Classical Propositional Decidability via Nuprl Proof Extraction

James L. Caldwell*

Department of Computer Science
Cornell University
Ithaca, N.Y.
caldwell@cs.cornell.edu

Abstract. This paper highlights a methodology of Nuprl proof that results in efficient programs that are more readable than those produced by other established methods for extracting programs from proofs. We describe a formal constructive proof of the decidability of a sequent calculus for classical propositional logic. The proof is implemented in the Nuprl system and the resulting proof object yields a "correct-by-construction" program for deciding propositional sequents. If the sequent is valid, the program reports that fact; otherwise, the program returns a counter-example in the form of a falsifying assignment. We employ Kleene's strong three-valued logic to give more informative counter-examples, it is also shown how this semantics agrees with the standard two-valued presentation.

1 Introduction

Nuprl is both a constructive type theory and an implementation of the type theory in the form of a proof development system. As a result of the constructivity, and by design, Nuprl proofs yield programs in the form of terms of an untyped lambda calculus.

This paper presents a Nuprl proof of decidability for a classical propositional logic along with the resulting programs. Nuprl is used here as a formal metatheory for a deep embedding of the syntax and semantics of the logic in Nuprl. Decidability for this embedded formal system is proved within the Nuprl system and the program extracted from the proof is a "correct-by-construction" propositional decider.

The idea of verifying of decision procedures is not a new one; proposals to extend theorem provers by adding formally verified decision procedures were made as as early as 1977 [7]. Harrison provides a detailed survey of two approaches to the disciplined extension of prover capabilities in [9]. 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

* Part of this work was performed while the author was a member of the Formal Methods Group at NASA Langley Research Center in Hampton VA.

of an `IF-THEN-ELSE` normalization procedure [2, 14, 16, 12, 15]. Both Shankar [17] and Hayashi [11] verify deciders for implicational fragments of propositional logic presented in sequent forms. Paulin-Mohring and Werner's work [15] is the closest in spirit to the work presented here in that they extract the program for the Boyer and Moore tautology checker from a constructive proof. In their development they address issues related to the efficiency of the extracted program.

1.1 Overview of the Approach

The development presented in this paper is based on the informal account given by Constable and Howe in [5]. The program extracted from the formal proof corresponds to the algorithm which searches for a sequent calculus proof via repeated (backward) application of the sequent rules until all propositional operators have been eliminated. The leaves of the resulting derivation tree form a collection of atomic sequents (sequents composed strictly of variables) which are easily checked for validity by determining if they are axioms. If they are all axioms, then the derivation tree is a proof and that fact is reported. If there is a leaf that is not axiomatic, it is used to construct a falsifying assignment which serves as a counter-example to the original goal. The core of the algorithm is the recursive procedure extracted from a normalization lemma proved via a well-founded (inverse image) induction on the rank of a sequent. This procedure collects the leaves of the derivation tree implicit in its recursion, *i.e.* the tree is not explicitly constructed but is implicit in the recursion.

The presentation given here is unique in that the semantics are defined via Kleene's strong three valued logic which is the natural partial evaluation semantics for classical propositional logic. Under a "fullness" condition defined for three-valued assignments, three-valued validity coincides with the standard Boolean semantics. As developed here, a formula is valid under the Kleene semantics when every assignment that contains enough information (assigns values to enough variables) to determine truth or falsity of the formula asserts it's truth. This notion of validity is lifted to sequents in the natural way. The Kleene semantics account for partial assignments in a particularly clean way and allow for tighter counter-examples by allowing "don't care" conditions in assignments.

The proof presented here is a version of the one presented by the author in [3] that has been optimized to produce more efficient and readable computational content. The Nuprl proofs for the earlier development are available on the web at the site noted in reference [3].

2 An Overview of the Nuprl System

The Nuprl type theory is a sequent presentation of a constructive type theory via type assignment rules. The underlying programming language is untyped and the objective of a proof is to either prove a type is inhabited, *i.e.* to show that some term (program) is a member of the type, or to show that a term inhabits a particular type. A complete presentation of the type theory can be found in the Nuprl book [6].

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]

2.1 The computation system

Nuprl's *terms* include the constructs of its untyped functional programming language with additional constructs for denoting types and propositions. Terms are printed here in `typewriter` font. The Nuprl computation system provides reduction rules for a left-most outermost (lazy) evaluation strategy.

For terms `t` and `t'` we will write `t` $\triangleright$ `t'` to indicate that `t` evaluates to `t'` under the reduction rules. The computation system can be extended via the rewrite facility. For terms `t` and `t'` we will write `t`$\triangleright_R$`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, $\mathtt{t[t_1,\cdots,t_n/x_1,\cdots,x_n]}$ denotes the simultaneous substitution of each $\mathtt{t}_i$ for each $\mathtt{x}_i$ in `t`. We will sometimes write $\bar{t}$ to denote a vector of terms or variables.

2.2 The type theory

A *Nuprl type* is a term `T` of the computation system together with a transitive and symmetric relation denoted by $\mathtt{x=y}\in\mathtt{T}$. This relation is known as *equality on T*. The term $\mathtt{x}\in\mathtt{T}$, meaning `x` is a member of `T`, is an abbreviation of $\mathtt{x=x}\in\mathtt{T}$. Equality on `T` is an equivalence relation when restricted to members of `T`, it is nonsense otherwise. Interpreting the type membership equality relation and type membership as types is made sensible via the propositions-as-types interpretation [6, pg.29–31].

In addition to the type membership equality provided with each type, there is an equality between 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 $\{\mathtt{x{:}T \mid True}\}$ have the same members and equality relations, they are not equal types in Nuprl.

Nuprl's type theory is predicative, supporting an unbounded cumulative hierarchy of type universes. Every universe is itself a type and every type is an element of some universe.

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

$\mathbb{U}\{\mathtt{i}\}$ denotes the type *universe* where i is a polymorphic specification of universe level. The members of the universe $\mathbb{U}\{\mathtt{i}\}$ are types and other universes $\mathbb{U}\{\mathtt{j}\}$ for j<i. When the level is ''i'', $\mathbb{U}\{\mathtt{i}\}$ is displayed simply as $\mathbb{U}$. The statement that T is a type is formally written T$\in\mathbb{U}$.

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

Nuprl includes the following types:

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(x) is an element of all types T, *i.e.* any(x)$\in$T.

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

Atom is the type whose elements are denoted by *strings* of the form ''$\cdots$'' where $\cdots$ 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$\in$T and t$\in$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$\rightarrow$B[y] is the *dependent function* type containing functions with domain of type A and where B[y] is a term and y is a variable possibly occurring free in B. When a$\in$A, B[a/y] is a type, and M[a/x]$\in$B[a/y], a lambda abstraction of the form λx.M is an element of the type y:A$\rightarrow$B[y]. These are the functions whose range may depend on the element of the domain applied to. Function equality is extensional.

A$\rightarrow$B is the *function* type which is an abbreviation for the term y:A$\rightarrow$B when y does not occur free in B.

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

A$\times$B is the *product* type and is an abbreviation for the term x:A$\times$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$\in$A and inr(b) for b$\in$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[x]) is the Nuprl *inductive type* constructor where x is a variable and T[x] is a term possibly containing a x free. 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$\in$T[rec(x.T)/x].

{y$\in$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 those elements in the set type.

$\cap$x:T.P[x] denotes the *intersection* type. It is a type whenever T is a type and P[z/x] can be shown to be a type under the condition that z is a hidden variable of type T. Two members a and b are equal in type $\cap$x:T.P[x] if T is a type and a=b$\in$P[z/x] for z an arbitrary element of T.

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.

2.3 Logic via propositions-as-types

A constructive logic is encoded within the Nuprl type theory. The following definitions in the Nuprl V4 core_1 system library encode the logic.

$\mathtt{True} \overset{\text{def}}{=} 0 \in \mathbb{Z}$	$\mathtt{False} \overset{\text{def}}{=} \mathtt{Void}$
$\mathtt{P} \wedge \mathtt{Q} \overset{\text{def}}{=} \mathtt{P} \times \mathtt{Q}$	$\mathtt{P} \vee \mathtt{Q} \overset{\text{def}}{=} \mathtt{P} \mid \mathtt{Q}$
$\mathtt{P} \Rightarrow \mathtt{Q} \overset{\text{def}}{=} \mathtt{P} \rightarrow \mathtt{Q}$	$\neg\mathtt{A} \overset{\text{def}}{=} \mathtt{A} \Rightarrow \mathtt{False}$
$\exists\mathtt{x{:}A.\ B[x]} \overset{\text{def}}{=} \mathtt{x{:}A} \times \mathtt{B[x]}$	$\forall\mathtt{x{:}A.\ B[x]} \overset{\text{def}}{=} \mathtt{x{:}A} \rightarrow \mathtt{B[x]}$

The Nuprl tactics have been built to manipulate both propositions and types uniformly.

2.4 Judgements

Nuprl judgements are the assertions one proves in the system. Nuprl judgements take the following form:

$$\mathtt{x_1{:}T_1,\cdots,x_n{:}T_n \;>>\; S\ [ext\ s]}$$

where $\mathtt{x_1,\cdots,x_n}$ are distinct variables and $\mathtt{T_1,\cdots,T_n}$, S, and s are terms (n may be 0), every free variable of $\mathtt{T}_i$ is one of $\mathtt{x_1,\cdots,x}_{i-1}$ and every free variable of S or of s is one of $\mathtt{x_1,\cdots,x}_n$. The list $\mathtt{x_1{:}T_1,\cdots,x_n{:}T_n}$ 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 is the *consequent* or *conclusion*, the term following the keyword ext is the *extract*, and the entire form is a Nuprl *sequent*. The extract component of judgements are not displayed as part of the implementation of the proof editor. A judgement of the form

$$\mathtt{x_1{:}T_1,\cdots,x_n{:}T_n \;>>\; s} \in \mathtt{S}$$

is called a *well-formedness goal*. Since s$\in$S is simply shorthand for s=s$\in$S by the propositions-as-types interpretation for type equality, the extract of a well-formedness goal is the constant Axiom.

Somewhat informally, a judgement asserts that, assuming the hypotheses are well-formed types, then the term S is an inhabited type and the extract s is an inhabitant [6, pg.141]. 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.

A Nuprl proof is a decorated tree of sequents, its root being the main goal of the proof and where the children of each node are sequents justifying the parent according to the rules of the type theory. A proof of a sequent shows that its main goal is both well-formed and inhabited. Given terms inhabiting the hypotheses of a rule, a proof specifies how to construct a term inhabiting the type in the conclusion of the rule; thus, proofs contain instructions for the construction of witness terms. *Extraction* is the process of constructing a witness term as specified by proof.

2.5 The Nuprl system

The Nuprl system supports construction of proofs by top-down refinement. The prover is implemented as a tactic based prover in the style of HOL [8]. Nuprl differs from HOL in that each tactic invocation defines more of the structure of an explicitly represented proof tree which is directly manipulated in the editor, stored in the Nuprl library, and retrieved for later editing. The tactic language is ML. In Nuprl the proposition-as-types interpretation allows for presentations to be cloaked in either logical or more purely type-theoretic terms.

The Nuprl system supports a powerful display mechanism. Nuprl terms are edited using a structure editor; however, the structure of a term is independent of its display. The display form is specified by the user and can be changed without changing the structure of the term. Thus, the displayed form of a Nuprl term is never parsed, the editor displays the terms to the user as specified, but manipulates the actual underlying structure. All Nuprl terms occurring in this paper appear on the page as they do in the Nuprl editor and library. In [1] Allen gives an example of a non-trivial application of the display mechanism.

2.6 Decidability, Stability, the Squash Type, and Squash Stability

Being constructive, Nuprl does not assume all propositions are decidable, *i.e.* in general the so-called law of excluded middle is not provable; that is, $\forall$P:$\mathbb{P}$.P$\vee\neg$P is not a theorem of Nuprl. Even though decidability for an arbitrary proposition P is not assumed, 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 decidability abstraction `Dec{P}`.

*ABS decidable $\quad$ Dec{P} $\stackrel{\text{def}}{=}$ P $\vee\neg$P
*THM decidable_wf $\quad$ $\forall$P:$\mathbb{P}${i}. (Dec{P} $\in$ $\mathbb{P}${i})

Note that the well-formedness theorem `decidable_wf` asserts the fact that the term `Dec{P}` is a type for all propositions `P`, but it does not prove it is inhabited for arbitrary propositions `P`.

A related notion is that of *stability* which is constructively weaker than, but classically equivalent to, decidability (*i.e.* they're both tautologies). Stability is also not constructively valid.

*ABS stable $\quad$ Stable{P} $\stackrel{\text{def}}{=}$ $\neg\neg$P$\Rightarrow$P
*THM stable_wf $\quad$ $\forall$P:$\mathbb{P}${i}. (Stable{P}$\in\mathbb{P}${i})

A `squashed type` (or proposition) is one whose computational content has been discarded. The squash operator is defined in Nuprl by a set type as follows:

```
*ABS squash      ↓(T) ≝ {True| T}
```

Thus for any type (proposition) `T`, `↓(T)` is inhabited if and only if `T` is, and furthermore, has as its only inhabitant the term `Axiom` (the sole inhabitant of the proposition `True`.) The operator is called `squash` because it identifies all inhabitants of `T` with the single constant `Axiom`.

If we can reconstruct an inhabitant of a type `P` simply from knowing `↓(P)` is inhabited we say `P` is *squash stable.*

```
*ABS sq_stable     SqStable{P} ≝ ↓(P)  → P
*THM sq_stable_wf  ∀P:ℙ{i}. (SqStable{P} ∈ ℙ{i})
```

Squash stability is weaker even that stability and is related to stability in that they are equivalent for decidable propositions.

2.7 Existential VS. Set Type

A method of generating efficient and readable extracts by the use of the set type (as opposed to the existential) was presented by the author in [4]. Earlier work by Hayashi [10] stressed a similar approach. We reiterate the main points here.

Inhabitants of the existential `∃x:T.P[x]` are pairs `<a,b>` where `a∈T` and `b∈P[a/x]`. The term `b` inhabiting `P[a/x]` specifies, as far as the proofs-as-programs interpretation goes, how to prove `P[a/x]`. When an existential type of the form above occurs as a hypothesis it can be decomposed into two hypotheses, one of the form `a:T` and another asserting `b:P[a/x]`. If `v` is the name of the variable denoting the existential hypothesis, occurrences of `a` in the final extract will appear as π_1`(v)`, and occurrences of `b` appear as π_2`(v)`.

Alternatively, consider the Nuprl set type `{y∈T|P[y]}`. Its inhabitants are elements of `T`, say `a`, such that `P[a/y]` holds. Thus, a set type does not carry the computational content associated with the logical part `P[a/y]`. Since the computational content is not available, the fact that the `a` has the property `P[a/x]` is not freely available in parts of a proof where it might find its way into an extract. When a set type of this form, occurring as a hypothesis, is decomposed it results in two new hypotheses: one of the form `a:T`; and the other, a "hidden" hypothesis, of the form `b:P[a/x]`. Recall that every hypothesis declares a variable. The proof rules prevent the variable of a hidden hypothesis from appearing free in the extract of a proof.

Nuprl system manages hidden hypotheses by "unhiding" them when appropriate and by preventing their inadvertent use. Hidden hypotheses become unhidden and are freely available in the parts of a proof where they do not contribute to computational content; these parts include proofs of well-formedness (membership) subgoals, equality subgoals, when the computational content on a branch of the proof has already been fully determined, or when the conclusion is decidable, stable, or squash stable. Hidden hypotheses may be "unhidden" when their computational content can be effectively decided; typically when they themselves can be shown to be decidable, stable, or squash stable.

3 Syntax and Semantics of Formulas and Sequents

In this section the Nuprl definitions supporting the statement and proof of the decidability theorem are presented.

3.1 Formulas

In the Nuprl formalization, formulas are modeled by a recursive type.

```
*ABS 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 type `Var` of propositional variables. These form the basis 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. These elements of the disjoint union will denote negations and will be displayed as (⌈∼⌉x). The third, fourth, and fifth elements of the disjoint union are the products of two recursively smaller formulas. When the semantics of propositional formulas is defined below it becomes clear that the pairs of formula in the third, fourth, and fifth disjuncts denote the operators for conjunction `(p⌈∧⌉q)`, disjunction `(p⌈∨⌉q)`, and implication `(p⌈⇒⌉q)`.

A formula of the form ⌈`x`⌉, where `x` denotes an element of type `Var`, will be called an *atomic formula.*

The destructor for the `Formula` type is given by a `formula_case` operator defined by nested case analysis on the disjoint union type. A measure on formulas is defined as the number of operators occurring in it. It is defined recursively as follows.

```
*ABS 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;
```

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

3.2 Three valued Semantics of propositional logic

We define a semantics of classical propositional logic in terms of Kleene's strong three-valued logic [13]. 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). For example, if either `p` or `q` is *false* under the Kleene valuation induced by a partial assignment

a, then p$\wedge_K$q is *false* under the valuation too. It does not matter what value the other conjunct has, or even if it is defined. Clearly, exhibiting a partial assignment that falsifies a formula is gives more information than a falsifying total assignment does.

$\mathbb{N}_3$ is the three valued type containing elements displayed as 0_3, 1_3, and 2_3 denoting *False*, *undefined*, and *True* respectively. The operators of Kleene's three valued logic [13] are defined over $\mathbb{N}_3$ as follows.

	$\sim_K$
0	2
1	1
2	0

$\wedge_K$	0	1	2
0	0	0	0
1	0	1	1
2	0	1	2

$\vee_K$	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

$\Rightarrow_K$	0	1	2
0	2	2	2
1	1	1	2
2	0	1	2

Inspection of their matrices reveals that on inputs restricted to 0_3 and 2_3 the operators behave exactly as the familiar boolean operators of the same names. Thus, these operators are uniquely determined as the strongest possible regular extensions of the classical 2-valued operators. These operators are formalized in Nuprl using `case` analysis over $\mathbb{N}_3$.

Three valued assignments are functions of type Var$\rightarrow\mathbb{N}_3$. The Kleene valuation of a formula F under the partial assignment a (displayed as (F under a)) is defined as follows.

```
*ABS valuation
(F under a)  def= (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
```

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

```
*ABS formula_sat            a |= F  def= ((F under a) = 2₃) ∈ ℕ₃
*ABS formula_falsifiable  a |≠ F  def= ((F under a) = 0₃) ∈ ℕ₃
```

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 $|\neq$ F when (F under a) evaluates to 0_3.

The satisfaction of a formula by an assignment is clearly a decidable property; to decide if a formula is satisfied by a, evaluate (F under a) and check whether the result is equal to 2_3. Falsification is similar. This property is captured by the following theorems.

```
*THM decidable_formula_sat:
   ∀a:Assignment. ∀F:Formula. Dec{a |= F}
*THM decidable_formula_falsifiable:
   ∀a:Assignment. ∀F:Formula. Dec{a |≠ F}
```

3.3 Sequents

Sequents are formalized as pairs of lists of formulas:

```
*ABS Sequent:  Sequent ≝ Formula list × Formula list
```

We define a measure function on sequents ($\boldsymbol{\rho}$) as the sum of the ranks of their hypothesis and conclusion lists. Note that we have not distinguished the display form for rank of a formulas from the display form for rank of a sequent. Their terms are distinguished in the system, but we have chosen to display them in the same way.

We call sequents having rank 0 *atomic* sequents. They contain only variables.

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 is true 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, $\langle [H_1, \ldots, H_n], [\,] \rangle$ evidently 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.

We are interested in the notion of satisfaction under a Kleene valuation induced by a partial assignment. A convenient definition is based on the observation that a sequent is satisfied by a partial assignment either, when it falsifies some hypothesis, or when there is some formula in the conclusion that it satisfies. This suggests the following definition.

```
*ABS sequent_satisfiable
a |= <hyp,concl> ≝ ∃F∈hyp. a |≠ F ∨ ∃F∈concl. a |= F
```

Similarly, a sequent is falsified by an assignment if it satisfies every hypotheses and falsifies every conclusion.

```
*ABS sequent_falsifiable
a |≠ <hyp,concl> ≝ ∀F∈hyp. a |= F  ∧ ∀F∈concl. a |≠ F
```

These definitions exhibit the first use here of list quantification. The term `∃x∈L. P[x]` is inhabited (true) if, for some member `x` of the list `L`, the predicate `P[x]` is non-void. Thus, for empty lists it is false. Similarly, the term `∀x∈L. P[x]` is true if every `x` in `L` satisfies `P[x]`. For the empty list, the quantifier is vacuously true.

Note that 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 formalized in two decidability lemmas.

A *full assignment for a formula* `F` is a partial assignment that either satisfies or falsifies `F`, *i.e.* it contains enough information to determine a value for `F`.

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

Validity is defined with respect to full assignments.

$*$`ABS sequent_valid` $\quad$ `|= S` $\stackrel{\text{def}}{=}$ $\forall$`a:Full(S). a |= S`

The author has shown elsewhere [3] that partial assignments are monotone with respect to satisfaction and falsification as defined here, thereby showing that the definition of validity just given agrees with the standard notion of validity over total Boolean assignments.

4 Decidability

The most natural formalization of the decidability theorem would simply say a sequent is either valid or not. A logically equivalent (and computationally stronger) form of falsifiablity gives the following theorem.

$\forall$`S:Sequent. |= S` $\vee$ $\exists$`a:Assignment. a |`$\not=$ `S`

A constructive proof of this theorem [3] results in a function accepting a sequent `S` as its argument and returning one of `inl(t)` or `inr(`$\langle$`a,e`$\rangle$`)`. We are interested here in the computational content of the theorem. The term `t` under the injection `inl` has no computational interest, and so we squash it. The first element of the pair $\langle$`a,e`$\rangle$ under the `inr` injection is the counter-example, but the second element of the pair, the witness for the falsifiablity of the sequent is not interesting. Thus, we modify the existential to be set type. This gives the final statement of the theorem proved here.

$*$`THM propositional_decidability`
$\forall$`S:Sequent.` $\downarrow$`(|= S)` $\vee$ $\{$`a:Assignment | a |`$\not=$ `S`$\}$

4.1 A strategy for the proof

Consider the following propositional sequent proof system.

$$\frac{}{M,q,N \vdash M',q,N'}$$

$$\frac{M,\, N \vdash p,\, concl}{M,\, \lceil\sim\rceil p,\, N \vdash concl} \qquad \frac{p,\, hyp \vdash M,\, N}{hyp \vdash M,\, \lceil\sim\rceil p,\, N}$$

$$\frac{q,\, r,\, M,\, N \vdash concl}{M,\, q\lceil\wedge\rceil r,\, N \vdash concl} \qquad \frac{hyp \vdash q,\, M,\, N \quad hyp \vdash r,\, M,\, N}{hyp \vdash M,\, q\lceil\wedge\rceil r,\, N}$$

$$\frac{q,\, M,\, N \vdash concl \quad r,\, M,\, N \vdash concl}{M,\, q\lceil\vee\rceil r,\, N \vdash concl} \qquad \frac{hyp \vdash q,\, r,\, M,\, N}{hyp \vdash M,\, q\lceil\vee\rceil r,\, N}$$

$$\frac{M,\, N \vdash q,\, concl \quad r,\, M,\, N \vdash concl}{M,\, q\lceil\Rightarrow\rceil r,\, N \vdash concl} \qquad \frac{q,\, hyp \vdash r,\, M,\, N}{hyp \vdash M,\, q\lceil\Rightarrow\rceil r,\, N}$$

A sound rule preserves validity, *i.e.* if the validity of its hypotheses implies the validity of its conclusion. A proof rule is said to be *invertible* when every assignment satisfying the conclusion also satisfies all the hypotheses. Thus if any hypothesis of an invertible rule is falsified by a given assignment, then the conclusion is falsified by the same assignment. Each of these rules has been shown to be both sound and invertible [3].

These facts coupled with the observation that the backwards application of each rule results in one or two sequents having smaller rank suggests a recursive procedure for eliminating propositional operators, resulting in a collection of sequents having the following properties:

i.) the induced sequents are all atomic,
ii.) if all the induced sequents are valid then so is the original sequent (by soundness), and
iii.) if any of the induced sequents is falsified by an assignment then that assignment falsifies the original sequent too (by invertibility).

This 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))}
```

It should be remarked here that the propositional proof rules given are the ordinary rules. The reader might suspect that since we are using Kleene semantics the logic is somehow special, but the Kleene semantics simply allows for the construction of tighter counter-examples. Above the layer of abstraction provided by the definitions of satisfaction, falsification, and validity, the effect of the Kleene semantics on the decidability proof and the extracted program is isolated to one point. That point occurs in the proof of the following lemma which asserts that every sequent in a collection of atomic sequents is either valid or there is an assignment falsifying it.

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

In the proof of this lemma, a decision must be made as to the values to assign to variables not occurring in an atomic sequent. Rather than arbitrarily choosing *True* or *False*, as we would have to do in a two valued semantics, using Kleene's semantics, we assign the "undefined" value 1_3.

4.2 Decidability proof

We present highlights of the Nuprl proof of decidability to show how the potentially troublesome hidden hypotheses generated by the set type are handled.

```
⊢ ∀S:Sequent. ↓(|= S) ∨ {a:Assignment| a |≠ S}
```

Decomposing the universal and instantiating the normalization lemma with `S` as the goal results in the following Nuprl sequent.

```
 1. S: Sequent
 2. L: Sequent list
[3.] ↓((∀s∈L.ρ(s) = 0) ∧
       (∀s∈L.|= s) ⇒ |= S ∧
       ∀a:Assignment. (∃s∈L.a |≠ s) ⇒ a |≠ S)
 ⊢ ↓(|= S) ∨ {a:Assignment| a |≠ S}
```

Instantiating the lemma `zero_rank_valid_or_falsifiable` with `L` leaves a disjunction asserting that either all elements of `L` are valid or some element of `L` is falsifiable. Decomposing this disjunction leaves two subgoals. In the first case we know all sequents in `L` are valid and so choose to prove the first disjunct of the conclusion. In the second case we have an assignment that falsifies some sequent in `L` and so choose to prove the second disjunct of the main goal in that case.

Consider the first case.

```
 4. ↓(∀s:Sequent. s∈L ⇒ |= s)
 ⊢ ↓(|= S)
```

Because the conclusion is squashed, the hidden hypothesis (3) can be freely unhidden. Eliminating the squash operators and then decomposing the conjuncts in 3 results in the following:

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

Backchaining through hypothesis 4 combined with the fact stated in 6 completes the proof of this branch.

Now consider the second case.

```
 4. {a:Assignment| ∃s:{s:Sequent| s∈L} . a |≠ s}
  ⊢ {a:Assignment| a |≠ S}
```

After decomposing the conjunction in hypothesis 3 (see above) and then decomposing the set type in hypothesis 4 we provide the resulting assignment as the witness for the set type in the conclusion. This yields the following subgoal.

```
 3. ∀s∈L.ρ(s) = 0
 4. (∀s∈L.|= s)  ⇒ |= S
 5. ∀a:Assignment. (∃s∈L.a |≠ s) ⇒ a |≠ S
 6. a:Assignment
 7. ∃s:{s:Sequent| s∈L} . a |≠ s
  ⊢ a |≠ S
```

The hidden hypotheses have been unhidden by the system because the computational content of the proof is completed. The remaining goal is proved by appeal to facts in hypotheses 5 and 7. This completes the proof.

The program extracted from this proof (after one step of reduction) is the following term.

```
λS.decide ext{valid_or_falsifiable}(ext{normalize}(S))
      of inl(%3) => inl(Axiom)
       | inr(%4) => inr(%4)
```

It accepts a sequent `S` as input and applies the normalization procedure to it. The result is a list of zero rank sequents which serve as input to `valid_or_falsifiable`. This returns a term of the form `inl(Ax)` or `inr(a)` where `a` is a partial assignment falsifying some element of `L` (and by extension which falsifies `S`.) A case split is made on the form of this term which is then packaged up and returned as the final result of the procedure. Thus, we see that this program is nearly the natural one to write given the procedures `ext{valid_or_falsifiable}` and `ext{normalize}`. A simple optimization results in the following simpler program which foregoes the redundant `decide`.

```
λS. ext{valid_or_falsifiable}(ext{normalize}(S))
```

4.3 The Normalization Proof

The proof of this lemma provides the core of the computational procedure. The proof is by induction on the rank of a sequent. Recall the statement of the lemma.

```
⊢ ∀G:Sequent
 {L:Sequent List|
  ↓((∀s∈L. ρ(s) = 0)
    ∧ ((∀s∈L. |= s ) ⇒ |= G )
    ∧ (∀a:Assignment. (∃s∈L. a |≠ s) ⇒ a |≠ G))}
```

The measure induction tactic is invoked with `sequent_rank` as the measure. Decomposing `G` into its component formula lists, `hyp` and `concl`, results in the following subgoal.

```
  1. hyp: Formula List
  2. concl: Formula List
  3. IH: ∀k:{k:Sequent| ρ(k) < ρ(<hyp, concl>)}
          {L:Sequent List|
           ↓((∀s∈L. ρ(s) = 0)
             ∧ ((∀s∈L. |= s ) ⇒ |= k )
             ∧ (∀a:Assignment. (∃s∈L. a |≠ s) ⇒ a |≠ k))}
⊢ {L:Sequent List|
    ↓((∀s∈L. ρ(s) = 0)
    ∧ ((∀s∈L. |= s ) ⇒ |= <hyp, concl> )
    ∧ (∀a:Assignment. (∃s∈L. a |≠ s) ⇒ a |≠ <hyp, concl>))}
```

The proof 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. Thus, to proceed with the proof we case split on whether the list `hyp` contains any non-zero rank formula. In the case where all formulas in `hyp` are atomic we do a case split on whether `concl` is atomic or not. Thus, in all, we have three cases, we consider this last case first.

The sequent is atomic: In this case the list `<hyp,concl>::[]` witnesses the set type. A step of reduction leaves the following squashed conjunction to prove.

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

By 4 and 5 the first conjunct holds and the remaining two conjuncts are trivial.

The hypothesis contains non-atomic formula: Now we consider the case where the formula list `hyp` contains a non-zero rank formula, `∃f∈hyp.(ρ(f) > 0)`.

Whenever property (P) is asserted to hold for some element of a list L, we use the following lemma to decompose the list, explicitly naming an element of the list having the property.

```
* THM list_exists_decomposition
∀T:𝕌. ∀P:T → ℙ. ∀L:T List.
  (∃x∈L.P[x]) ⇒ ∃M:T List.∃x:T.{N:T List| L = M @ (x::N) ∧ P[x]}
```

Forward chaining through this lemma with hypothesis `∃f∈hyp.(ρ(f) > 0)` yields

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

We provide the following term as a witness for the set type in the conclusion.

```
case f:
 ⌜x⌝ → [];
 ⌜¬⌝x → (IH(<M @ N, x::concl>));
 x1⌜∧⌝x2 → (IH(<x1::x2::(M @ N), concl>));
 x1⌜∨⌝x2 → (IH(<x1::(M @ N), concl>) @ IH(<x2::(M @ N), concl>));
 y1⌜⇒⌝y2 → (IH(<y2::(M @ N), concl>) @ IH(<M @ N, y1::concl>));
```

This term encodes the left rules of the sequent proof system presented above. This step results in two subgoals: the first a well-formedness goal to show that the term is in the type `Sequent List` and the second to show that term satisfies the three part conjunction defining the set. At this point the computational content for this branch of the proof is complete. The remaining proof goals serve to verify the logical part of the theorem and do not contribute to its computational content.

The conclusion contains non-atomic formula: This cased is similarly verified. After a second instantiation and decomposition of the the lemma `list_exists_decomposition` we must prove the following:

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

In this case the set type in the conclusion is eliminated by the following term.

```
case f:
 ⌈x⌉ → [];
 ⌈¬⌉x → (IH(<x::hyp, M @ N>));
 x1⌈∧⌉x2 → (IH(<hyp, x1::(M @ N)>) @ IH(<hyp, x2::(M @ N)>));
 x1⌈∨⌉x2 → (IH(<hyp, x1::x2::(M @ N)>));
 y1⌈⇒⌉y2 → (IH(<y1::hyp, y2::(M @ N)>));
```

This completes the computationally significant part of the proof of the normalization lemma. The remaining part of the proof verifies that the extracted term does indeed satisfy the specification.

A purer application of the proofs as programs method would have implicitly constructed the case statements and recursive calls to the computational content of the induction hypothesis. A purer proof, equivalent to the one presented here, having the same extract, proceeds in the two branches by decomposing the formula `f` in hypothesis 6, resulting in five subgoals in each case. One subgoal for each class of formula. The proof then proceeds by appeal to the induction hypothesis. The proof presented here is more compact.

The extracted program: The extract of the proof is shown in the Figure 1. The term in the figure is nearly, but not completely a raw extract term; it is shown after one step of computation has been performed, two definitions have been folded, and some system generated variables have been renamed for readability. The structure of the program reveals the natural structure of the recursion which reflects the structure of the inductive proof. Those familiar with programs extracted from formal proofs may be surprised at the readability and naturalness of this extract. These properties are a result of the careful use of the set and squash types in the specification

5 Conclusions

The principal aim of this paper has been to exhibit recently established methodology for generating efficient and clean programs from Nuprl proofs. This paper extends the work reported on in [4] and applies those techniques to a reasonably sized example. Propositional decidability is a well understood but non-trivial test-bed for these techniques. The formalization presented in this paper shows how the use of the Nuprl set type and squash type eliminates unnecessary and inefficient computational content from proof extracts.

6 Acknowledgements

The author would like to thank the anonymous referees for their helpful comments. Thanks are also due to Bob Constable and especially Stuart Allen for his careful reading and insightful comments on this paper.


```
λG.(letrec normalize(S) =
   let <hyp,concl> = S in
     case ∃f∈hyp.(ρ(f) > 0)
     of inl(%2) =>
          let M,f@0,N = (ext{list_exists_decomposition}
                          (Formula)(λ2f.ρ(f) > 0)(hyp)(%2)) in
           case f@0:
             ⌜x⌝ → [];
             ⌜∼⌝x → (normalize(<M @ N, x::concl>));
            x1⌜∧⌝x2 → (normalize(<x1::x2::(M @ N), concl>));
            x1⌜∨⌝x2 → (normalize(<x1::(M @ N), concl>)
                             @ normalize(<x2::(M @ N), concl>));
            y1⌜⇒⌝y2 → (normalize(<y2::(M @ N), concl>)
                             @ normalize(<M @ N, y1::concl>));
       | inr(%3) =>
           case ∃f∈concl.(ρ(f) > 0)
           of inl(%5) =>
                let M,f@0,N = (ext{list_exists_decomposition}
                                (Formula)(λ2f.ρ(f) > 0)(concl)(%5)) in
               case f@0:
                 ⌜x⌝ → [];
                 ⌜∼⌝x → (normalize(<x::hyp, M @ N>));
                x1⌜∧⌝x2 → (normalize(<hyp, x1::(M @ N)>)
                                 @ normalize(<hyp, x2::(M @ N)>));
                x1⌜∨⌝x2 → (normalize(<hyp, x1::x2::(M @ N)>));
                y1⌜⇒⌝y2 → (normalize(<y1::hyp, y2::(M @ N)>));
           | inr(%6) => <hyp, concl>::[]  )
      (G)
```

Fig. 1. Extract of the Normalization Lemma

References

1. Stuart F. Allen. From dy/dx to [ ]P: A matter of notation. In *Proceedings of User Interfaces for Theorem Provers 1998*. Eindhoven University of Technology, July 1998.
2. R.S. Boyer and J.S. Moore. *A Computational Logic*. NY:Academic Press, 1979.
3. James Caldwell. Extracting propositional decidability: A proof of propositional decidability in constructive type theory and its extract. Available at `http://simon.cs.cornell.edu/Info/People/caldwell/papers.html`, March 1997.
4. James Caldwell. Moving proofs-as-programs into practice. In *Proceedings, 12th IEEE International Conference Automated Software Engineering*. IEEE Computer Society, 1997.
5. 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.
6. Robert L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
7. M. Davis and J. Schwartz. Metamathematical extensibility for theorm verifiers and proof checkers. Technical Report 12, Courant Institute of Mathematical Sciences, New York, 1977.
8. Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.
9. John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.
10. Susumu Hayashi. Singleton, union, and intersection types for program extraction. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software TACS'91*, volume 526 of *Lecture Notes in Computer Science*, pages 701–730, Berlin, 1991. Springer Verlag.
11. Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic*. Foundations of Computing. MIT Press, Cambridge, MA, 1988.
12. M. Hedberg. Normalising the associative law: An experiment with Martin-Löf's type theory. *Formal Aspects of Computing*, 3:218–252, 1991.
13. Stephen C. Kleene. *Introduction to Metamathematics*. van Nostrand, Princeton, 1952.
14. 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.
15. C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5-6):607–640, 1993.
16. Lawrence Paulson. Proving termination of normalization functions for conditional expressions. *Journal of Automated Reasoning*, 2:63–74, 1986.
17. N Shanker. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.