
             ∧ (∀s∈L.|= s  ⇒ |= S )
             ∧ (∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ S))
⊢ ∃L:Sequent list
|   ∀s∈L.ρ s = 0
|   ∧ (∀s∈L.|= s  ⇒ |= <M @ (x1⌈∨⌉x2::N),concl> )
|   ∧ (∀a:Assignment.
|       ∃s∈L.a |≠ s ⇒ a |≠ <M @ (x1⌈∨⌉x2::N),concl>)
```

The first induction hypothesis is instantiated with the sequent `<x1::M @ N,concl>` and the second induction hypothesis is instantiated with the sequent `<x2::M @ N,concl>`. The first instantiation results in the declaration of a list `L` and the second in `L1`. Discharging the existential quantifier in the conclusion with the list `(L @ L1)` (the append of `L` and `L1`) results in the following Nuprl goal.

```
10. L: Sequent list
11. ∀s∈L.ρ s = 0
12. ∀s∈L.|= s  ⇒ |= <x1::M @ N,concl>
13. ∀a:Assignment. ∃s∈L.a |≠ s ⇒ a |≠ <x1::M @ N,concl>
14. L1: Sequent list
15. ∀s∈L1.ρ s = 0
16. ∀s∈L1.|= s  ⇒ |= <x2::M @ N,concl>
17. ∀a:Assignment. ∃s∈L1.a |≠ s ⇒ a |≠ <x2::M @ N,concl>
⊢ (∀s∈L.ρ s = 0 ∧ ∀s∈L1.ρ s = 0)
|  ∧ (∀s∈L.|= s  ∧ ∀s∈L1.|= s  ⇒ |= <M @ (x1⌈∨⌉x2::N),concl> )
|  ∧ (∀a:Assignment
|      ∃s∈(L @ L1).a |≠ s ⇒ a |≠ <M @ (x1⌈∨⌉x2::N),concl>)
```

To complete the proof of this case we must prove the three conjuncts of the conclusion.

The first conjunct, that `L` and `L1` only contain zero rank sequents, is justified by hypotheses `11` and `15`.

The second conjunct, that the validity of all sequents in `L` and the validity of all sequents in `L1` implies the validity of the goal sequent, is justified by the following lemma.

```
*T formula_or_left_sound
   ∀concl,M,N:Formula list. ∀q,r:Formula.
     |= <q::M @ N,concl>  ⇒ |= <r::M @ N,concl>
     ⇒ |= <M @ (q⌈∨⌉r::N),concl>
```

Backchaining through this lemma and then through the hypotheses completes the proof of this branch.

Finally, the third conjunct, specifying that the falsifiability of any sequent in the list `(L @ L1)` implies the falsifiability of the goal, is justified by the following lemma.

```
*T formula_or_left_falsifiable
   ∀concl,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment.
       a |≠ <q::M @ N,concl> ∨ a |≠ <r::M @ N,concl>
        ⟺ a |≠ <M @ (q⌈∨⌉r::N),concl>
```

As above, backchaining through the lemma, and then through the hypotheses, completes the proof of this case.

4.4.2.5 Implication on the left Like disjunction, the occurrence of an implication on the left requires two instances of the induction hypothesis. One is instantiated with the sequent `<r::M @ N,concl>` and the other with the sequent `<M @ N,q::concl>`. To see why these sequents are the correct eliminants for an implication on the left, consider the cases already given for disjunction on the the left and negation on the left and then recall the equivalence of a classical implication `(q⌈⇒⌉r)` with the disjunction `(⌈¬⌉q⌈∨⌉r)`.

After instantiating the two induction hypotheses with these eliminants we are left with list `L` and `L1` having the properties specified by the induction hypothesis. Discharging the existential quantifier in the conclusion of the proof node with the list `(L @ L1)` we are left to prove the familiar three part conjunct. The proof closely reflects that given above for the case of a disjunction on the left. The first conjunct is trivially discharged because of the properties of the lists `L` and `L1`. The lemmas supporting the proofs of the second and third conjuncts are as follows.

```
*T formula_imp_left_sound
   ∀concl,M,N:Formula list. ∀q,r:Formula.
     |= <M @ N,q::concl>  ⇒ |= <r::M @ N,concl>
     ⇒ |= <M @ (q⌈⇒⌉r::N),concl>
*T formula_imp_left_falsifiable
   ∀concl,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment.
      a |≠ <r::M @ N,concl> ∨ a |≠ <M @ N,q::concl>
       ⟺ a |≠ <M @ (q⌈⇒⌉r::N),concl>
```

4.4.3 The conclusion contains non-atomic formula

Should the hypothesis list `hyp` be atomic and the conclusion list `concl` contain a non-atomic formula the proofs are nearly symmetrical with those presented above, and we do not give them here. The falsifiability and soundness lemmas supporting the cases are listed below without further comment.

4.4.3.1 A variable on the right The case of a variable formula occurring on the right and assumed to have non-zero rank is contradictory and this case is discharged as it was above.

4.4.3.2 Negation on the right

```
*T formula_not_right_sound
 ∀hyp,M,N:Formula list. ∀f:Formula.
   |= <f::hyp,M @ N>  ⇒ |= <hyp,M @ (⌈~⌉f::N)>
*T formula_not_right_falsifiable
   ∀hyp,M,N:Formula list. ∀f:Formula. ∀a:Assignment.
      a |≠ <f::hyp,M @ N>⟺ a |≠ <hyp,M @ (⌈~⌉f::N)>
```

4.4.3.3 Conjunction on the right

```
*T formula_and_right_sound
   ∀hyp,M,N:Formula list. ∀q,r:Formula.
     |= <hyp,q::M @ N>  ⇒ |= <hyp,r::M @ N>
    ⇒ |= <hyp,M @ (q⌈∧⌉r::N)>
*T formula_and_right_falsifiable
   ∀hyp,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment.
      a |≠ <hyp,q::M @ N> ∨ a |≠ <hyp,r::M @ N>⇐
      ⇒ a |≠ <hyp,M @ (q⌈∧⌉r::N)>
```

4.4.3.4 Disjunction on the right

```
*T formula_or_right_sound
   ∀hyp,M,N:Formula list. ∀q,r:Formula.
     |= <hyp,q::r::M @ N>  ⇒ |= <hyp,M @ (q⌈∨⌉r::N)>
*T formula_or_right_falsifiable
   ∀hyp,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment.
     a |≠ <hyp,q::r::M @ N>⟺ a |≠ <hyp,M @ (q⌈∨⌉r::N)>
```

4.4.3.5 Implication on the right

```
*T formula_imp_right_sound
   ∀hyp,M,N:Formula list. ∀q,r:Formula.
     |= <q::hyp,r::M @ N>  ⇒ |= <hyp,M @ (q⌈⇒⌉r::N)>
*T formula_imp_right_falsifiable
   ∀hyp,M,N:Formula list. ∀q,r:Formula. ∀a:Assignment.
      a |≠ <q::hyp,r::M @ N>⟺ a |≠ <hyp,M @ (q⌈⇒⌉r::N)>
```

This completes the proof of the normalization lemma.

4.5 Remarks on Normalization and Sequent Proof Rules

Those familiar with sequent presentations of classical propositional logic may have noticed the correspondence between the inductive proof steps of the normalization lemma and the ordinary sequent proof rules.

A sequent proof rule is a figure having one of three forms:

$$\frac{}{S} \quad \text{or} \quad \frac{S_1}{S} \quad \text{or} \quad \frac{S_1 \quad S_2}{S}$$

where S, S_1 and S_2 are sequents, S_1 and S_2 being the hypotheses of the rule and S being the conclusion. The rule having no hypotheses is an axiom.

We construct a sequent proof system from the proof of decidability by adding one rule for each inductive step in the proof of the normalization lemma and adding one axiom which we will relate to the proof of the lemma `zero_rank_valid_or_falsifiable`. For each inductive step in the normalization proof the goal sequent is made the conclusion of a new rule and the sequent(s) used to instantiate the inductive hypothesis are the hypotheses. Thus, the rules can be viewed as backward elimination steps, each rule specifying how to eliminant one operator occurring either on the left or the right side of a sequent. Following this prescription, the proof of the normalization lemma yields the following sequent proof system.

$$\frac{<M@N,\, p :: concl>}{<M@(\lceil\sim\rceil p :: N),\, concl>} \qquad \frac{<p :: hyp,\, M@N>}{<hyp,\, M@(\lceil\sim\rceil p :: N)>}$$

$$\frac{<q :: r :: M@N,\, concl>}{<M@(q\lceil\wedge\rceil r :: N),\, concl>} \qquad \frac{<hyp,\, q :: M@N> \quad <hyp,\, r :: M@N>}{<hyp,\, M@(q\lceil\wedge\rceil r :: N)>}$$

$$\frac{<q :: M@N,\, concl> \quad <r :: M@N,\, concl>}{<M@(q\lceil\vee\rceil r :: N),\, concl>} \qquad \frac{<hyp,\, q :: r :: M@N>}{<hyp,\, M@(q\lceil\vee\rceil r :: N)>}$$

$$\frac{<M@N,\, q :: concl> \quad <r :: M@N,\, concl>}{<M@(q\lceil\Rightarrow\rceil r :: N),\, concl>} \qquad \frac{<q :: hyp,\, r :: M@N>}{<hyp,\, M@(q\lceil\Rightarrow\rceil r :: N)>}$$

Examination of the proof of the lemma `zero_rank_valid_or_falsifiable` reveals that atomic sequents are valid whenever its hypothesis list and conclusion lists share a formula. This yields the following axiom to complete the proof system.

$$\frac{}{<M\,@\,q :: N,\, M'\,@\,q :: N'>}$$

In the proof we've presented here, application of the axiom rule (in the form of the lemma `zero_rank_valid_or_falsifiable`) is further restricted to the case when all formulas in the hypothesis and conclusion lists are atomic. This restriction is not required for soundness.

References

[1] William Aitken, Robert Constable, and Judith Underwood. Metalogical frameworks II: Using reflected decision procedures. unpublished manuscript.

[2] Henk P. Barendregt. The lambda calculus: its syntax and symantics. In *Studies in Logic*, volume 103. Amsterdam:North-Holland, 1981.

[3] Henk P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[4] D. Basin and R. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, chapter 1, pages 1–29. Cambridge University Press, Great Britain, 1993.

[5] R.S. Boyer and J.S. Moore. *A Computational Logic.* NY:Academic Press, 1979.

[6] R. Constable and D. Howe. Implementing metamathematics as an approach to automatic theorem proving. In R.B. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Source Book.* Elsevier Science Publishers (North-Holland), 1990.

[7] Robert L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[8] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[9] M. Davis and J. Schwartz. Metamathematical extensibility for theorm verifies and proof checkers. Technical Report 12, Courant Institute of Mathematical Sciences, New York, 1977.

[10] Michael Dummett. *Elements of Intuitionism.* Clarendon Press, Oxford, 1977.

[11] Peter. Dybjer. Comparing integrated and external logics of functional programs. *Science of Computer Programming*, pages 14:207–14., 1990.

[12] Jean-Yevs Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[13] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF: a mechanized logic of computation. *Lecture Notes in Computer Science*, 78, 1979.

[14] Paul B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, Ithaca, NY, 1995. TR95-1509.

[15] Stephen C. Kleene. *Introduction to Metamathematics.* van Nostrand, Princeton, 1952.

[16] J. Leszczylowski. An experiment with Edinburgh LCF. In W. Bibel and R. Kowalski, editors, *5th International Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 170–181, New York, 1981. Springer-Verlag.

[17] Per. Martin–Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73.*, pages 73–118. Amsterdam:North-Holland, 1973.

[18] Per. Martin–Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. Amsterdam:North Holland, 1982.

[19] Elliott Mendelson. *Introduction to Mathematical Logic*. D. Van Nostrand, second edition, 1979.

[20] Bengt Nordstrom. Programming in constructive set theory: Some examples. In *Proceedings 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 290–341. Portsmouth, England, 1981.

[21] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system coq. *Journal of Symbolic Computation*, 15(5-6):607–640, 1993.

[22] Lawrence Paulson. Proving termination of normalization functions for conditional expressions. *Journal of Automated Reasoning*, pages 2:63–74, 1986.

[23] Raymond M. Smullyan. *First–Order Logic*. Springer–Verlag, New York, 1968.

[24] Judith Underwood. Tableau for intuitionistic predicate logic as metatheory. In Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors, *Theorem Proving with Analytic Tableaux and Related Methods*, number 918 in Lecture Notes in Artificial Intelligence. Springer, 1995.