Intuitionisitic Tableau Extracted

James Caldwell

Department of Computer Science
University of Wyoming
Laramie, WY
`caldwell@denali.cs.uwyo.edu`

Abstract. This paper presents a formalization of a sequent presentation of intuitionisitic propositional logic and proof of decidability. The proof is implemented in the Nuprl system and the resulting proof object yields a "correct-by-construction" program for deciding intuitionisitc propositional sequents. The extracted program turns out to be an implementation of the tableau algorithm. If the argument to the resulting decision procedure is a valid sequent, a formal proof of that fact is returned, otherwise a counter-example in the form of a Kripke Countermodel is returned. The formalization roughly follows Aitken, Constable and Underwood's presentation in [1] but a number of adjustments and corrections have been made to ensure the extracted program is clean (no non-computational junk) and efficient.

1 Introduction

Confronted with the notion of automated verification the astute skeptic correctly asks, "Who verifies the verifier?" This paper, presenting a formally developed decision procedure for a sequent presentation of intuitionistic propositional logic, addresses the skeptics question, even if only peripherally. We describe the formalization and mechanical checking, in Nuprl, of a proof that intuitionistic propositional logic is decidable. The program extracted from the formal proof is a tableau decision procedure: invoked with a sequent as its argument, it returns either a multi-succedent sequent proof or a Kripke counter-example depending on whether the formula to be decided is valid or not. With the proof of decidability as our focus, we describe the formal development of a sequent proof theory, the tableau construction, and a formal theory of Kripke counter-examples which are used here as evidence of unprovability. A principle goal of the work reported here is the extraction of a reasonably readable and efficient program from the formal proof via the "proof-as-programs" interpretation implemented in Nuprl.

1.1 Related Work

In a series of papers [19, 18, 20, 1], Underwood and her colleagues presented constructive completeness proofs for intuitionistic propositional logic having tableau decision procedures as their computational content. The work reported on here extends those efforts. Underwood worked out a type theoretic presentation of the problem and presented informal proofs, including a new termination argument for the tableau construction. The formalization and proof presented here follows the proof presented in the paper by Aitken, Constable and Underwood [1] (hereafter referred to as ACU.) A fuller account of the formalization and proof can be found in [6]. In this paper we describe the formal implementation in Nuprl and adjustments made to the formalization that result in a readable and "efficient"[1] extracted program.

The idea of verifying decision procedures is not new but actual verifications are not common. One example that has published at least five times and in a number of systems is Boyer and Moore's (classical) propositional tautology checker which takes the form of an `IF-THEN-ELSE` normalization procedure. Of those efforts, Paulin-Mohring and Werner's extraction of an ML program [14] is closest in spirit to the presentation here. Both Shankar [16] and Hayashi [12] have verified deciders for implicational fragments of classical propositional logic presented in sequent forms. Caldwell [4, 6] extracted a tableau decision procedure from a proof of the decidability of a sequent presentation of classical propositional logic.

Weich [21] formalized a proof of decidability for the implicational fragment of propositional intuitionistic logic in MINLOG. His work is also closely related to the proof presented here; indeed, his effort was also inspired by Underwood's formulation of constructive decidability. Weich's proof differs from the one reported on here in that it is based on a contraction-free calculus. He reports [22] that the extracted program is huge (about 60KB) and efforts are underway to minimize its size.

1.2 Results

The program extracted from the proof of intuitionistic decidability presented here is the first to include a full propositional logic, *i.e.* the logic formalized here includes propositional variables, a constant denoting false, and operators for conjunction, disjunction, and implication. The extracted program is readable and efficient in the sense that it does not perform extraneous computation related to the logical part of the specification, nor does it contain unreadable artifacts of the proof in its text. These qualities will be most evident to those familiar with the state of the art in program extraction.

In the course of the development presented here, a number of minor errors in the ACU presentation were discovered, additionally a more serious error was uncovered. Indeed, discovering errors like these is one point of formal machine checked proofs. The presentation here differs from that of ACU in two significant ways. First, we have made modifications to the type theoretic formalization to guarantee the program extracted from the proof is free of the

[1] Of course intuitionistic propositional logic is known to be PSPACE complete, what we mean here by "efficient" is that the extracted program doesn't do unnecessary computation and that the program does not contain non-computational artifacts of the proof.

non-computational junk that often clutters programs extracted from constructive proofs. The methodology of using set types in place of existential quantifiers to generate efficient extracts has been described elsewhere [5, 6]. The second difference between the formalization presented here and that of ACU is in the proof type used as evidence of validity. We formalize a multi-succedent sequent calculus while ACU attepted to push the argument through for a single succedent calculus. Although the overall structure most of the details of the ACU proof survive in the version presented here, the ACU proof is incorrect. We simply remark that ACU failed to fully consider the case of reconstructing a proof object after the application of the tableau rule for a negatively occurring disjunction.

2 Nuprl

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 [7].

The Nuprl system supports construction of proofs by top-down refinement. The prover is implemented as a tactic based prover in the style of LCF. The tactic language is ML. Nuprl differs from other LCF-style provers in that tactic invocations define 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 Nuprl system also supports a unique and powerful display mechanism. Nuprl terms are edited using a structure editor; term structure is independent of display which is user specified. All Nuprl terms occurring in this paper are set in `typewriter font` and appear on the page as they do in the Nuprl editor and library.

Complete documentation is included in the Nuprl V4.2 distribution. [2]

2.1 Clean and Efficient Extracts

Methods of generating efficient and readable extracts by the use of the set type (as opposed to the existential type) and by efficient general recursion combinators have been presented by the author in [5, 4, 6]. We reiterate the main points here.

Inhabitants of the existential type `∃x:T.P[x]` are pairs `<a,b>` where `a∈T` and `b∈P[a]`. The term `b` inhabiting `P[a]` specifies, as far as the proofs-as-programs interpretation goes, how to prove `P[a]`. When an existential type occurs as a hypothesis it can be decomposed into two hypotheses, one of the form `a:T` and another asserting `b:P[a]`. If `v` is the name of the variable denoting the existential hypothesis, occurrences of `a` in the final extract appear as `v.1`, and occurrences of `b` appear as `v.2` (the first and second projections).

[2] The Nuprl system is freely available on the Nuprl group web pages at Cornell, `http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`.

Alternatively, consider the Nuprl set type `{y∈T|P[y]}`. Its inhabitants are elements of type `T`, say `a`, such that `P[a]` holds. Thus, a set type does not carry the computational content associated with the logical part `P[a]`. Since the proof that `P[a]` holds is not witnessed by inhabitants of the set type, the fact that `P[a]` holds is not freely available in parts of a proof where it might find its way into an extract. When a set type 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]`. The Nuprl system prevents the variable of a hidden hypothesis from appearing free in the extract of a proof by restrictions on its use. Hidden hypotheses are unhidden by the system in parts of the proof where no computational content is constructed.

Although these issues may appear to be Nuprl specific technicalities, they arise in all constructive systems implementing the proofs-as-programs interpretation.

2.2 Efficient Induction Schemes

We are interested in extracting efficient programs from proofs; to do so we carefully construct proofs of the induction principles to ensure their extracts are efficient recursion combinators.

The Nuprl standard library includes the following type characterizing well-founded binary relations:

```
WellFnd(A;x,y.R[x;y])  ≝
 ∀P:A → Prop.(∀j:A. (∀k:A. R[k; j] ⇒ P[k]) ⇒ P[j]) ⇒ ∀n:A. P[n]
```

Well-founded induction on the natural numbers over the ordinary less-than ordering is specified by a lemma of the form `WellFnd(ℕ;x,y. x < y)`.

The following recursion scheme inhabits this type.

```
λP,g. (letrec f(n) = g(n)(λk,p. f(k)))
```

Here `P` is a proposition (over type `A`), and `g` corresponds to the computational content of the induction hypothesis. In this scheme, `g` takes two arguments, the first being the principal argument on which the recursion is formed, while its second argument is a function inhabiting the proposition `∀k:A. R[k;j]⇒ P[k]`, *i.e.* a function which accepts some element `k` of type `A` along with evidence for `R[k;j]` and which produces evidence for `P[j]`. In the scheme, the evidence that `R[k;j]` holds takes the form of the argument `p` to the innermost **λ**-binding. The variable `p` occurs nowhere else in the term and does not contribute to the actual computation of `P[j]`; instead it is a vestige of the typing. In the context of any complete proof, this argument will be a term justifying `R[k;j]`. In any program extracted from a proof using this scheme, the useless argument `p` must be supplied. This term is non-computational junk.

As an alternative, we give the following definition of well-founded binary relations that hides the ordering in a set type; this type, simply called `WF` is defined as follows:

```
WF(A;x,y.R[x; y])  ≝
 ∀P:A → Prop.(∀j:A. (∀k:{k:A| R[k; j]} . P[k]) ⇒ P[j]) ⇒ ∀n:A. P[n]
```

Since the ordering relation is now hidden in the right side of a set type, it does not contribute to the computational content of the extracted programs. The recursion scheme extracted from a proof of this type is nearly identical to the previous one, but the extra (useless) lambda-abstraction is gone.

```
    λP,g. (letrec f(n) = g(n)(λk. f(k)))
```

For an arbitrary type T and a measure function ρ:T→ℕ, following lemma defines an efficient measure induction principle.

```
∀T:Type. ∀ρ:T → ℕ.  WF(T;x,y.ρ(x) < ρ(y))
Extraction:
  λT,ρ,P,g.(letrec f(n) = g(n)(λk.f (k)))
```

Note that the measure function ρ does not occur in the body of the extract, logically it belongs to the termination argument which is not part of the computational content.

The proof of intuitionistic decidability presented below is by induction on the lexicographic ordering of a pair of inverse images (measures functions mapping systems onto the natural numbers.) This induction principle is established by the following lemma.

```
∀T:Type. ∀ρ,ρ':T → ℕ.
  WF(T;k,j.ρ(k) < ρ(j) ∨ (ρ(k) = ρ(j) ∧ ρ'(k) < ρ'(j)))
Extraction:
  λT,ρ,ρ',P,g.(letrec f(n) = g(n)(λj.f j))
```

Note that the recursion combinator does not mention the measure functions.

3 The Tableau Algorithm

Our goal is to extract a tableau decision procedure from the formal proof. Tableau methods for proof search in intuitionistic logic go back to Beth [3] and are analyzed in detail by Fitting [11]. Roughly, tableau methods are search procedures that work by systematically exploring all consequences of an assumption in the search for a counter-example. For example, if a formula of the form $P \wedge Q$ is assumed to be false, then one of P or Q must also be false; the step of tableau development for this formula will split into two paths, one with the added assumption that P is false and the other with the added assumption that Q is false. The *tableau* is the tree-like structure that records the development of the search, keeping track in each node of those formulas assumed to true and those formulas assumed to be false.

If, in the process of developing a path of the tableau, it occurs that a formula is assumed to be both true and false, then that path is contradictory and we say it is *closed*. If a path is developed to the point where further application of the tableau rules can only result in redundant nodes being added to the path, then we stop development and say the path is *open*. If all the paths developed in this process are closed then the initial assumption must be false and the formula is provable; *i.e.* if the initial assumption that the formula is false always leads to a contradiction, then the formula must be true. Using the tableau so constructed we construct a proof of the formula. If on the other hand some path in the tableau is open, that path is interpreted as a Kripke counter-example to the initial formula.

It is easy to check whether a path is closed. The complexity of the decidability argument arises in determining whether further development of an open path is redundant. Underwood [18] provided a new termination argument based on a lexicographic ordering of tableau systems based on two measures:

`i1`: bounding the number of nodes that can be added to a tableau system, and
`i2`: bounding the number of formulas that can be added to any node.

Ultimately, these measures depend on the fact that tableau construction has the *subformula property*, *i.e.* in the tableau development, only subformulas of formulas already occurring in the tableau are ever added.

Measures `i1` and `i2` are calculated by computing conservative upper bounds on the sizes of the respective structures they measure and then by taking the difference between these bounds and the actual sizes of the objects in the tableau being constructed. Since nodes and systems grow during each step of tableau development, the difference decreases. Thus, at each step of the tableau construction process, one or the other measure decreases, which is enough to show termination. The bounds are never achievable in an actual tableau development and so we terminate the process when all nodes are completely developed and when the system is completely developed.

4 Intuitionistic Proof Systems, Kripke Counter-Examples and Tableau Systems

The final output of the algorithm we are interested in will either be a proof that the initial system is valid or a Kripke model serving as a counter-example, we formalize these structures now.

4.1 Formulas and Sequents

Propositional formulas are formalized by the following Nuprl recursive type:

$$\texttt{Formula} \overset{\text{def}}{=} \texttt{rec(F.Var | Unit | F} \times \texttt{F | F} \times \texttt{F | F} \times \texttt{F)}$$

Reading left to right, a formula is either: a variable (which is displayed as $\ulcorner\texttt{x}\urcorner$); the constant inhabiting the type `Unit` which is interpreted as *false* and displayed $\ulcorner\texttt{false}\urcorner$; a pair of formulas representing a conjunction displayed as $\texttt{p}\ulcorner\wedge\urcorner\texttt{q}$; a pair of formulas, representing a disjunction displayed $\texttt{p}\ulcorner\vee\urcorner\texttt{q}$); or a pair of formulas, representing an implication and displayed $(\texttt{p}\ulcorner\Rightarrow\urcorner\texttt{q})$. Negation $(\neg\texttt{P})$ is defined as $(\texttt{P}\ulcorner\Rightarrow\urcorner\ulcorner\texttt{false}\urcorner)$ and we do not include it explicitly in our formula type; neither do we include an operator for equivalence. `Formula` is a discrete type, *i.e.* it is decidable whether two formulas are equal.

We model the type of variables using the Nuprl `Atom` type; however, any discrete type may be substituted. other than this constraint, `Var` may be considered an uninterpreted type.

The sequent type (`Sequent`) consists of pairs of formula lists. If `S` is a sequent, `Hyps(S)` denotes the list of formulas that are in the antecedent of `S` (the hypotheses) and `Concl(S)` denotes the list of formulas in the succedent of `S` (the conclusions.) `Sequent` is a discrete type since `Formula` is.

A sequent is deemed true whenever the conjunction of the antecedents implies the disjunction of the succedents (by convention, an empty disjunction is true and an empty conjunction is false.)

4.2 Multi-Succedent Proofs

Our proof type is based on the sequent calculus *MJ* presented in Figure 1. *MJ* is essentially the propositional fragment of Dragalin's [8, pg.11] multi-succedent calculus. The form of our rules differs from Dragalin's in logically insignificant ways that support the use of lists instead of sets.

$$\frac{}{M, \texttt{false}, N \vdash C}\ (\texttt{false}\,l) \qquad \frac{}{M, q, N \vdash M', q, N'}\ (Ax)$$

$$\frac{q, M, q\vee r, N \vdash C \qquad r, M, q\vee r, N \vdash C}{M, q\vee r, N \vdash C}\ (\vee l) \qquad \frac{H \vdash q, M, q\vee r, N}{H \vdash M, q\vee r, N}\ (\vee r1) \qquad \frac{H \vdash r, M, q\vee r, N}{H \vdash M, q\vee r, N}\ (\vee r2)$$

$$\frac{q, M, q\wedge r, N \vdash C}{M, q\wedge r, N \vdash C}\ (\wedge l1) \qquad \frac{r, M, q\wedge r, N \vdash C}{M, q\wedge r, N \vdash C}\ (\wedge l2) \qquad \frac{H \vdash q, M, q\wedge r, N \qquad H \vdash r, M, q\wedge r, N}{H \vdash M, q\wedge r, N}\ (\wedge r)$$

$$\frac{M, q\Rightarrow r, N \vdash q, C \qquad r, M, q\Rightarrow r, N \vdash C}{M, q\Rightarrow r, N \vdash C}\ (\Rightarrow l) \qquad \frac{q, H \vdash r}{H \vdash M, q\Rightarrow r, N}\ (\Rightarrow r)$$

Fig. 1. System *MJ*

To read these rule schemas, M, N, C and H denote (possibly empty) formula lists and q and r denote individual formulas. Consider the figure for the rule labelled ($\Rightarrow r$), this rule characterizes the multi-conclusion intuitionistic sequent calculus. To derive the sequent $H \vdash M, q \Rightarrow r, N$ it is enough to show the sequent $q, H \vdash r$. Note that, in distinction to the other rules, the formulas in the succedent of the conclusion of ($\Rightarrow r$) (formulas in the list $M, q \Rightarrow r, N$) have been replaced by the single formula r.

MJ proofs are formally modeled in the Nuprl implementation in two stages. In the first, a recursive type of pre-proofs is defined to represent the shape (tree structure) of a proof. In the second stage, the type of pre-proofs is narrowed to include only those trees representing actual proofs.

```
pre_proof  =def  rec(P. Sequent
                    | Sequent × Sequent × P
                    | Sequent × Sequent × P × Sequent × P)
```

We display the three classes of `pre_proofs` as `C\`, `C\<H,p>`, and `C\<H,p>,<H',p'>` respectively where `C`, `H`, and `H'` are sequents and `p` and `p'` are pre-proofs.

For a pre-proof `P` let `Concl(P)` be the sequent that is the root of the pre-proof.

Excluding axioms, the rules of system *MJ* have either one or two hypotheses. These rule classes are characterized by two definitions, one for rules having a single hypothesis (`proof_rule1`: $\vee r1$, $\vee r2$, $\Rightarrow r$, $\wedge l1$, and $\wedge l2$) and another for rules having two hypotheses (`proof_rule2`: $\vee l$, $\wedge r$, and $\Rightarrow r$). We give the definition of `proof_rule1` here.

```
c\h is a rule instance  ≝
  ∃a,b:Formula.
     ((a⌈∨⌉b) ∈ Concl(c)  ∧  h = <Hyps(c),a::Concl(c)>)
     ∨  ((a⌈∨⌉b) ∈ Concl(c)  ∧  h = <Hyps(c),b::Concl(c)>)
     ∨  ((a⌈⇒⌉b) ∈ Concl(c)  ∧  h = <a::Hyps(c), b::[]>)
     ∨  ((a⌈∧⌉b) ∈ Hyps(c)  ∧  h = <a::Hyps(c), Concl(c)>)
     ∨  ((a⌈∧⌉b) ∈ Hyps(c)  ∧  h = <b::Hyps(c), Concl(c)>)
```

The equality used here is the type equality for sequents (defined as pairs of formula lists) and so order counts; this is not the semantic (permutation) equality on sequents. The reader can verify by inspection that these clauses match the appropriate rules of system *MJ.*

In the second stage of modeling *MJ* proofs. A well-formedness predicate is defined to narrow the class of pre-proofs to those structures that actually model proofs of system *MJ.* For a pre-proof `P` we write `P is a Proof` if:

i.) its leaves are all instances of the `false`l rule or the Ax rule, and
ii.) every non-leaf node matches a conclusion of some rule instance and its children match the premises of that rule.

This characterization is formalized by a recursive function we omit for lack of space. Thus proofs are characterized by the subtype of pre-proofs that are well formed.

```
    Proof  ≝   {p:pre_proof| p is a Proof}
```

A proof `P` proves a sequent `S` if `Concl(P) = S`.

4.3 Kripke Counter-examples as Evidence of Unprovability

It is a well known negative result that no finite valuation captures intuitionistic propositional logic. Thus, models for intuitionistic logic are necessarily more complex than models for classical logics. Following the account given by Underwood in [18], we use Kripke models to witness the unprovability of a formula. This interpretation is not without some subtlety as Kripke models provide for classical analyses of intuitionistic logic but are not faithful to intuitionistic semantics. Smorynski [17] and Dummett [9] discuss this in some detail. Never-the-less, following Underwood [18, pg.11–15], Kripke models are used here as evidence of unprovability. Failed tableau searches yield Kripke counter-examples. This use of Kripke models as counter-examples to intuitionistic provability has received attention elsewhere [15, 13].

The type of Kripke models is a dependent triple consisting of a type (of states), a reflexive and transitive relation on the states, and an atomic forcing function.

```
Kripke ≝ T:Type
     ×R:{R:(T × T) → Prop | Reflexive(R) ∧ Transitive(R)}
     ×{af:T → Var → Prop |
         ∀a:T. ∀v:Var. af(a)(v) ⇒ (∀b:T. R(<a, b>) ⇒ af(b)(v))}
```

The selectors for the three components of a Kripke model `K` are displayed as `Σ(K)`, `≤{K}`, and `K.af` respectively. For states `s` and `s'` we display `s≤{K}s'` for `≤{K}(<s,s'>)`.

Truth in a Kripke model is defined by the forcing relation. The statement of the main theorem requires definitions of both *forces*, and its complement *not forces*. The reader may realize that we cannot simply define the complementary notion by taking the constructive negation of the definition of forcing. To avoid this problem, following a suggestion of Underwood, we define the forces and not-forces relations simultaneously by mutual recursion. Definition by mutual recursion is not directly supported by Nuprl tactics (although there is no technical reason it cannot be) and we use the pairing trick to implement it.

```
<forces,not_forces>{K} ≝
    (letrec f_nf(s)(f) =
      case f:
        ⌜x⌝ → <K.af(s)(x), ¬(K.af(s)(x))>;
        ⌜false⌝ → <False, True>;
        a⌜∧⌝b → <(f_nf(s)(a)).1 ∧ (f_nf(s)(b)).1,
                 (f_nf(s)(a)).2 ∨ (f_nf(s)(b)).2>;
        a⌜∨⌝b → <(f_nf(s)(a)).1 ∨ (f_nf(s)(b)).1,
                 (f_nf(s)(a)).2 ∧ (f_nf(s)(b)).2>;
        a⌜⇒⌝b → <∀s':Σ(K). s ≤{K} s' ⇒
                    (f_nf(s')(a)).2 ∨ (f_nf(s')(b)).1,
                 ∃s':Σ(K). s ≤{K} s' ∧
                    (f_nf(s')(a)).1 ∧ (f_nf(s')(b)).2>;
    )
```

Using this definition we further define `forces(K,S,f)` and `not_forces(K,S,f)` to be the first and second projections of the term `<forces,not_forces>{K}(S)(f)`.

4.4 Tableau Systems

The tree structure representing an actual tableau is never explicitly constructed by the program extracted from the proof presented here. Rather, the paths in the tableau are represented by lists of tableau nodes, these lists are called *Systems* and the overall structure of the tableau is implicit in the unfolding of the recursion.

Like sequents, tableau nodes (type `Node`) are represented by pairs of formula lists. The elements in the first component of a node are those formulas assumed to be true, the elements in the second component are those elements assumed to be false. We refer to these components by writing `T(N)` for the true part and

F(N) for the false part. Of course, Node is a discrete type. asSequent(N) casts the node N to the type Sequent.

A System is a non-empty list of nodes.

There is a close correspondence between the steps of tableau construction and the proof rules of system *MJ*. For each proof rule there is a corresponding step of tableau development. For proof rules having a single premise there is a corresponding tableau development step in which an existing node is extended or, in the case of $\Rightarrow r$, the tableau system itself is extended by the addition of a new node. For proof rules having two premises, the corresponding tableau step extends an existing node in the tableau in two different ways, invoking the induction hypothesis (unfolding a step of recursion) on these extended systems. This bifurcation of systems corresponds to a branching in the tableau structure. We call the tableau steps corresponding to rules other than the $\Rightarrow r$ rule *local* rules, as they only extend existing nodes.

When a node has been developed as far as possible under the local rules we say it is *node complete* (we write nComplete(N).) The type of *eligible systems* (ESystem) are those systems restricted to contain at most one member that is not node complete. Tableau systems containing all possible node extensions induced by occurrences of $\Rightarrow r$ are called *system complete*; for a system S we write sComplete(S) to indicate S is system complete.

In the case of a failed tableau search, culminating in a system S, the corresponding Kripke structure K(S) will serve as the counter-example. Eventually, we are interested in viewing tableau systems as Kripke structures. The following function serves to map systems into a triple which is a Kripke model.

$$\texttt{K(S)} \overset{\text{def}}{=} \texttt{<\{N:Node| N}\in\texttt{S\} , }\boldsymbol{\lambda}\texttt{<n,m>.T(n)}\subseteq\texttt{T(m), }\boldsymbol{\lambda}\texttt{N,x.}\ulcorner\texttt{x}\urcorner\in\texttt{T(N)>}$$

Thus, under the interpretation, states of the corresponding Kripke model consist of the type whose members are those nodes in the system. The ordering on pairs of nodes is defined by sublist inclusion on the formulas assumed to be true in the nodes. The atomic forcing function for a state N and a variable x is defined by membership of the atomic formula $\ulcorner$x$\urcorner$ among formulas assumed true at N. That systems do indeed map to Kripke models under K is established by a well-formedness theorem for K.

5 Intuitionistic Decidability

A proof of a constructive disjunction $(P \vee Q)$ must indicate which of P or Q was proved and also must give evidence for its truth. Thus, if intuitionistic decidability is stated as follows:

$$\forall\texttt{S:Sequent. (}\exists\texttt{p:Proof. p proves S)} \;\vee\; \texttt{(}\exists\texttt{c:counter\_example. c refutes S)}$$

the resulting computational content is a function that takes a sequent as input and which either returns evidence for its validity or returns a counter-example.

We do not prove this theorem directly, but instead prove a more general theorem having the structure to support an inductive proof. The more general theorem does not apply directly to formulas, but applies to systems (lists of tableau nodes) satisfying the eligibility condition of being members of the type

ESystem. Evidence for the provability of an ESystem takes the form of a formal proof in the sequent calculus *MJ*. Evidence for its absurdity takes the form of a Kripke counter-example. Formally stated, the theorem we eventually prove here is the following:

```
∀S:ESystem
    (∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)} )
    ∨ {K:Kripke| ∃f:Node → Σ(K)
        ∀N:{N:Node| N∈S}
          ∀F:Formula. (F∈(T(N)) ⇒ forces(K,f N,F))
                    ∧ (F∈(F(N)) ⇒ not_forces(K,f N,F))}
```

To decide a formula ϕ, we will apply the computational content of this more general theorem to an eligible system containing a single node in which ϕ is assumed to be false. Should ϕ turn out to be provable, the result is a pair consisting of a tableau node and a proof of that node regarded as a sequent. Since the computational content of the theorem is intended to be applied to systems consisting of single nodes which contain a single formula, this evidently corresponds to a proof of the sequent <[],[ϕ]>. Should ϕ turn out not to be provable, the result is a Kripke counter-example. Kripke counter-examples here take the form of Kripke models defined over tableau nodes N∈S such that every formula in the true portion of the node (T(N)) is forced and every element in the false portion of the node (F(N)) is not forced. Since we will be applying the extracted program to initial systems consisting of a single nodes containing a single formula assumed to be false, the formula is not forced in the resulting Kripke model and so it serves as a counter-example.

5.1 The Proof

The proof of the theorem stated above is by induction on eligible systems, *i.e.* on systems having at most one node that is not node complete. The induction principle is the lexicographic measure induction presented above in Section 2.2. We apply it here using the measure functions i1 and i2 defined above in Section 3. Recall that the first measure decreases with every node added to the system while the second decreases as formulas are added to the eligible node.

After inducting on the eligible system S we are left with the following Nuprl state.

```
1.  S: ESystem
2. ∀k:{k:ESystem| k < S}
      (∃N:{N:Node| N∈k} . {p:Proof| p proves asSequent(N)})
      ∨ {K:Kripke| ∃g:Node → Σ(K)
          ∀N:{N:Node| N∈k}
            ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                      ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
⊢ (∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)})
    ∨ {K:Kripke| ∃g:Node → Σ(K)
        ∀N:{N:Node| N∈S}
          ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                    ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
```

Thus, we may assume (by hypothesis 2) that there is either a proof or a Kripke counter example for eligible systems lexicographically below `S`. The recursive structure (the outermost `letrec`) of the extracted program (see Fig. 2) arises from this step of induction.

Consider the eligible system `S` decalred in hypothesis 1 above; either all nodes in `S` are node complete or not. This property is decidable and appears in the extracted program as the first if-then-else clause.

Consider the else case first, *i.e.* there exists some node `N` in `S` that is not node complete (`¬nComplete(N)`). Since eligible nodes are expanded in place by adding subformulas of formulas already occurring in `S`, the tableau expansion steps for these rules reduce the measure `i2`. The proof rules $\vee r1$, $\vee r2$, $\wedge l1$, and $\wedge l2$ correspond to local tableau steps and all have one premise. In these cases, the induction hypothesis is instantiated with the system constructed from `S` by extending the eligible node with subformulas as specified by the corresponding proof rule. The proof rules $\vee l$, $\wedge r$, and $\Rightarrow l$ all have two premises and so we instantiate two copies of the induction hypothesis; one with the system constructed by expanding the eligible node with the subformulas specified in the left premise of the corresponding proof rule; and the other with a system created by expanding the eligible node by adding subformulas as specified by the right premise of the corresponding rule. In each case, the result of instantiating the induction hypothesis is a new hypothesis asserting the existence of a node-proof pair system or a Kripke counter-example for the extended (and therefore lexicographically smaller) system. Whenever a Kripke counter-example exists, it serves to refute the `S` as well. In the case a node-proof pair results from the instantiated induction hypotheses, they are used to identify a node in `S` and to construct a proof for it. Instantiations of the induction hyptothesis in the proof generates a recursive call to the tableau procedure in the extract (Fig. 2). The computations corresponding to the seven local rules can be identified in the extract.

Suppose instead that there is no eligible node in `S` (this is the then-clause of the outermost if-then-else in the extracted program.) Either the system is system complete (`sComplete(S)`) or not. If it is not system complete then there is some node containing an occurrence of $\Rightarrow r$ (say of the form `P`$\ulcorner\Rightarrow\urcorner$`Q`) which has not been accounted for in `S`, call this node `N`. In this case, decompose the induction hypothesis with the system constructed by extending `S` with a new node constructed from `N` which accounts for the application of the $\Rightarrow r$ rule. This new node is constructed by replacing `F(N)` with the single formula `Q` and by adding the formula `P` to the formulas in `T(N)` . This extended system is lower in the lexicographic ordering of systems since the measure `i1` is reduced whenever a node is added to `S`. As above, the instantiation of the induction hypothesis results in a recursive call to the tableau procedure in the extracted program, which returns either a node-proof pair or a Kripke counter-example for the expanded system.

Finally, if all nodes are complete and the system is complete, then we are in the base case where one of a node-proof pair or a Kripke counter example is constructed directly without reference to the induction hypothesis. This is accounted for in the extract by a call to the extract of the lemma (`decidability_base`):

```
∀S:ESystem
    sComplete(S)
    ⇒ ∀N∈S.nComplete(N)
    ⇒ (∃N:{N:Node| N∈S} . {p:Proof| p proves asSequent(N)})
       ∨ {K:Kripke| ∃g:Node → Σ(K)
           ∀N:{N:Node| N∈S}
             ∀f:Formula. (f∈(T(N)) ⇒ forces(K,g(N),f))
                       ∧ (f∈(F(N)) ⇒ not_forces(K,g(N),f))}
```

If the system contains a node that, viewed as a sequent, is an instance of an axiom, then that node is returned paired with the instance of the axiom rule. If not, then a Kripke counter-model is constructed by applying the function `K` to the system (defined above in Section 4.4.)

This completes the proof of decidability in the intuitionistic case.

5.2 Remarks on the Extract

Figure 2 exhibits the extracted decision procedure. The program shown there has been symbolically transformed within Nuprl using the direct computation system to eliminate some unnecessary steps of computation. This mostly entails β-reducing occurrences of applications of the identity function. These transformations are entirely formal and since, by the semantics of Nuprl, direct computation is allowed anywhere within a term, they do not change the meaning of the program. The program has further been hand edited to format it and to rename unreadable system generated variable names. This is only for display.

6 Future Work

Study of the extracted program reveals that there is room for the introduction of abstractions which would both make the extracted program clearer and would result in a shorter proof. This process of tuning a proof by examination of the extract and of tuning the extract by studying the proof is an interesting part of the methodology of using a constructive system like Nuprl.

Integrating of the extracted decider for intuitionistic propositions into Nuprl is an immediate goal. However, if we are to preserve Nuprl's program extraction capabilities, this poses some problems. Nuprl's proof system is a single succedent sequent calculus. To repair the error in the ACU proof we have resorted to a multi-succedent calculus. Egly and Schmidtt [10] give cut-free translations of multi-succedent proofs into single succedent proofs which preserve reasonable extracts.

The program extracted here can easily be translated into ML and used as part of a tactic to decide propositional fragments of Nuprl's type theory. The resulting tactic would fail, returning the Kripke model as evidence against the validity of a formula should it turn out not to be valid; alternatively, it would use the formal proof returned by the decision procedure, in concert with the Egly

```
letrec tableau(S) =
  if ∀N∈S.nComplete(N) then
    if sComplete(S) then
      ext{decidability_base}(S)(·)(·)
    else let <N,a,b,mp,_> = (ext{not_system_complete}(S)(·)) in
            case tableau(<a::T(N), b::[]>::S)
            of inl(<N1,p1>) =>
              inl(if (N1 = <a::T(N), b::[]>)
                  then <N, let <p',_> =
                             (ext{imp_right_proof}
                               (N)(a)(b)(mp)(p1)(Ax)) in p'>
                  else <N1, p1>)
            | inr(K) => inr(K)
  else
    let <N,t> = (∃N:{N:Node | N∈S}. ¬nComplete(N))
      let <a,b,op_type> = (ext{not_node_complete}(N)(t)) in
        case op_type of inl(<_,V14>) => case V14
           of inl(_) =>
             case tableau(<a::T(N), F(N)>::remove(N;S))
             of inl(<N1,p1>) =>
               inl(if (N1 = <a::T(N), F(N)>)
                   then <N, mk_proof(N,<N1,p1>)>
                   else <N1, p1>)
             | inr(K) => inr(K)
           | inr(_) =>
               case tableau(<b::T(N), F(N)>::remove(N;S))
               of inl(<N1,p1>) =>
                 inl(if (N1 = <b::T(N), F(N)>)
                     then <N, mk_proof(N,<N1,p1>)>
                     else <N1, p1>)
               | inr(K) => inr K
        | inr(V13) => case V13
           of inl(_) =>
             case tableau(<a::T(N), F(N)>::remove(N;S))
             of inl(<N1,p1>) =>
                 if (N1 = <a::T(N), F(N)>) then
                   case tableau(<b::T(N), F(N)>::remove(N;S))
                   of inl(<N2,p2>) =>
                     inl(if (N2 = <b::T(N), F(N)>)
                          then <N, mk_proof(N,<N1,p1>,<N2,p2>)>
                          else <N2, p2>)
                   | inr(K) => inr(K)
                 else  inl(<N1, p1>)
             | inr(K) => inr(K)
        | inr(V15) => case V15
            of inl(_) =>
              case tableau(<b::T(N),F(N)>::remove(N;S))
              of inl(<N1,p1>) =>
                if (N1 = <b::T(N), F(N)>) then
                  case tableau(<T(N), a::F(N)>::remove(N;S))
                  of inl(<N2,p2>) =>
                    inl(if (N2 = <T(N), a::F(N)>)
                        then <N, mk_proof(N,<N1,p1>,<N2,p2>)>
                        else <N2, p2>)
                  | inr(K) => inr(K)
                else  inl(<N1, N2>)
              | inr(K) => inr(K)
            | inr(V17) => case V17
                of inl(_) =>
                  case tableau(<T(N), b::F(N)>::remove(N;S))
                  of inl(<N1,p1>) =>
                    if (N1 = <T(N), b::F(N)>) then
                      case tableau(<T(N), a::F(N)>::remove(N;S))
                      of inl(<N2,p2>) =>
                        inl(if (N2 = <T(N), a::F(N)>)
                          then <N, mk_proof(N,<N2,p2>,<N1,p1>)>
                          else <N2, p2>)
                      | inr(K) => inr(K)
                    else inl(<N1, p1>)
                  | inr(K) => inr(K)
                | inr(V19) => let <_,V21> = V19 in
                      case V21
                      of inl(_) =>
                        case
                         tableau(<T(N), a::F(N)>::remove(N;S))
                        of inl(<N1,p1>) =>
                          inl(if (N1 = <T(N), a::F(N)>)
                          then  <N, mk_proof(N,<N1,p1>)>
                          else <N1, p1>)
                    | inr(K) => inr(K)
                  | inr(_) =>
                      case tableau(<T(N), b::F(N)>::remove(N;S))
                      of inl(<N1,p1>) =>
                        inl(if (N1 = <T(N), b::F(N)>)
                            then <N, mk_proof(N,<N1,p1>)>
                            else <N1, p1>)
                      | inr(K) => inr(K)
```

Fig. 2. The extract of the decidiability proof

and Schmidtt procedure, to construct a Nuprl tactic, which it could then apply to discharge the goal.

Another line of development that needs to be explored is the reflection of this decision procedure into Nuprl. Reflection [2, 1] was the motivation for the proof outlined in [1].

References

1. William Aitken, Robert Constable, and Judith Underwood. Metalogical frameworks II: Using reflected decision procedures. *Unpublished Manuscript*.
2. Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 95–197. IEEE, June 1990.
3. E. W. Beth. *The Foundations of Mathematics*. North-Holland, 1959.
4. J. L. Caldwell. Classical propositional decidability via Nuprl proof extraction. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving In Higer Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, 1998.
5. James Caldwell. Moving proofs-as-programs into practice. In *Proceedings, 12th IEEE International Conference Automated Software Engineering*, pages 10–17. IEEE Computer Society, 1997.

6. James Caldwell. *Decidability Extracted: Synthesizing "Correct-by-Construction" Decision Procedures from Constructive Proofs.* PhD thesis, Cornell University, Ithaca, NY, August 1998.
7. Robert L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
8. A. G. Dragalin. *Mathematical Intuitionism: Introduction to Proof Theory*, volume 67 of *Translations of Mathematical Monographs.* American Mathematical Society, 1987.
9. Michael Dummett. *Elements of Intuitionism.* Oxford Logic Series. Clarendon Press, 1977.
10. U. Egly and S. Schmitt. Intuitionistic proof transformations and their application to constructive program synthesis. In J. Calmet and J. Plaza, editors, *Proceedings of the 4^{th} International Conference on Artificial Intelligence and Symbolic Computation (AISC'98)*, number 1476 in Lecture Notes in Artificial Intelligence, pages 132–144. Springer Verlag, Berlin, Heidelberg, New-York, 1998.
11. Melvin Fitting. *Proof Methods for Modal and Intuitionistic Logics*, volume 169 of *Synthese Library.* D. Reidel, 1983.
12. Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic.* Foundations of Computing. MIT Press, Cambridge, MA, 1988.
13. J. Hudelmaier. A note on Kripkean countermodels for intuitionistically unprovable sequents. In W. Bibel, U. Furbach, R. Hasegawa, and M. Stickel, editors, *Seminar on Deduction*, February 1997. Dagstuhl report 9709.
14. C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5-6):607–640, 1993.
15. L. Pinto and R. Dyckhoff. Loop-free construction of counter-models for intuitionistic propositional logic. In *Symposia Gaussiana*, pages 225–232, Berlin, New York, 1995. Walter de Gruyter and Co.
16. N Shankar. Towards mechanical metamathematics. *J. Automated Reasoning*, 1(4):407–434, 1985.
17. C. A. Smorynski. Applications of Kripke models. In A. Troelstra, editor, *Metamathematical Investigation of Intuitionistic Mathematics*, volume 344 of *Lecture Notes in Mathematics*, pages 324–391. Springer-Verlag, 1973.
18. J. Underwood. *Aspects of the Computational Content of Proofs.* PhD thesis, Cornell University, 1994.
19. Judith Underwood. The tableau algorithm for intuitionistic propositional calculus as a constructive completeness proof. In *Proceedings of the Workshop on Theorem Proving with Analytic Tableaux, Marseille, France*, pages 245–248, 1993.
20. 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*, volume 918 of *Lecture Notes in Artificial Intelligence.* Springer, 1995.
21. K. Weich. Decision procedures for intuitionistic propositional logic by program extraction.
22. K. Weich. Private communication. February 1998.