Tableau for Intuitionistic Predicate Logic as Metatheory

Judith Underwood*

February 20, 1995

Abstract

We show how the tableau procedure for intuitionistic predicate logic can be interpreted as the computational content of a metatheorem about Kripke models for the logic. The metatheory we use is based on constructive type theory, but we shall explore classical extensions to the theory. We begin by describing how infinite tableau developments may be represented as co-inductive types in such a theory. We then show how a bounded tableau search procedure can be interpreted as the computational content of a theorem about prefixes of (possibly infinite) Kripke countermodels. Next, we examine how classical reasoning may be introduced in the proof without affecting the computational content. This leads to a stronger theorem with the same bounded tableau search as its computational content. Finally, we discuss the unbounded tableau procedure as the computational content of a related theorem.

In previous work [14], we showed that the tableau algorithm for intuitionistic propositional logic could be expressed as the computational content of a proof of completeness of finite Kripke models for the logic. The resulting tableau algorithm produced, for any propositional formula φ, either a proof of φ or a finite Kripke model in which φ is not forced. In this paper, we

*To appear in the Proceedings of the Fourth Workshop on Theorem Proving with Analytic Tableaux and Related Methods. This research was supported by EPSRC grant GR/H73103 and NSF grant CCR-9244739.

explore extensions of this result to the case of intuitionistic predicate logic. Since provability for intuitionistic predicate logic is not decidable, the computational content of a metatheorem cannot be a decision procedure; however, there are a number of ways in which the undecidability can be manifested while still preserving the computational value of the proof in the form of a tableau procedure.

The proofs will be carried out in a framework based on constructive type theory. Some versions of the theorem will be based on deviations from a purely computational metatheory; however, these will be clearly highlighted.

The first part of the paper introduces co-inductive types as a tool for representing infinite tableau developments. We then describe the tableau developments for predicate logic as elements of a co-inductive type. Using this definition, we show how a bounded tableau search procedure can be interpreted as the computational content of a theorem about prefixes of (possibly infinite) Kripke countermodels.

We then discuss a variation on this theorem which uses nonconstructive logic in its proof but still retains computational content. Some previous work [15, 4] on intuitionistic metatheory for intuitionistic predicate calculus is explained in this context. Finally, we use a computational interpretation of classical logic to show how an unbounded tableau search can be represented as the computational content of a proof of a metatheorem.

1 Constructive Models in Type Theory

In order to describe the possibly infinite tableau development in type theory, we use *co-inductive types*. Just as inductive types are the least fixed point of a type equation $T = F(T)$, co-inductive types are defined to be the greatest fixed point of a type equation. In his thesis, Mendler [9] described how co-inductive types could be added to Nuprl, and co-inductive types have also been treated in other contexts [8, 2, 12].

The tableau procedure can be described as a search for a countermodel for a formula; should this search fail, the result is a proof. (It is straightforward to extract a sequent proof from a closed tableau [6].) The general approach is similar to that in Dummett [5]; however, we combine his notions of "proof tree-trunk" and "refutation tree-trunk" (finite approximations to proofs and countermodels) into a single structure. We focus in this section on how this structure can be implemented in constructive type theory.

1.1 Introduction to co-inductive types

To understand the idea behind co-inductive types, we first consider an example of an inductive type. The type of lists of natural numbers can be defined as a recursive type, i.e.

$$NatList = Nil + Nat \times NatList$$

where $+$ denotes disjoint union, and $\times$ represents pairing.

However, there is actually more than one type satisfying the above equation. The type of lists of natural numbers is one, but the type of possibly infinite natural number streams also meets the conditions. Generally, recursive types are constructed so that the result is the least fixed point of the type equation. This can be made more explicit by the use of a least fixed point operator μ. The definition above could be rewritten

$$NatList = \mu T.Nil + Nat \times T$$

where T takes the place of $NatList$ in the right hand side of the definition.

If, instead, we want the greatest fixed point of the type equation, we use the ν operator. So,

$$NatStream = \nu T.Nil + Nat \times T$$

is the type of possibly infinite streams of natural numbers.

To define elements of ν-types, we use a ν_ind form $\nu\_ind(d; z, y.b) \in \nu T.B$, where d is an element of some type D, y is a variable of type D, and z is a variable with function type $D \rightarrow T$. For example, a stream of natural numbers in the type $\nu T.Nat \times T$ is defined:

$$\nu\_ind(0; z, y.\langle y, z(y+1)\rangle)$$

The ν_ind form is evaluated lazily, according to the rule

$$\nu\_ind(d; z, y.b) \rightarrow_1 b[\lambda x.\nu\_ind(x; z, y.b)/z, d/y]$$

so that, when evaluation is forced, we have

$$\begin{aligned}\nu\_ind(0; z, y.\langle y, z(y+1)\rangle) &\rightarrow_1 \langle 0, \lambda x.\nu\_ind(x; z, y.\langle y, z(y+1)\rangle)(1)\rangle \\ &\rightarrow_1 \langle 0, \nu\_ind(1; z, y.\langle y, z(y+1)\rangle)\rangle\end{aligned}$$

Further details on these types are are in Mendler's thesis [9].

In the current case, the data structure we use to describe the type of Kripke models and tableau proofs will represent the developing tableau. The general form of the definition is that of a possibly infinite labelled binary tree

$$\nu T.(S + S \times T + S \times T \times T)$$

where S is the type of the information contained at each stage of the developing tableau. In other words, S actually represents a partial tableau, and the children of a node represent the tableau or tableaux resulting from the application of a rule to S. (For this reason, the type permits points on a branch to have one or two subtrees.) There are other choices for representing tableau developments as elements of co-inductive types; this representation allows decisions to be made (about rule applications, closure conditions, etc.) using only local information.

The ν_ind form will be used to define the developing tableau inductively.

1.2 Types for Kripke models and tableau proofs

In this section, we define the type underlying tableau proofs and Kripke model developments using the ν types described in the previous section. Since we have the general form of the definition, we need only determine precisely what information is required at each stage of the development. We then describe the tableau development itself as a function from an initial set of information to the tableau with that information at its root.

Recall that a Kripke model for intuitionistic predicate logic is a tuple

$$\langle K, \leq, af, dom\rangle$$

such that K is a set of states partially ordered by $\leq$, af is a relation between K and atomic formulas which is monotone with respect to the order on K and which describes the atomic formulas forced in each state, and dom is a monotone function from K to sets of constants available at each state. We wish to be able to construct a Kripke model from a finite failed tableau, so this will guide our definition of the type of tableau developments.

The information required at each stage of the tableau development includes a set of formulas tagged with their truth values. So, we will define a *partial proof tree* (PPT) to be a (possibly infinite) tree labelled at each vertex with these sets. However, we require more information than the sets of formulas alone provide. To each tagged formula, we add a label describing the state in the Kripke model with which the formula should be associated. Let *Labels* be the type of this information. These labels must eventually describe a partial order on the set of states. Since states may be added to the set during development, we will need a function *NewLabel* which, given a label ℓ, returns a new label associated with a state greater than ℓ in the partial order and incomparable to all other states not less than ℓ. We assume $\ell_0 \in \mathit{Labels}$ is given as a starting point.

Furthermore, since we may have formulas tagged with different labels at any given stage in the tableau development, we will make explicit the domain associated with each formula. So, the information required at each stage in the tableau consists of a set of formulas, each associated with a label (representing a state in the possible Kripke countermodel), a truth value, and a set of constants in the domain at that state. (These constants also play the role of parameters in the tableau.) We consider the information associated with each formula to be tagged onto the formula, so we will define a type *TaggedFormula* to represent it. We will refer to the collection of this information present at each vertex in the partial proof tree as a *stage descriptor (SD)*.

Now we may describe the type of our partial proof trees. Let *CSet* be the type of sets of constants. Let *TaggedFormula* be the 4-tuple

$$\mathit{CSet} \times \mathit{Labels} \times \{T, F\} \times \mathit{Wff}$$

and let *SD* be the type of sets of *TaggedFormula*. The type of partial proof trees is then

$$PPT = \nu T.(SD + (SD \times T) + (SD \times T \times T))$$

It will be clear from context which of the three elements of the disjoint union is intended.

We may now define a function which represents the development of a tableau. This function takes a stage descriptor and returns a description of a tableau development rooted at that stage. For each node, the function chooses a rule which applies to a formula in the stage descriptor, applies it, and returns the resulting (partial) tableau. In each stage descriptor, we keep information about the entire branch leading to that stage, so that the decision of which tableau rule to apply can be made with local information.

In the rules given below, when the domain and label of the elements of SD are not specified, they are assumed to be the same for all tagged formulas mentioned in the rule. In practice, domains (i.e. elements of $CSet$) will be finite, so issues involved in quantifying over them will be disregarded for the moment. Tagged formulas will be given their usual notation of $T\alpha$ or $F\alpha$. We use the definition $\neg A = A \rightarrow \perp$ to avoid extra rules for negation.

Develop: $SD \rightarrow PPT$ is defined using the ν_ind form. The outline of the definition is

$$Develop = \lambda S_0 . \nu\_ind(S_0; z, S.\text{cases depending on } S)$$

where S_0 is an element of SD, S is a variable of type SD, and z is a variable of type $SD \rightarrow T$. During reduction, the variable z is the placeholder for the term representing the recursive call; for clarity, we use the variable name *develop* instead of z. The outline of the definition of *Develop* is then

$$Develop = \lambda S_0 . \nu\_ind(S_0; develop, S.\text{cases depending on } S)$$

and the cases are as follows:

$\boldsymbol{T\alpha \wedge \beta}$ If $(T\alpha \wedge \beta) \in S$ and $(T\alpha \notin S$ or $T\beta \notin S)$ then return

$$\langle S, develop(S \cup \{T\alpha, T\beta\})\rangle$$

$\boldsymbol{F\alpha \wedge \beta}$ If $(F\alpha \wedge \beta) \in S$ and $(F\alpha \notin S$ and $F\beta \notin S)$ then return

$$\langle S, develop(S \cup \{F\alpha\}), develop(S \cup \{F\beta\})\rangle$$

$\boldsymbol{T\alpha \vee \beta}$ If $(T\alpha \vee \beta) \in S$ and $(T\alpha \notin S$ and $T\beta \notin S)$ then return

$$\langle S, develop(S \cup \{T\alpha\}), develop(S \cup \{T\beta\})\rangle$$

$\boldsymbol{F\alpha \vee \beta}$ If $(F\alpha \vee \beta) \in S$ and $(F\alpha \notin S$ or $F\beta \notin S)$ then return

$$\langle S, \mathit{develop}(S \cup \{F\alpha, F\beta\})\rangle$$

$\boldsymbol{T\alpha \rightarrow \beta}$ If $(T\alpha \rightarrow \beta) \in S$ and $(F\alpha \notin S$ or $T\beta \notin S)$ then return

$$\langle S, \mathit{develop}(S \cup \{F\alpha\}), \mathit{develop}(S \cup \{T\beta\})\rangle$$

$\boldsymbol{F\exists x.\varphi(x)\beta}$ If $\langle C, \ell, F, \exists x\varphi(x)\rangle \in S$ and $\exists c \in C$, $\langle C, \ell, F, \varphi(c)\rangle \notin S$ then return

$$\langle S, \mathit{develop}(S \cup \{\langle C, \ell, F, \varphi(c)\rangle | c \in C\})\rangle$$

$\boldsymbol{T\forall x.\varphi(x)\beta}$ If $\langle C, \ell, T, \forall x\varphi(x)\rangle \in S$ and $\exists c \in C$, $\langle C, \ell, T, \varphi(c)\rangle \notin S$ then return

$$\langle S, \mathit{develop}(S \cup \{\langle C, \ell, T, \varphi(c)\rangle | c \in C\})\rangle$$

$\boldsymbol{T\exists x.\varphi(x)\beta}$ If $\langle C, \ell, T, \exists x\varphi(x)\rangle \in S$ and $\forall a \in C$, $\langle C, \ell, T, \varphi(a)\rangle \notin S$ then let c be a new constant, let $\hat{C} = C \cup \{c\}$ and let

$$\begin{aligned} S' &= \{\langle C', \ell', V, \alpha\rangle | (\ell' \neq \ell \text{ and } \langle C', \ell', V, \alpha\rangle \in S) \\ &\quad \text{or } (\ell' = \ell \text{ and } \langle C, \ell, V, \alpha\rangle \in S \text{ and } C' = \hat{C})\} \\ &\cup \{\langle \hat{C}, \ell, T, \varphi(c)\}\end{aligned}$$

and return

$$\langle S, \mathit{develop}(S')\rangle$$

This procedure adds the constant c to the domain of all formulas associated with the label ℓ and adds the tagged formula $T\varphi(c)$ to the set of formulas associated with ℓ.

The next two rules are the only ones which require the addition of new labels, since they are the only ones which add states to the tableau. Recall that *NewLabel*(ℓ) returns a label greater than ℓ and incomparable to any other label not less than ℓ.

$\boldsymbol{F\alpha\to\beta}$ If $\langle C, \ell, F, \alpha\to\beta\rangle \in S$ and $(\forall \ell_0 \geq \ell . \langle C, \ell_0, T, \alpha\rangle \notin S$ or $\langle C, \ell_0, F, \beta\rangle \notin S)$ then let $\ell' = \mathit{NewLabel}(\ell)$, let

$$S' = S \cup \{\langle C, \ell', T, \gamma\rangle | \langle C, \ell, T, \gamma\rangle \in S\} \cup \{\langle C, \ell', T, \alpha\rangle, \langle C, \ell', F, \beta\rangle\}$$

and return

$$\langle S, \mathit{develop}(S')\rangle$$

$\boldsymbol{F\forall x.\varphi(x)\beta}$ If $\langle C, \ell, F, \forall x\varphi(x)\rangle \in S$ and $\forall a \in C$, $\langle C, \ell, F, \varphi(a)\rangle \notin S$ then let c be a new constant, let $C' = C \cup \{c\}$, let $\ell' = \mathit{NewLabel}(\ell)$, let

$$S' = S \cup \{\langle C', \ell', T, \gamma\rangle | \langle C, \ell, T, \gamma\rangle \in S\} \cup \{\langle C', \ell', F, \varphi(c)\rangle\}$$

and return

$$\langle S, \mathit{develop}(S')\rangle$$

Closure if there is a formula α such that $T\alpha \in S$ and $F\alpha \in S$, or if $T\bot \in S$, then return $\emptyset$. This signifies the closure of this branch of the development.

Completion of countermodel If none of the preceding rules apply, then return S. This branch cannot be further developed, and describes a countermodel.

The result of the application of *Develop* is a partial proof tree P which must be evaluated lazily. We assume there is an operator $\mathit{unwind}(n, P)$ which unrolls the definition of P n stages along each branch.

Note that there is a function from *Labels* to *CSet* describing the set C associated with a label ℓ. There is only one such set because whenever the set C is modified, either we have created a new label or all tuples involving the label are updated with the new set C.

We stipulate that each element of S be chosen as a candidate for the application of a rule in rotation. This means that if a rule can be applied to an element of S and no application of another rule causes that rule to be no longer applicable, then eventually the rule will be applied. The reason for this restriction will be explained in the next section.

1.3 Correctness of the construction

We demonstrate that the partial proof trees created by evaluation of the *Develop* function defined in the previous section are a sound representation of tableau developments.

A tableau development for φ is a partial proof tree beginning with $F\varphi$ and such that along each branch, each stage descriptor is the result of the application of some tableau rule to the previous one. Given a provable formula φ, it is not the case that every possible tableau development produces a tableau proof. However, if a formula is provable, there exists a tableau proof, that is, a terminating tableau development such that all branches are closed. Although the rules given in the definition of *Develop* are standard tableau development rules, some justification must be made for the correctness of the algorithm which results from the evaluation of *Develop*. To do this, we prove:

Theorem 1 *If φ is provable, the evaluation of $Develop(\langle\emptyset, \ell_0, F, \varphi\rangle)$ will terminate.*

Proof: If φ is provable, then there is a standard tableau proof of φ, i.e. a tableau development which is closed on all branches. Call this tableau proof $\vec{T}$. We need to show that a tableau (not necessarily the same tableau) is discovered by the evaluation of $T = Develop(\langle\emptyset, \ell_0, F, \varphi\rangle)$ according to the rules above. In particular, we show that the rules applied on a branch in $\vec{T}$ are equivalent to ones applied on some branch in T (possibly modulo a uniform renaming of added constants), and that every branch in T contains all consequences of rules applied on some branch in $\vec{T}$.

First, since all rules which can always be applied are eventually applied, if a tableau rule present in $\vec{T}$ is not applied, it is because the consequences of its application were already present in the formulas at that stage as a result of some other rule application. So, all rules in $\vec{T}$ are either applied or have their consequences subsumed by the application of another rule. (We do not consider the particular choice of new constant introduced to be essential to the consequences of a rule. In λ-calculus terms, we identify α-convertible tableaux.) In either case, the consequences of all applications of rules on some branch in $\vec{T}$ are present on some branch in T.

Second, note that although the development of T at a given state may have more branches than $\vec{T}$, every branch of T corresponds to some branch of $\vec{T}$ in that it contains at least the tagged formulas present on that branch. Whenever a rule creating two branches is applied in T which was not applied in $\vec{T}$, both new branches still contain all the information on the single branch in $\vec{T}$.

So, since every branch of T represents the consequences of some branch of $\vec{T}$, and all branches in $\vec{T}$ close, every branch of the development of T closes. $\Box$

In the next section we shall need the following easy lemma:

Lemma 2 *Suppose ℓ is such that no rule can be applied to any tuple of the form $\langle C, \ell, V, \alpha \rangle$ in S. Then we have the following properties (omitting C and ℓ for brevity):*

$$
\begin{array}{rcl}
T\alpha \wedge \beta \in S & \Rightarrow & T\alpha \in S \text{ and } T\beta \in S \\
F\alpha \wedge \beta \in S & \Rightarrow & F\alpha \in S \text{ or } F\beta \in S \\
T\alpha \vee \beta \in S & \Rightarrow & T\alpha \in S \text{ or } T\beta \in S \\
F\alpha \vee \beta \in S & \Rightarrow & F\alpha \in S \text{ and } F\beta \in S \\
T\alpha \rightarrow \beta \in S & \Rightarrow & F\alpha \in S \text{ or } T\beta \in S \\
F\alpha \rightarrow \beta \in S & \Rightarrow & \exists \ell', C'. \ell \leq \ell' \langle C', \ell', T, \alpha \rangle \in S \text{ and } \langle C', \ell', T, \beta \rangle \in S \\
T\exists x.\alpha(x) \in S & \Rightarrow & \exists c \in C. T\alpha(c) \in S \\
F\exists x.\alpha(x) \in S & \Rightarrow & \forall c \in C. F\alpha(c) \in S \\
T\forall x.\alpha(x) \in S & \Rightarrow & \forall c \in C. T\alpha(c) \in S \\
F\forall x.\alpha(x) \in S & \Rightarrow & \exists \ell', C'. \ell \leq \ell' \text{ and } \exists c \in C' \langle C', \ell', F, \alpha(c) \rangle \in S
\end{array}
$$

Proof: If any of the above conditions failed, the corresponding rule would be applicable. $\Box$

2 Constructive Version of the Completeness Theorem

Given the problems inherent in reasoning constructively about infinite objects and undecidable properties, it is not surprising that our constructive completeness theorem is fairly weak. Its computational content is a bounded decision procedure. Given a formula φ and a bound n, the algorithm performs an n-depth tableau search for a tableau proof of φ. If all branches of the tableau close at a stage less than or equal to n, then the closed proof is returned. Otherwise, the open branches can be shown to be consistent with a prefix of a countermodel for φ.

This latter condition, while not completely satisfactory, is about the best we can hope for. Since we cannot bound the depth to which the tableau must be searched, we cannot be certain that we have found a countermodel if tableau development rules can still be applied.

Before we prove the theorem, we make the notion of a prefix of a possible countermodel for φ precise:

Definition 2.1 *A prefix of a possible countermodel for φ is a finite Kripke model*

$$\langle K, \leq, af, dom \rangle$$

with a minimum state $k_0 \in K$, a subset of K designated as open *states, and a pair of functions Treq and Freq from K to sets of formulas, satisfying*

1. *for all atomic formulas α and for all $k \in K$, if $\alpha \in Treq(k)$ then $af(k, \alpha)$*
2. *for all $k \in K$, $Treq(k) \cap Freq(k) = \emptyset$*
3. *for all open $k \in K$, if we assume k forces all formulas in $Treq(k)$ and does not force any formula in $Freq(k)$, then φ is not forced at k_0.*

Note that if a prefix of a possible countermodel for φ is such that there are no open $k \in K$, we must have that K is a countermodel.

We may now state the constructive completeness theorem.

Theorem 3

$$\forall \varphi . \forall n : N$$

$$\textit{let } P = \textit{unwind}(n, \textit{Develop}(\{\langle \emptyset, \ell_0, F, \varphi \rangle\})) \textit{ in}$$
$$P \textit{ is a closed tableau proof of } \varphi \textit{ or}$$
$$P \textit{ contains a description of a prefix of a possible countermodel for } \varphi$$

Proof: If all branches of P are closed, it is a closed tableau proof. If not, there is at least one branch which is not closed. We use the information on this branch to produce a prefix of a possible countermodel for φ. There are only finitely many branches, so we can pick any branch which has not closed, and let S be the last stage descriptor on this branch. Let

$$K = \{L \in \mathit{Labels} | \exists C : \mathit{CSet}, V \in \{T, F\}, \alpha : \mathit{Wff}. \langle C, L, V, \alpha \rangle \in S\},$$

let the order on K be the inherited order on Labels, let $af(k, \alpha)$ be true when α is atomic and $\exists C. \langle C, k, T, \alpha \rangle \in S$ and let $dom(k)$ be any set C such that $\langle C, k, V, \alpha \rangle \in S$ for some truth value V and formula α. (Recall that all such sets C are the same for any given k.)

Finally let $k_0 = \ell_0$, let the open states in K be the states corresponding to labels in S which match the conditions for the application of some development rule, and let $\mathit{Treq}(k) = \{\alpha | \langle dom(k), k, T, \alpha \rangle \in S\}$ and $\mathit{Freq}(k) = \{\alpha | \langle dom(k), k, F, \alpha \rangle \in S\}$ We now show that with these functions,

$$\langle K, \leq, af, dom \rangle$$

is a prefix of a possible countermodel for φ.

To complete the proof of the theorem, we need:

1. for all atomic formulas α and for all $k \in K$, if $\alpha \in \mathit{Treq}(k)$ then $af(k, \alpha)$. This is true by definition of af.

2. for all $k \in K$, $\mathit{Treq}(k) \cap \mathit{Freq}(k) = \emptyset$. This is true because S was not closed.

3. for all open $k \in K$, if we assume k forces all formulas in $\mathit{Treq}(k)$ and does not force any formula in $\mathit{Freq}(k)$, then φ is not forced at k_0.

   This is the only nontrivial requirement. Given the assumptions, we will actually prove the stronger statement that for all $k \in K$, k forces all formulas in $\mathit{Treq}(k)$ and no formulas in $\mathit{Freq}(k)$. Since $\varphi \in \mathit{FSet}(k_0)$, this suffices to complete the theorem.

   If k is open, then we have k forces all formulas in $\mathit{Treq}(k)$ and no formulas in $\mathit{Freq}(k)$ as a hypothesis, and we are done. So suppose k is not open. Let α be a formula in $\mathit{Treq}(k)$ or $\mathit{Freq}(k)$. The proof that α is forced is by induction on the complexity of α. If α is atomic, it is forced by definition if it is $Treq(k)$, and not forced if it is in $Freq(k)$ because $Treq(k) \cap Freq(k)$ is empty. If α is not atomic, then since k is not open, lemma 2 applies to S and k if k is treated as a label. The lemma, when interpreted in the context of the model, reduces the requirement that α is forced or not at k to a statement about smaller formulas being forced or not at k, and by the inductive hypothesis we are done.

□

Although it is not a full completeness theorem, it is not vacuous. Because each rule is applied only if the information it adds is not already present, the information in the possible countermodel increases with increasing n.

We also have the following corollary.

Theorem 4 *If evaluation of $Develop(\langle\emptyset, \ell_0, F, \varphi\rangle)$ terminates after n steps, then the resulting finite tree either describes a tableau proof or contains a countermodel for φ.*

Proof: If evaluation of $Develop(\langle\emptyset, \ell_0, F, \varphi\rangle)$ terminates in n steps, apply the theorem with this n. If all branches are closed, then there is a tableau proof. If not, then there is a finite prefix of a countermodel for φ. Call this prefix $\langle K, \leq, af, dom\rangle$, as in the theorem. Since the evaluation has terminated, there are no open states in this countermodel. Hence, the assumptions of the third condition in the definition of a finite prefix of a countermodel are vacuous, and we have k_0 does not force φ. □

Unfortunately, we cannot prove that if a finite countermodel exists, the evaluation of *Develop* will terminate. A simple counterexample is the tableau for $F\neg\forall x\exists y A(x,y)$. The actual countermodel consists of two states, 0 and 1, with $0 \leq 1$ and with

$$\begin{aligned} af(0) &= \emptyset \\ dom(0) &= \emptyset \\ af(1) &= \{A(a,b) | a, b \in C\} \\ dom(1) &= C \end{aligned}$$

where C is the set of all constants in the language of the model. The tableau development of this model adds only one constant at a time to the domain of the second state, and thus never terminates. This is a problem with tableau systems for predicate logic in general, including classical logic [13]. It would be interesting to explore the possibility of adding "infinitary rules" to the tableau system in order to try to account for these cases. In this context, however, this would raise problems with the rules for quantified formulas, which depend on quantification over all elements of the domain at a state in order to determine their applicability.

3 Alternative Theorem I: Intuitionistic Models

The constructive theorem in the previous section is limited by the difficulty in reasoning about the behaviour of an infinite process. All structures are finite and the "prefix of possible countermodel" restricts us to finite approximations of models. If we can use some classical reasoning, however, we can prove stronger but still computational theorems.

The work in this section was inspired by the work of intuitionists towards an (intuitionistically) acceptable semantics for their logic. The model theories developed by Veldman and de Swart [15, 4] demonstrate a point of divergence between some forms of intuitionism and the computational approach.

In the completeness proofs for these model theories, the fan theorem is used to construct a proof from an assumption that all branches of the tableau eventually close. The fan theorem is actually an axiom adopted by some intuitionists for reasoning about properties of infinite trees. Its justification depends critically on the assumption that the sequences representing branches in the trees can be generated arbitrarily; in particular, it conflicts with the idea that branches must be recursively computable. In the realm of computer science, however, recursive functions are widely accepted to represent the computable functions; this principle is known as Church's thesis. Since Church's thesis is incompatible with the fan theorem, the fan theorem is somewhat dubious in the context of computability.

The formulation of the statement and proof of theorem 3 were chosen so that they did not require any noncomputational reasoning, including the fan theorem. The motivation for this choice was computational; we wished to describe the tableau procedure. Thus, we would like that any stronger theorems using classical logic still have the same computational property.

In this section, we present a theorem using a model theory based on that of Veldman and de Swart. The theorem itself is a variation of theorem 3, and its proof employs classical reasoning in a limited way to prove a stronger statement about the possible countermodels for φ, without modifying the computational content of the proof.

The aspect of the model theories defined by Veldman and de Swart which we adopt is an extension of the definition of forcing. This extension allows states which force $\perp$. The requirement for forcing a negated formula $\neg\varphi$ at a state s then becomes "for all states $s' \geq s$, if s' forces φ then s' forces $\perp$. Note that a state forcing $\neg\varphi$ under the usual definition of Kripke model still forces $\neg\varphi$, so we have not modified the properties of regular models in any way. We say a Kripke model K *explodes* if any state in the model forces $\perp$. A Kripke model K is a countermodel for φ if K forces φ implies K explodes.

Given this extended definition of forcing, we can define the notion of a prefix of a countermodel with respect these models.

Definition 3.1 *A prefix of a countermodel for φ is a finite Kripke model*

$$\langle K, \leq, af, dom \rangle$$

with a minimum state $k_0 \in K$, a subset of K designated as open *states, and a pair of functions Treq and Freq from K to sets of formulas, satisfying*

1. *for all atomic formulas α and for all $k \in K$, if $\alpha \in Treq(k)$ then $af(k, \alpha)$*

2. *for all $k \in K$, $Treq(k) \cap Freq(k) = \emptyset$*

3. *for all Kripke models K' containing K and satisfying, for all open $k \in K$,*

   (a) k forces all formulas in $Treq(k)$ in K'

   (b) k forces any formula in $Freq(k)$ in K' implies k forces $\perp$

   then if φ is forced at k_0, K' explodes.

With this definition, we can state a not-completely-constructive but computational theorem:

Theorem 5

$$\begin{array}{l} \forall\varphi.\forall n : N \\ \quad \mathit{let}\ P = \mathit{unwind}(n, \mathit{Develop}(\{\langle\emptyset, \ell_0, F, \varphi\rangle\}))\ \mathit{in} \\ \qquad P\ \mathit{is\ a\ closed\ tableau\ proof\ of}\ \varphi\ \mathit{or} \\ \quad P\ \mathit{describes\ a\ prefix\ of\ a\ countermodel\ for}\ \{\varphi\} \end{array}$$

This theorem has the same computational content as the completely constructive one; however, its proof uses classical reasoning if the nth development is not a closed proof.

Proof: By theorem 3, we have either P is a closed tableau proof of φ or P contains a prefix of a possible countermodel for φ. We need only show that if P describes a prefix of a possible countermodel for φ, then the same description is actually a description of a prefix of a countermodel for φ as defined in definition 3.1.

The proofs for the first two cases of definition 3.1 are as in theorem 3. For the third case of definition 3.1, fix a K' containing K and satisfying the conditions given. Let k range over states open in K (hence $k \in K'$). Using classical reasoning on the second condition "k forces any formula in $\mathit{Freq}(k)$ implies k forces $\perp$," we can say that either k does not force any formula in $\mathit{Freq}(k)$ or k forces $\perp$. By the proof of theorem 3, we have that if, for all k, k does not force (in K') any formula in $\mathit{Freq}(k)$, then φ is not forced at k_0. Otherwise, if there is a k such that k forces $\perp$, then K' explodes. □

The proof of this theorem uses classical reasoning to conclude that, given a particular open branch P of the partial proof tree, either it generates a model which does not force φ, or eventually it closes. In this latter case, the "exploding model" interpretation means that any closed branch generates a model which forces $\perp$ at some state. Hence, such a model explodes. In either case, P describes a prefix of a countermodel. This means that a prefix of a potential countermodel defined in the previous section is always a prefix of a countermodel in this setting. Note that this means we will sometimes have a "countermodel" for provable formulas; however, we will only have exploding countermodels for such formulas. The undecidability arises from the fact that we cannot determine whether or not all extensions of a given prefix of a model explode.

The computational content of this theorem is still a bounded tree search procedure. Since details were not given of an evaluation order for terms in the lazy language in which *Develop* is defined, we do not know whether the branches are evaluated depth first or breadth first. However, since the same finite tableau results either way as long as the choice of rules to apply at a given stage is deterministic, the precise evaluation order is unimportant. No matter what order is chosen, the bound n prevents an unbounded search.

4 Alternative Theorem II: Computational Content of Classical Reasoning

In this section, we present a version of the theorem such that the computational content of the proof could be specified to be an unbounded depth first tableau search. Since the procedure can be non-terminating, this cannot be done constructively. In order to reason about the evaluation of a branch of the tableau, we will need a logical tool which, computationally interpreted, allows us to switch from one branch to another if and when evaluation of one branch terminates. Since termination is undecidable, we need a computational method for reasoning about undecidable propositions. In other words, we need a computational interpretation of classical reasoning.

To begin, we give a brief introduction to the use of the nonlocal control operator *call/cc* as the computational content of a classical axiom. This interpretation is based on ideas of Griffin [7] (who worked with a slightly different control operator) and later developed by Murthy [10, 11]. Using this operator to represent the computational interpretation of a nondeterministic choice, we can extract a depth-first tableau procedure from a completeness proof which uses classical logic.

The nonlocal control operator *call/cc* (found in the Scheme programming language) can be given the type $((\neg A)\rightarrow A)\rightarrow A$, a classical tautology which, when added to an intuitionistic system, extends it to all of classical logic. In this way, *call/cc* extends the "propositions as types" analogy used to explain the computational content of constructive proofs. The control operator, when evaluated, captures the current evaluation context in such a way that it may be restored at a later point in the evaluation by application of an abort operator $\mathcal{A}$. More concisely, these rules can be written

$$\begin{aligned} E[call/cc(\lambda k.t)] &\to_1 E[t[k := \lambda x.\mathcal{A}(E[x])]] \\ E[\mathcal{A}(E'[t])] &\to_1 E'[t] \end{aligned}$$

To see how this works, suppose we have an undecidable predicate A and, in a program computing a result g of type G, we wish to perform a case split on the truth value of A. In other words, we wish to split the computation into two branches, one which produces a result g_A of type G if A is true, and one which produces a result $g_{\neg A}$ of type G if $\neg A$ is true. Computationally, since we cannot decide which of A or $\neg A$ is true, we cannot proceed. However, using *call/cc* we can construct a term which appears to be of type $A \vee \neg A$. When this term is evaluated, the current continuation is saved (so that it may be restarted later), and a new term $t_{\neg A}$ is created which appears to be computational evidence for $\neg A$. Using this, we can produce $g_{\neg A}$ of type G.

Now, if G is the type of an actual data value (as opposed to, say, a function type), and if the term $g_{\neg A}$ is evaluated to provide that data value, then there are two possibilities. One is that the term $g_{\neg A}$ did not use the term $t_{\neg A}$ of type $\neg A$ to produce its result. In this case, the internal structure of the purported term of type $\neg A$ cannot be examined, so cannot have affected the result. The other possibility is that $t_{\neg A}$ is used as computational evidence for $\neg A$. Since $\neg A = A \to \bot$, the only way in which $t_{\neg A}$ can be used is if it is applied to a term of type A to produce a contradiction, in the form of an inhabitant of the empty type $\bot$. In that case, we would like to switch from the branch of the computation based on the assumption $\neg A$ to the one which computes g_A of type G from A, using the argument to $t_{\neg A}$ as the needed evidence for A.

The control operator *call/cc* provides a mechanism for this kind of branch switching. Since it saved the evaluation context when the choice between A and $\neg A$ was made, we can apply the continuation to the new evidence for A to return to that point in the evaluation and continue along that computation branch.

4.1 Depth-first tableau procedure

Returning to the tableau procedure for intuitionistic predicate logic, we can see that a control operator will provide the necessary means of switching from one tableau branch to another when the first branch closes, without the need for explicit backtracking. In fact, control operators were introduced to functional languages for reasons of efficiency. The use of continuations can avoid the need to unwind deep layers of recursive calls when a result is found. This is especially useful in the context of the inductive types used to define *Develop*; we need not explicitly keep track of previous stages in the tableau in order to return to them and evaluate another branch.

More concretely, we can classically prove a theorem of the form

Theorem 6

$$\begin{array}{l} \forall\varphi.\exists P: \ PPT. \\ \qquad P \text{ evaluates to a proof of } \varphi \text{ or} \\ \qquad P \text{ evaluates to a countermodel for } \varphi \end{array}$$

Proof outline: Without delving deeply into the logical structure of the proof, we give an argument that there is a classical proof of this theorem which corresponds computationally to a depth-first search of the tableau using control operators. The control operator is used to "decide" the undecidable disjunction "either there is not a proof, or there is a proof". Under the supposition that there is not a proof, the development describes the construction of a countermodel. Since the countermodel may be infinite, the construction actually describes a process which, in the limit, produces a countermodel. However, should a contradiction arise on one branch, that branch of the development actually represents a proof. In this case, another branch is explored, or, if no other branch is left, we have a proof of φ.

The proof involves reasoning about evaluation of the *Develop* function defined earlier. The classical reasoning is, in effect, wrapped around the constructive *Develop* function.

For this proof, the control operator is really needed only in two places. First, at the outset, so that we can claim immediately that either there is no proof or there is a proof. The result of evaluating the initial term should be a claim (which is really a conjecture at this point) that the process of evaluating $Develop(\{F\varphi\})$ constructs a countermodel in its limit, and the computational content of this proof should consist of this evaluation.

Second, at any point where the tableau branches, we need to repeat the reasoning for the two branches. For the first branch, either there is no proof or there is a proof. As at the initial step, the result of this decision should be a claim that evaluating the first branch produces a countermodel, along with the creation of a continuation at this decision. If it turns out a contradiction is found, then there is a proof for the first branch. At this point, the continuation returns control to this point and the second branch is conjectured to produce a countermodel.

We want the *Develop* tree to be evaluated eagerly, since we do not want a result which contains possible continuation applications. This means that if the resulting program were actually evaluated, there would be three possible outcomes:

1. The development halts and produces a proof.
2. The development halts and produces a finite countermodel.
3. The development does not halt, and in the limit produces a countermodel.

Unlike the constructive theorems which were restricted to reasoning about finite prefixes, we must actually prove in this case that, in the limit, the procedure described produces a countermodel. A (classical) proof of this for a similar procedure is found in [6]. Essentially, we can use the tableau procedure to define a sequence of approximations to a countermodel, then (classically) take the union of these approximations as the completed countermodel. (It is more difficult to reason intuitionistically about the result of an infinite process.)

5 Conclusions

We have shown how the tableau procedure for intuitionistic predicate logic can be interpreted as the computational content of a theorem about the logic. This was done in three ways. First, we gave a completely constructive metatheorem which had a bounded tableau search as its computational content. Second, we showed how earlier work on intuitionistic model theory leads to a stronger statement of this metatheorem. Finally, we showed that a computational interpretation of classical logic allows the extraction of unbounded tableau search as the computational content of a metatheorem.

In related work [3, 1], it was shown how a theorem proving environment which was strong enough to describe its own proofs could extend its own store of tactics by using the computational content of metatheorems along with a reflection rule. The example given in [1] is developed from [14]. It is hoped that the ideas in this paper will contribute to an extension of that earlier result, in particular, allowing the limited use of classical reasoning in a computational context.

References

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

[2] R.C. Backhouse. Constructive type theory–an introduction. In M. Broy, editor, *Constructive Methods in Computer Science, NATO ASI Series, Vol. F55: Computer & System Sciences*, pages 6–92. Springer-Verlag, New York, 1989.

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

[4] H. de Swart. Another intuitionistic completeness proof. *Journal of Symbolic Logic*, 41(3):644–662, September 1976.

[5] M. Dummett. *Elements of Intuitionism.* Oxford Logic Series. Clarendon Press, 1977.

[6] Melvin Fitting. *Intuitionistic Logic, Model Theory, and Forcing.* North-Holland, 1969.

[7] T. Griffin. A formulas-as-types notion of control. In *Proc. of the Seventeenth Annual Symp. on Principles of Programming Languages*, pages 47–58, 1990.

[8] N. Mendler, R. Constable, and P. Panangaden. Infinite Objects in Type Theory. In *Proc. of First Symp. on Logic in Comp. Sci.*, pages 249–257. IEEE, June 1986. (Cornell TR 86-743).

[9] P.F. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, Ithaca, NY, 1988.

[10] C. Murthy. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, Dept. of Computer Science, 1990. (TR 89-1151).

[11] C. Murthy. An evaluation semantics for classical proofs. In *Proc. of Sixth Symp. on Logic in Comp. Sci.*, pages 96–109. IEEE, Amsterdam, The Netherlands, 1991.

[12] A. M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124(2):195–219, 1994.

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

[14] J. 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. Available as Technical Report MPI-I-93-213 Max-Planck-Institut für Informatik, Saarbrücken, Germany.

[15] Wim Veldman. An intuitionistic completeness theorem for intuitionistic predicate logic. *Journal of Symbolic Logic*, 41(1):159–166, March 1976.