A Tableau-Based Theorem Proving Method for Intuitionistic Logic

Oliver Bittel
Fachhochschule Konstanz*

Abstract

A new tableau-based calculus for first-order intuitionistic logic is proposed. The calculus is obtained from the tableau calculus for classical logic by extending its rules by λ-terms. λ-terms are seen as compact representation of natural deduction proofs. The benefits from that approach are two-fold.

First, proof search methods known for classical logic can be adopted: Run-time-Skolemization and unification. In contrast to the conventional tableau, sequent, or natural deduction calculus for intuitionistic logic we get no problem with order dependance of rule applications in our calculus. Therefore, backtracking is only necessary in the selection of unifiers.

Second, as a by-product λ-terms are synthesiszed for free during proof search. λ-terms are important, when intuitionistic logic is applied in a program synthesis framework.

We implemented the calculus in Prolog. A strategy which is similar to model elimination has been built in. Several formulas (including program synthesis problems) have been proven automatically.

1 Introduction

In many papers, the intuitionistic logic has been proposed as a logic for program synthesis (see e.g. [Mar82], [Con86], [BSII90]). A formula of the form $\forall x.\exists y.P(x,y)$ specifies a program, which takes an input x and yields an output y such that the condition $P(x,y)$ holds. A constructive (intuitionistic) proof of such a specification can be seen as a program satisfying this specification. E.g. the following formula specifies the integer square root function:

$$\forall n \in Nat.\ \exists r \in Nat.\ r^2 \leq n < (r+1)^2$$

A proof of this formula consists of a function f mapping a natural number n to a pair $\langle r, M\rangle$, whereby r is the integer square root of n and M is a proof for that. The main characteristic of this program synthesis paradigm is that proofs are performed in the natural deduction calculus. Instead of taking a natural deduction proof as program, it is more appropriate to use a more compact representation, usually a λ-term. There is a direct correspondence between proof rules and program constructs: case analysis in proofs and λ-terms, inductions and recursions, and finally lemmas and subroutines.

*Author's address: Fachhochschule Konstanz, Fachbereich Informatik, Postfach 10 05 43, D-78405 Konstanz, Germany, E-mail: bittel@fh-konstanz.de

In order to solve realistic problems, it is important to automate at least the trivial parts of a proof. For that, we need an automatic theorem prover for the intuitionistic logic which is efficient and yields λ-terms if formulas are valid.

Our approach is based on a signed version of a tableau calculus for classical logic. Its rules are extended by λ-terms. Two kind of formulas may occur in a tableau: $+M : \alpha$ means that (the λ-term) M is a proof for α, and $-M : \alpha$ means that M is not a proof for α. Most of the extended tableau rules are straightforward translations of the natural deduction rules with λ-terms. E.g., if $+M : \alpha \wedge \beta$ occurs in a branch, then $+\mathbf{fst}(M) : \alpha$ and $+\mathbf{snd}(M) : \beta$ may added to the branch;[1] if $-\langle M, N\rangle : \alpha \wedge \beta$ occurs in a branch, then a branching may be introduced with $-M : \alpha$ as new left node and $-N : \beta$ as new right node. However, the translation is not straightforward for the disjunction rules.

If we have $+M : \alpha \vee \beta$, then M represents either a proof for α or one for β. But we have no linguistic means (λ-term constructs) to express this. For that, we introduce some new λ-term constructs: *partially defined λ-terms* and *implicit case analysis*. In the case of $+M : \alpha \vee \beta$, we get partially defined λ-terms as proofs for α and for β. Implicit case analysis is a list of λ-terms and is usefull for formulas, whose proofs consist of several cases. There is some simalarity between guarded commands and implicit case analysis.

Our calculus is presented in two versions. The ground version is helpful for making soundness and completeness proofs easier. If a formula α has to be proven in that calculus, one must guess a λ-term M and search a closed tableau for $-M : \alpha$.[2] The second version is a lifted version of the ground calculus. Guessing λ-terms and individual terms are not necessary any longer. Meta-variables are used which are instantiated by unification.[3]

One of the main drawbacks of the conventional tableau, sequent or natural deduction calculus is the strong order dependance of rule applications. The problem is well-known for quantifier rules in classical logic. There, the

[1] In intuitionistic logic a proof for $\alpha \wedge \beta$ is a pair consisting of a proof for α and a proof for β. To get the components of the pair the projections fst and snd are used.

[2] A closed tableau for $-M : \alpha$ means, that the assumption "M is not a proof for α" leads to a contradiction. Thus, M is a proof for α.

[3] The usual term unification is meant. We need no unification modolo β-reduction or anything else.

problem can be solved by Skolemization, which is not possible for intuitionistic logic. Moreover, in intuitionistic logic the problem already exists on the propositional level as the following example shows.[4] If we want to prove $a \vee b, a \rightarrow c, b \rightarrow d \vdash c \vee d$, we have several possibilities to apply a rule. Decomposing the right-hand side yields the two subgoals $a \vee b, a \rightarrow c, b \rightarrow d \vdash c$ and $a \vee b, a \rightarrow c, b \rightarrow d \vdash d$ and leads to a dead end as it can be easily seen. The only way to succeed is to decompose $a \vee b$ first. Note, that decomposing one of the implications on the left-hand side would also lead to a dead end. In the sequent calculus for classical logic in contrast, several formulas on the right-hand side are allowed. A decomposition of $c \vee d$ leads to two new fomulas on the right-hand side: $a \vee b, a \rightarrow c, b \rightarrow d \vdash c, d$. So, we can not choose wrong. That is the reason, why there is no dead ends problem in classical propositional logic. In our calculus we have the same situation: several negativly signed formulas are allowed to occur in a tableau branch.[5] A decomposition of $-c \vee d$ extends the branch by two new nodes $-c$ and $-d$. Therefore, in our calculus there is no bracktracking necessary for that example.

In literature, there are mainly two approaches for automatic theorem proving in intuitionistic logic:

1. Systems such as NUPRL [Con86] and OYSTER [BSH90] are based on the sequent or the natural deduction calculus. They provide tactics and tacticals for conducting backward proofs. The calculi are very much appropriate for synthesizing λ-terms. But as discussed above, the main problem is the strong order dependence of the rules. In [Sha92] a kind of Herbrandization is used to record the impermutabilities of some of the intuitionistic sequent rules. The amount of backtracking could be reduced. But the problem with the disjunction (see example above) still exists.

2. In the last years, several proof methods for modal and intuitionistic logics have been proposed.[6] Among them are matrix-based proof methods [Wal88] and resolution calculi [Ohl88]. Both approaches are based on Kripke-style semantics and refutation theorem proving. The main idea is that the formula to be proven is translated to a matrix [Wal88] or a set of clauses [Ohl88] such that it can be handled like in the classical logic. The Kripke-style semantics is taken into account by introducing so-called world paths in predicates and functions and to use a specific unification procedure for the world paths. These proof procedures work well if only the question of validity of a formula is interesting. However, in order to integrate an automatic theorem prover in the program synthesis approach considered here it is mandatory to get a λ-term (or proof in the natural deduction calculus) in case a formula is proven valid. At this point it should be mentioned that in classical logic there are some methods which allow the transformation from matrix and resolution proofs to natural deduction proofs (see [Wos90] as overview). However, these translations generate non-constructive proofs in some cases.

[4] A sequent style formulation is used. The same problem also occurs in the tableau and the natural deduction calculus.

[5] There is also no branch modification rule as it is the case in the conventional tableau calculus for intuitionistic logic [Fit83]. The branch modification rule deletes all negativly signed formulas in a branch and must be applied additionally by some rule applications.

[6] There is a strong correspondence between the modal logic S4 and the intuitionistic logic [Fit83].

2 First-Order Intuitionistic Logic

In this section, a first-order language, λ-terms and the calculus of natural deduction with λ-terms are defined. This calculus can be seen as a subset of Martin-Löf's intuitionistic type theory [Mar82]. The definition is very similar to the one found in [Coq90].

Let $\mathcal{V}$ be a set of variables. An *individual term* is either a variable or of the form $f(t_1, ..., t_n)$, where f is a function symbol with arity $n \geq 0$ and t_i are individual terms. The set of all individual terms are named by $\mathcal{T}$. *Fomulas* are defined in the usual way by using relation symbols, individual terms and the logical connectives $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), $\bot$ (falsity), quantifier symbols $\forall$ and $\exists$. $\neg\alpha$ is defined as $\alpha \rightarrow \bot$. For a formula α, a variable x and a term t, $\alpha[x/t]$ denotes the result of substituting each free occurrence of x in α by t.[7] $FV(\alpha)$ is defined to be the set of all free variables in α.

The set Λ of all *λ-terms* is defined by the following inductive definition. Let $x \in \mathcal{V}$, $t \in \mathcal{T}$ and M, N, N' λ-terms. Then the following expressions are also λ-terms.

x	*variable*
t	*individual term*
MN	*application*
$\lambda x.M$	*abstraction*
$\langle M, N \rangle$	*pairing*
$\mathbf{fst}(M)$, $\mathbf{snd}(M)$	*projections*
$\mathbf{split}(M, N)$	*pair application*
$\mathbf{inl}(M)$, $\mathbf{inr}(M)$	*injections*
$\mathbf{when}(M, N, N')$	*case analysis*
$\mathbf{absurd}(M)$	

For a λ-term M, a variable x and a λ-term N, $M[x/N]$ denotes the result of substituting each free occurrence of x in M by N.[7] $FV(M)$ means the set of all free variables in M. There are also some conversion rules[8]

$(\lambda x.M)N \rhd M[x/N]$
$\mathbf{when}(\mathbf{inl}(N), \lambda x.M, O) \rhd M[x/N]$
$\mathbf{when}(\mathbf{inr}(N), O, \lambda x.M) \rhd M[x/N]$
$\mathbf{split}(\langle N_1, N_2 \rangle, \lambda x_1.\lambda x_2.M) \rhd M[x_1/N_1][x_2/N_2]$

A *context* is a finite set $\{x_1 : \alpha_1, ..., x_n : \alpha_n\}$, where $\alpha_1, ..., \alpha_n$ are formulas and $x_1, ..., x_n$ are pairwise distinct variables which must not occur in α_1,..., α_n. An expression like $M : \alpha$ is also called a *proof relation*.[9] For any

[7] We assume that renaming of bounded variables is done in order to avoid name clashes.

[8] We have omitted the so-called commuting conversions. For a complete definition see e.g. [Gir89], [Bit91].

[9] I.e. M is proof for α. It is also known as type relation, i.e. M is of type α. There is an isomorphism between types and formulas (see e.g. [How80]).

context Σ, the set of all free variables occuring in Σ is denoted by $FV(\Sigma)$.

For any context Σ, λ-term M and formula α, we define the relation $\Sigma \vdash M : \alpha$ by the rules shown in table 1. If $\Sigma \vdash M : \alpha$ holds we also say that α is *(intuitionistically) valid under the context Σ and the λ-term M is a proof for that.* If $\emptyset \vdash M : \alpha$, then α is *(intuitionistically) valid.*

Remark: If the λ-terms in the rules are omitted we exactly get the usual natural deduction calculus for intuitionistic logic. However, the calculus with λ-terms offers the possibilty not only to check the validity of a formula α but also to construct a λ-term which can be seen as the desired program satisfying α.

3 The Ground Calculus

In this section a tableau-based calculus is presented which allows to prove relations like $\Sigma \vdash M : \alpha$. During proof search λ-terms and individual terms have to be guessed. Therefore, we call it ground calculus. It is obtained from the tableau calculus for classical logic by extending its rules by λ-terms. For that, we introduce projections for existentially bounded formulas and some new λ-term constructs which we call implicit case analysis and inverses to the injections. The inverses lead to partially defined λ-terms. With the help of a transformation procedure, projections and implicit case analysis (together with the inverses) can be replaced by pair-application (**split**-construct) and explicit case analysis (**when**-construct), respectively.

3.1 Informal Description of the Calculus

Tableaux are trees labeled with signed proof relations of the form $+M : \alpha$ and $-M : \alpha$. The intuitive meaning is the following:

occurs in a tableau branch π	meaning
$+M : \alpha$	$\Sigma_\pi \vdash M : \alpha$ [10]
$-M : \alpha$	$\Sigma_\pi \not\vdash M : \alpha$

In order to check $\{x_1 : \gamma_1, ..., x_n : \gamma_n\} \vdash M : \alpha$, we start with the following intitial tableau which consists of one branch with $n+1$ nodes:

$$\begin{array}{l} +x_1 : \gamma_1 \\ \quad\vdots \\ +x_n : \gamma_n \\ -M : \alpha \end{array}$$

To this initial tableau several tableau rules (shown in table 2) are applied until a closed tableau is reached, i.e. a tableau where each branch contains a complementary pair $+N : \gamma$ and $-N : \gamma$.

Most of the tableau rules are straightforward translations of the natural deduction rules with λ-terms. We want to motivate some of them. If $\lambda x.M$ is not a proof for $\alpha \to \beta$ (i.e. $-\lambda x.M : \alpha \to \beta$) then by the $(- \to)$-rule, M can not be a proof for β (i.e. $-M : \beta$) under the assumption that x is a proof for α (i.e. $+x : \alpha$). If we know M to be a proof for $\alpha \to \beta$ (i.e. $+M : \alpha \to \beta$) we may introduce a branching using the $(+ \to)$-rule. For an arbitrary λ-term N, we may assume that either N is not a proof for α (i.e. $-N : \alpha$; left branch) or N is a proof for α. In the latter case, we infer that MN is a proof for β (i.e. $+MN : \beta$; right branch).

The most complicated rules are the disjunction rules and the rules for the existential quantifier. This is not so much surprising, since disjunction and existence are known to be the two most typically intuitionistic connectors [Gir89].

If we have a proof M for $\alpha \vee \beta$, then M represents either a proof for α or one for β. But we have no linguistic means (λ-term constructs) to express this. For that, we introduce two new constructors **l** and **r**. $\mathbf{l}(M)$ is only defined if M represents a proof for the left part of a disjunction. Thus, $\mathbf{l}(\mathbf{inl}(N))$ is defined whereas $\mathbf{l}(\mathbf{inr}(N))$ is not. This property is expressed by the following conversion rules:

$$\begin{array}{ll} \mathbf{l}(\mathbf{inl}(N)) & \rhd N \\ \mathbf{l}(\mathbf{inr}(N)) & \rhd \uparrow \text{ (undefined)} \end{array}$$

$\mathbf{r}(M)$ can be considered analogously. Thus, $\mathbf{l}(M)$ and $\mathbf{r}(M)$ are partially defined λ-terms. In some sense, **l** and **r** is inverse to **inl** and **inr**, respectively. With the new constructors, the formulation of the $(+\vee)$-rule becomes straightforward.

Next, we introduce the implicit case analysis, which is a list of partially or totally defined λ-terms:

$$[\![M_1, M_2, ..., M_n]\!]$$

Each M_i represents one case. If one of the cases is totally defined, the implicit case analysis can be reduced to that case:

$$[\![M_1, M_2, ..., M_n]\!] \rhd M_i, \text{ if not } M_i \rhd \uparrow$$

The implicit case analysis is very similar to a guarded command, where the conditions are left implicit. The usual (explicit) case analysis

$$\mathbf{when}(M, \lambda p_1.N_1, \lambda p_2.N_2)$$

can be expressed by an implicit case analysis

$$[\![N_1[p_1/\mathbf{l}(M)], N_2[p_2/\mathbf{r}(M)]]\!]$$

The $(-\vee)$-rule can now be read in the following way: if $[\![\mathbf{inl}(M), \mathbf{inr}(N)]\!]$ is not a proof for $\alpha \vee \beta$, then both M is not a proof for α and N is not a proof for β.

Since a proof for an arbitrary formula may be a case analysis, the rule $(-[\![\,]\!])$ is also needed.

As we will see in the following section λ-terms with implicit case analysis can easily be transformed to those with the usual explicit case analysis.

At last, the $(+\exists)$-rule shall be considered. If M is a proof for $\exists x.\alpha$, then M represents a pair, consisting of

[10] Σ_π is the context that corresponds to the branch π, i.e. $\Sigma_\pi = \{x : \alpha \mid +x{:}\alpha \in \pi \text{ and } x \in \mathcal{V}\}$.

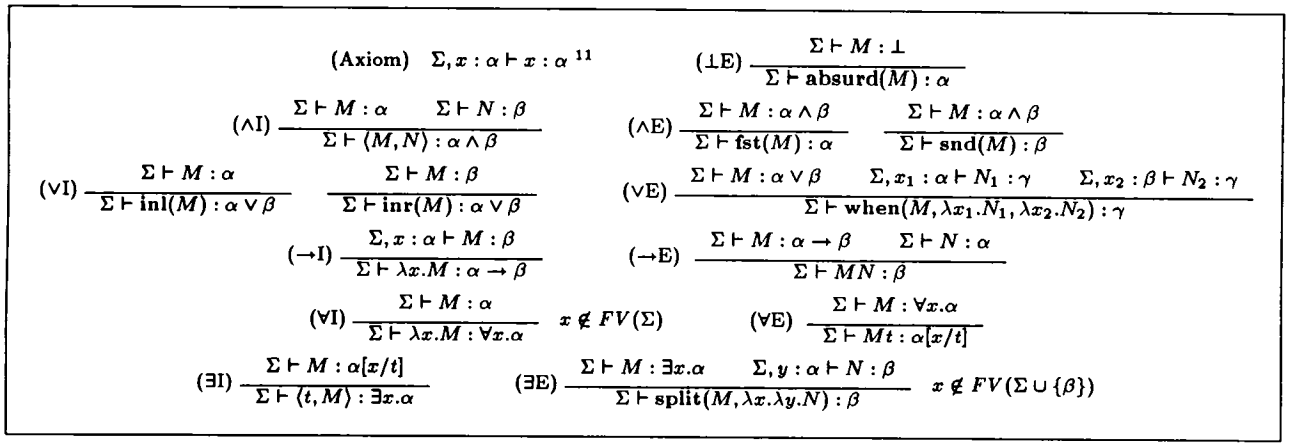

Table 1: Natural deduction calculus with λ-terms

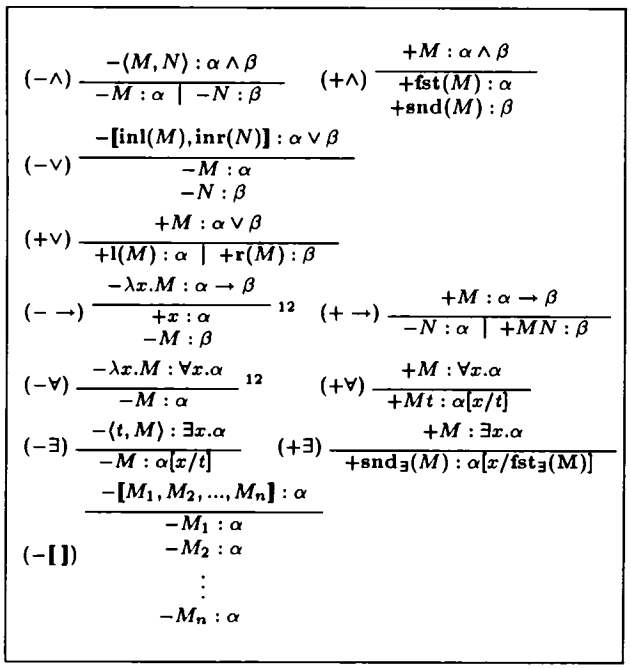

Table 2: Tableau calculus with λ-terms

a term t and a proof for $\alpha[x/t]$. We use the projection functions $\mathbf{fst}_\exists$ and $\mathbf{snd}_\exists$ to access to the pair components. Thus, the formulation of the $(+\exists)$-rule becomes straightforward. The index $\exists$ in the projection functions is used to distinguish them from the projections used for conjunctions. Now, λ-terms of the form $\mathbf{fst}_\exists(M)$ are also allowed to occur in formulas. They denote in some way constants. These terms together with individual terms will be called extended individual terms. t in the rules $(-\exists)$ and $(+\forall)$ is restricted to be an extended individual term.

3.2 Formal Definition of the Calculus

Definition 1 (Extended λ-terms) *The set Λ_E of all extended λ-terms is defined inductively as follows. Let f be a function symbol with arity $n \geq 0$, $x \in \mathcal{V}$ and $M, N, M_1, ..., M_n$ extended λ-terms. Then the following expressions are also extended λ-terms:*

x	variable
$f(M_1, ..., M_n)$	individual term
MN	application
$\lambda x.M$	abstraction
$\langle M, N\rangle$	pairing
$\mathbf{fst}(M)$, $\mathbf{snd}(M)$	projections for proofs of $\wedge$-formulas
$\mathbf{fst}_\exists(M)$, $\mathbf{snd}_\exists(M)$	projections for proofs of $\exists$-formulas
$\mathbf{inl}(M)$, $\mathbf{inr}(M)$	injections
$\mathbf{l}(M)$, $\mathbf{r}(M)$	inverses to the injections
$[\![M_1, ..., M_n]\!], n \geq 0$	implicit case analysis
$\mathbf{absurd}(M)$	

Definition 2 (Extended individual terms) *The set $\mathcal{T}_E$ of all extended individual terms is defined inductively as follows. Let $x \in \mathcal{V}$, $M \in \Lambda_E$, f a function symbol with arity $n \geq 0$ and $t_1, ..., t_n$ extended individual terms. Then $x, f(t_1, ..., t_n), \mathbf{fst}_\exists(M)$ are extended individual terms, too.*

Definition 3 (Extended formula) *The set of all extended formulas is defined inductively. Each formula is also an extended formula. If α is an extended formula and $t \in \mathcal{T}_E$, then $\alpha[x/t]$ is an extended formula.*

Definition 4 (Tableau) *A tableau is a binary tree which is labeled with signed proof relations of the form $+M : \alpha$ and $-M : \alpha$ where $M \in \Lambda_E$ and α is an extended formula. Let Γ be a set of signed proof relations. An initial tableau for Γ consists of exactly one branch π such that Γ is the set of all labels in π. If no confusion is possible, we identify Γ with some initial tableau for Γ.*

[11] We write $\Sigma, x : \alpha$ instead of $\Sigma \cup \{x : \alpha\}$.

[12] x must not occur free in the branch where the rule is applied.

A branch π is called closed if both $+M : \alpha$ and $-M : \alpha$ or if both $+M : \bot$ and $-\mathbf{absurd}(M) : \alpha$ (complementary pairs) occur in π for some $M \in \Lambda_E$ and some extended formula α. A tableau is closed if all its branches are closed.

Definition 5 (Tableau extension) *Let T, T' be tableaux and r one of the tableau rules. We define the relation $T \xrightarrow{r} T'$ (T is extended to T' by rule r). In general, r is either a branching or a non-branching rule of the following form:*

$$(1)\ \frac{A}{A_1 \mid A_2} \qquad (2)\ \begin{array}{c} A \\ \hline A_1 \\ \vdots \\ A_n \end{array} \qquad n \geq 1$$

$T \xrightarrow{r} T'$ holds if there exists some branch π in T such that A (the upper part of r) occurs in π and T' is obtained from T by extending π in the following way:

- *If r is a branching rule of the form (1), then π is extended by a branching with one left and one right node labeled with A_1 and A_2, respectively.*
- *If r is a non-branching rule of the form (2), then π is extended by n nodes which are labeled with $A_1, \ldots, A_n$.*

The proviso of the rules $(- \rightarrow)$ and $(-\forall)$ must be taken into account. We write $T \xrightarrow{gc} T'$ if there exists a rule r such that $T \xrightarrow{r} T'$. $\xrightarrow{gc*}$ is defined to be the reflexive and transitive closure of $\xrightarrow{gc}$.

Example: Let $\Sigma = \{y_0 : a \wedge b \rightarrow g, y_1 : c \rightarrow g, y_2 : b \vee c\}$ be a context. The figure 1 shows a closed tableau T where

$$+\Sigma \cup \{-\lambda x.[\![y_1 \mathbf{r}(y_2), y_0\langle x, \mathbf{l}(y_2)\rangle]\!] : a \rightarrow g\} \xrightarrow{gc*} T$$ [13]

By the soundness result (see below), we also get:

$$\Sigma \vdash a \rightarrow g$$

The nodes (5) and (6) in T result from node (4) by application of $(- \rightarrow)$. Note, that the pairs of nodes (9), (10) and (15), (16) result from the application of $(+ \rightarrow)$. For that, the λ-terms in node (9) and (15) were guessed. □

Comparison between tableau calculus and natural deduction calculus: Suppose we want to find a proof in the natural deduction calculus for $\Sigma \vdash a \rightarrow g$.[14] For that, it is appropriate to start with the goal sequent $\Sigma \vdash a \rightarrow g$ and apply rules in a backward manner. Using the $(\rightarrow$I) rule we obtain the subgoal $\Sigma, a \vdash g$. Now, there are several possibilities to continue. If we decide to decompose $a \wedge b \rightarrow g \in \Sigma$ with the $(\rightarrow$E)-rule, we get the two subgoals

$$\Sigma, a \vdash a \wedge b \rightarrow g \text{ and } \Sigma, a \vdash a \wedge b.$$

The first sequent is trivially true but the second sequent can not be derived. Thus, we went into a dead end. The right way would be to carry out first a case analysis using the assumption $b \vee c$ and the $(\vee$E)-rule. It is clear that this property of the natural deduction calculus becomes worse if Σ increases in size or nested case analysis are required.

On the contrary, the tableau calculus is free from dead end. It does not matter in which order rules are applied. As we can see in the example, first $+a \wedge b \rightarrow g$ was decomposed before we introduce a case analysis by decomposing $+b \vee c$. The tableau would also be closed if the rule order would be changed. The order of the rule applications in the example comes up if a strategy is used which requires branches to be closed as soon as possible.[15] E.g. after node (8), the decomposition of $+a \wedge b \rightarrow g$ was chosen, because it leads to two closed branches (see node (11) and (10)). □

In order to define the notion of validity some global constraints on tableaux are necessary. As the example above shows, the tableau T would remain closed if the implicit case analysis in node (4) and (6) would be extented by a further arbitrary λ-term N.[16] Since we want to transform λ-terms of Λ_E to usual λ-terms of Λ, such an additional arbitrary λ-term N would have a disturbing effect. Thus, we require each negatively signed proof relation to be participated either in a rule application or in a complementary pair. Another problem comes from the fact that in the $(+\forall)$-rule t may contain a λ-term $\mathbf{fst}_\exists(N)$, where N can be a completely senseless term. However, if a node labeled with $+N : \exists x.\alpha$ appears in the tableau, the term $\mathbf{fst}_\exists(N)$ would make sense. We comprise these global constraints in the following minimality condition.

Definition 6 (Minimal tableaux) *A tableau T is called minimal iff two conditions hold:*

(i) For each $-M : \alpha$ occuring in T the following is true:

- *$M = [\![]\!]$ (empty case analysis), or*
- *a tableau rule is applied to $-M : \alpha$, or*
- *there exists some $+M : \alpha$ in T, such that both $-M : \alpha$ and $+M : \alpha$ occur in the same branch (complementary pair), or*
- *there exists some $+N : \bot$ in T, such that $M = \mathbf{absurd}(N)$ and both $-M : \alpha$ and $+N : \bot$ occur in the same branch (complementary pair).*

(ii) For each $\mathbf{fst}_\exists(N)$ occuring in T there exists some formula $\exists x.\alpha$ and some node in T which is labeled with $+N : \exists x.\alpha$.

Definition 7 (gc-Validity) *Let Σ be any context, $M \in \Lambda_E$ and α any formula. We define $\Sigma \vdash_{gc} M : \alpha$ if there exists some closed and minimal tableau T such that $+\Sigma \cup \{-M : \alpha\} \xrightarrow{gc*} T$.*

[13] $+\Sigma = \{+M : \alpha \mid M : \alpha \in \Sigma\}$.

[14] We choose Σ as in the example above. For simplicity, λ-terms are omitted.

[15] Actually, this strategy is taken form [Sch85] and is similar to model elimination.

[16] An extension by N would lead to $[\![y_1 \mathbf{r}(y_2), y_0\langle x, \mathbf{l}(y_2)\rangle, N]\!]$.

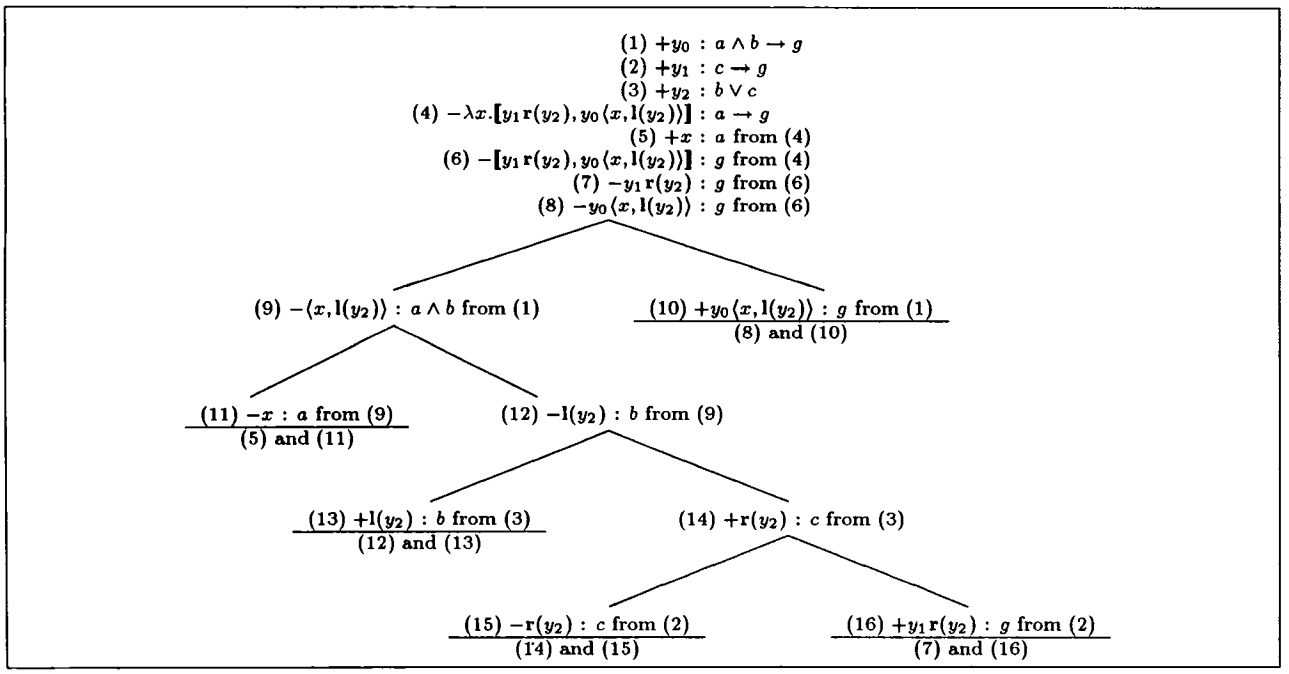

Figure 1: Closed tableau T

3.3 Soundness

As we have seen in the previous section extended λ-terms are an appropriate means for representing tableau proofs. Extended λ-terms can also be seen as programs which can be evaluated by some special conversion rules. However for efficiency, λ-terms with explicit case analysis are more suitable for programs. Therefore, we give a procedure which transfoms extended λ-terms to usual λ-terms with explicit case analysis (instead of implicit) and pair applications (instead of projections). This transformation is also used to establish a soundness result.

The idea of the transformation is simple: the λ-term is visited from outermost to innermost. As far as λ-terms $\mathbf{l}(N)$ and $\mathbf{r}(N)$ are detected, where the transformation of $N \in \Lambda_E$ leads to a λ-term $N' \in \Lambda$, we replace $\mathbf{l}(N)$ and $\mathbf{r}(N)$ by introducing a **when**-construct with N' as first argument. Similarly, projections are handled.

Definition 8 (Transfomation t)

$t(M) = \mathbf{when}(t(N), \lambda x_1.t(M[\mathbf{l}(N)/x_1]), \lambda x_2.t(M[\mathbf{r}(N)/x_2]))$
  if $S_{lr}(M) \neq \emptyset$, $N = min(S_{lr}(M)) = min(S(M))$[17]
$t(M) = \mathbf{split}(t(N), \lambda x_1.\lambda x_2.t(M[\mathbf{fst}_\exists(N)/x_1][\mathbf{snd}_\exists(N)/x_2]))$
  if $S_{fstsnd}(M) \neq \emptyset$, $N = min(S_{fstsnd}(M)) = min(S(M))$[17]
$t(M) = t_1(M)$, if $S(M) = \emptyset$.

$S_{lr}(M) = \{P \mid \mathbf{l}(P) \in F(M), \mathbf{r}(P) \in F(M) \text{ and } t(P) \in \Lambda\}$ [18]
$S_{fstsnd}(M) = \{P \mid \mathbf{fst}_\exists(P) \in F(M) \text{ and } t(P) \in \Lambda\} \cup$
$\{P \mid \mathbf{snd}_\exists(P) \in F(M) \text{ and } t(P) \in \Lambda\}$
$S(M) = S_{lr}(M) \cup S_{fstsnd}(M)$

$t_1(\lambda x.M) = \lambda x.t(M)$
$t_1(MN) = t(M)t(N)$
$t_1(\langle M, N\rangle) = \langle t(M), t(N)\rangle$
$t_1(\mathbf{inl}(M)) = \mathbf{inl}(t(M))$
$t_1(\mathbf{inr}(M)) = \mathbf{inr}(t(M))$
$t_1(\mathbf{absurd}(M)) = \mathbf{absurd}(t(M))$
$t_1([\![M_1, M_2, ..., M_n]\!]) = min\{t(M_i) \mid 1 \leq i \leq n \text{ and } t(M_i) \in \Lambda\}$
$t_1(f(M_1, M_2, ..., M_n)) = f(t(M_1), t(M_2), ..., t(M_n))$
$t_1(\mathbf{fst}(M)) = \mathbf{fst}(t(M))$
$t_1(\mathbf{snd}(M)) = \mathbf{snd}(t(M))$
$t_1(\mathbf{fst}_\exists(M)) = c$, where c is any constant [19]
$t_1(x) = x$

Note, that the function t is partial; e.g. $t(\lambda x.\mathbf{l}(x))$ is not defined.

Theorem 9 (Soundness)
If $\Sigma \vdash_{gc} M : \alpha$ *then* $\Sigma \vdash t(M) : \alpha$.

In the example above we have shown that

$$\Sigma \vdash_{gc} \lambda x.[\![y_1\mathbf{r}(y_2), y_0\langle x, \mathbf{l}(y_2)\rangle]\!] : a \to g$$

By transformation t and the soundness result it follows

$$\Sigma \vdash \mathbf{when}(y_2, \lambda x_1.\lambda x.(y_0\langle x, x_1\rangle), \lambda x_2.\lambda x.(y_1 x_2)) : a \to g$$

This λ-term corresponds to a natural deduction proof, where first a case analysis is carried out by applying the ($\vee$E)-rule to the assumption $b \vee c$. Then in each case, $a \to g$ on the right-hand side is decomposed by ($\to$I).

3.4 Completeness

The main key to prove completeness is the normalization theorem (see e.g. [Gir89]), which says that each typed λ-term $M \in \Lambda$ (i.e. $\Sigma \vdash M : \alpha$, for some Σ and α), can be

[17] x_1, x_2 are new. *min* is defined wrt. any given term ordering.
[18] $F(M) = \{N \mid N$ occurs in M and no free variable of N is bound in $M\}$. E.g., $F(\lambda x.x(yz)) = \{y, z, yz, \lambda x.x(yz)\}$.
[19] This case is helpful in the proof of soundness.

brought into a normal form M' such that $M \rhd^* M'$ and $\Sigma \vdash M' : \alpha$. Then, by using a function $t' : \Lambda \longrightarrow \Lambda_E$, we can show that from $\Sigma \vdash M' : \alpha$ it follows $\Sigma \vdash_{gc} t'(M') : \alpha$.

Definition 10 (Transformation t')

$t'(\lambda x.M) = \lambda x.t'(M)$
$t'(MN) = t'(M)t'(N)$
$t'((M,N)) = (t'(M), t'(N))$
$t'(\mathbf{inl}(M)) = [\mathbf{inl}(t'(M)), \mathbf{inr}([\,])]$
$t'(\mathbf{inr}(M)) = [\mathbf{inl}([\,]), \mathbf{inr}(t'(M))]$
$t'(\mathbf{fst}(M)) = \mathbf{fst}(t'(M))$
$t'(\mathbf{snd}(M)) = \mathbf{snd}(t'(M))$
$t'(\mathbf{absurd}(M)) = \mathbf{absurd}(t'(M))$
$t'(\mathbf{split}(M, \lambda x_1.\lambda x_2.N))$
$\quad = t'(N)[x_1/\mathbf{fst}_3(t'(M))][x_2/\mathbf{snd}_3(t'(M))]$
$t'(\mathbf{when}(M, \lambda x_1.N_1, \lambda x_2.N_2)) = [\ t'(N_1)[x_1/\mathbf{l}(t'(M))],$
$\quad t'(N_2)[x_2/\mathbf{r}(t'(M))]\ ]$
$t'(f(M_1, M_2, ..., M_n)) = f(t'(M_1), t'(M_2), ..., t'(M_n))$
$t'(x) = x$

Lemma 11 *If $\Sigma \vdash M : \alpha$ then $\Sigma \vdash_{gc} t'(M) : \alpha$, for any $M \in \Lambda$ in normal form (wrt. $\rhd$).*

From this, the completeness result is obtained by the normalization theorem.

Theorem 12 (Completeness) *If $\Sigma \vdash M : \alpha$ then there exists some $M' \in \Lambda_E$ such that $\Sigma \vdash_{gc} M' : \alpha$.*

4 The Lifted Calculus

In contrast to the ground calculus, guessing λ-terms and individual terms is avoided by using metavariables, which may be instantiated during tableau extensions. The instantiations result from applications of tableau rules and closing of branches. This technique is also known as lifting technique. Actually, the lifted calculus is not a proper lifted version of the ground calculus, because the $(-[\,])$-rule has been made implicit for some reasons which shall be discussed later.

E.g., the rules for the conjunction are as follows

$$(+\wedge)\ \frac{+M : \alpha \wedge \beta}{\begin{array}{l}+\mathbf{fst}(M) : \alpha \\ +\mathbf{snd}(M) : \beta\end{array}} \qquad (-\wedge)\ \frac{-\langle X, Y\rangle : \alpha \wedge \beta}{-X : \alpha \mid -Y : \beta}$$

where X and Y are metavariables. $(+\wedge)$ works as before whereas applications of $(-\wedge)$ lead to instantiations. In order to synthesize a λ-term M such that $x : a \wedge b \vdash_{gc} M : b \wedge a$ we start with the initial tableau T_1 (shown in figure 2), where X_1 is a metavariable. By application of $(+\wedge)$ to (1) we get tableau T_2. Applying $(-\wedge)$ to (2), the tableau is extended to T_3 and X_1 is instantiated by $\langle X_2, X_3\rangle$, where X_2, X_3 are new metavariables. Both branches can be closed by the substitutions $[X_2/\mathbf{snd}(x)]$ and $[X_3/\mathbf{fst}(x)]$ (see T_4). The substitutions are found by unification. The desired λ-term $\langle \mathbf{snd}(x), \mathbf{fst}(x)\rangle$ can now be picked up at node (2).

In order to fulfill the proviso for the rules $(- \rightarrow)$ and $(-\forall)$, we employ Skolemization technique.

$$(- \rightarrow)\ \frac{-\mathbf{lambda}(f(VL), X) : \alpha \rightarrow \beta}{\begin{array}{l}+f(VL) : \alpha \\ -X : \beta\end{array}}$$

Here, f is a new Skolem function, and VL is the set of all metavariables occuring in the branch where the rule is applied. So, the Skolem expression $f(VL)$ can be understood as a new variable. In order to avoid conflicts with substitutions, we use a non-binding abstraction $\mathbf{lambda}(N, M)$. As far as tableaux are closed, non-binding abstractions $\mathbf{lambda}(N, M)$ are replaced by usual abstractions $\lambda x.M[N/x]$.

The main difficulty arises from the implicit case analysis. If the $(-[\,])$-rule would be applied the number of cases n would have to be guessed. For that reason, this rule is omitted and we presume that the λ-term in each negatively signed proof relation is a priori an implicit case analysis. In general, a negatively signed proof relation is of the following form:

$$-[\![M_1, M_2, ..., M_n | L]\!] : \alpha,\ ^{20} \text{ where } n \geq 0. \qquad (*)$$

L is a metavariable which stands for a list of further cases. L can be refined by rule applications and closing of branches. If α in $(*)$ would be $\beta \wedge \gamma$, then $(-\wedge)$ may be applied. That would lead to a refinement of $(*)$ by one further case

$$-[\![M_1, M_2, ..., M_n, \langle L_1, L_2\rangle | L_3]\!] : \beta \wedge \gamma$$

and to a tableau extension by a branching with two new nodes labeled with

$$-L_1 : \beta \text{ and } -L_2 : \gamma$$

Now, L_1, L_2, L_3 are metavariables which may be refined later.

Similarly, L would have been refined if $(*)$ would have been involved in closing a branch.

The other tableau rules are defined as described for the conjunction. Rules for positively signed proof relations remain as before (see table 2). The only exception is for the rules $(+ \rightarrow)$ and $(+\forall)$, where metavariables L and X are used instead of N and t, respectively. Rules for negatively signed proof relations are modified by using metavariables instead of λ-terms. Skolem expressions are introduced in the rules $(- \rightarrow)$ and $(-\forall)$. The rule $(-[\![]\!])$ is omitted. We leave out a complete definition of the lifted calculus and finally mention the main result.

Theorem 13 (Soundness and Completeness) *The lifted calculus is sound and complete wrt. the ground calculus.*

5 Conclusions

There are two worlds of proof methods for the intuitonistic first-order logic: proof-theoretic calculi and semantic-based calculi. The natural deduction calculus and the related sequent calculus come from the proof theory area. Because of the strong order-dependence of rule applications, the dead ends problem arises which results in a big amount of backtracking. However, λ-terms are given easily, when a formula is proven valid. Resolution and matrix

[20] $[\![M_1, M_2, ..., M_n | L]\!]$ is an abbreviation for $[\![M_1, M_2, ..., M_n]\!] \div L$, where $\div$ means concatination.

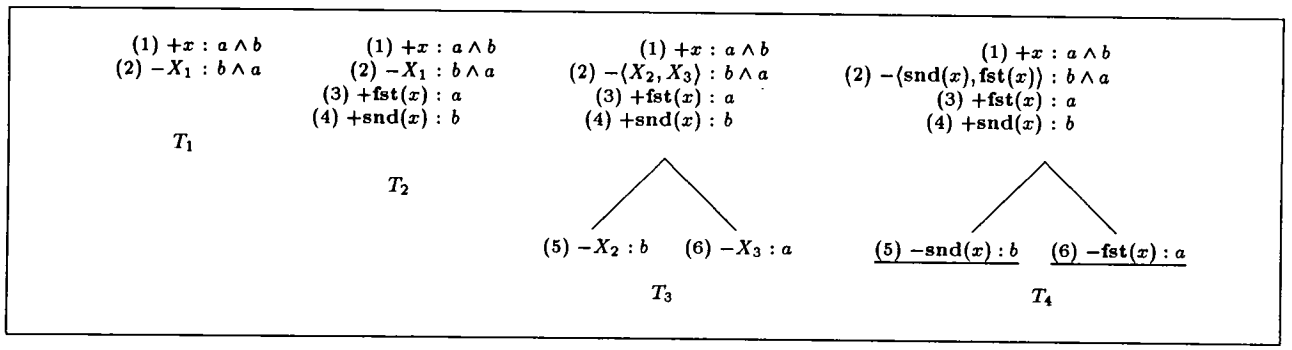

Figure 2: Some tableau extensions with the lifted calculus

method are semantic-based proof methods, dedicated to automatic reasoning. They do not have dead ends problem. However, synthesis of λ-terms is not considered.

The tableau calculus presented here subsumes the advantages of both worlds: it is automatic and efficient (free from dead ends) and allows the synthesis of λ-terms. One of the key notions is case analysis. In natural deduction proofs case analysis ($(\vee E)$-rule) has often to be done before other rule applications, while in tableau proofs case analysis ($(+\vee)$-rule) can be arbitrarily delayed. This property of the tableau calculus is reflected by a new λ-term construct, which we call implicit case analysis. Therefore and because of the use of metavariables and unification, the order of rule applications does not play any role.

Since the new calculus is based on the one for classical logic, all the techniques developed in the area of automated tableau-based theorem proving for classical logic (e.g. [Sch85], [OpSu88], [Fit90]) can be carried over. We have implemented the calculus in Prolog. A strategy [Sch85] which is similar to model elimination has been built in. Several formulas like the maximum of two and of three numbers and integer square root were proven automatically.[21] It is clear that this theorem prover can easily be integrated as automatic tool in a more interactive system like NUPRL and OYSTER.

There is an interesting relation to the deductive program synthesis approach of Manna and Waldinger [MaWa80]. Their approach is also based on the classical tableau calculus, but branching (splitting) is not allowed. E.g. a goal $\alpha \wedge \beta$ (in our calculus $-\alpha \wedge \beta$) is not allowed to be decomposed. This property lies inherently in their calculus. In contrary, in our calculus branching is allowed and therefore decompostion of all formula kinds is possible. The advantage is more flexibility in integrating different proof search strategies.

References

[Bit91] Bittel, O., *Ein tableaubasierter Theorembeweiser für die intuitionistische Logik*, Ph.D. thesis, GMD-Report 198, Oldenbourg-Verlag, 1991.

[Bit92] Bittel, O., *The λ-Tableau Calculus: A new approach to Theorem Proving in the Intuitionistic Logic*, Workshop on Theorem Proving with Analytic Tableaux and Related Methods, Universität Karlsruhe, Fakultät für Informatik, Interner Bericht 8/92, 1992.

[BSH90] Bundy, A., Smaill, A. Hesketh, J., *Turning Eureka Steps into Calculations in Automatic Programm Syntesis*, in Proceedings of the First Workshop on Logical Frameworks, Antibes, 1990.

[Con86] Constable, R. L. et al., *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, 1986.

[Coq90] Coquand, T., *On the Analogy Between Propositions and Types*, in Logical Foundations of Functional Programming, G. Huet (ed.), Addison-Wesley, 1990.

[Fit83] Fitting, M., *Proof Methods for Modal and Intuitionistic Logics*, Holland, 1983.

[Fit90] Fitting, M., *First-Order Logic and Automated Theorem Proving*, Springer-Verlag, 1990.

[Gir89] Girard, J.-Y., Lafont, Y., Taylor, P., *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science 7, 1989.

[How80] Howard, W. A., *The Formulae as Types Notion of Construction*, in H. B. Curry – essays on Combinatory Logic, λ -calculus and Formalism, Seldin/Hindley (Eds.), 579-606, Academic Press, 1980.

[Mar82] Martin-Löf, P., *Constructive Mathematics and Computer Programming*, in I. J. Cohen, J. Los, H. Pfeiffer and K. D. Podewski (Eds.), Logic, Methodology and Phylosophy of Science VI, 153-179, North-Holland, 1982.

[MaWa80] Manna, Z., Waldinger, R., *A Deductive Approach to Program Synthesis*, ACM TOPLAS Vol. 2, No. 1, 90-121, 1980.

[Ohl88] Ohlbach, H. J., *A Resolution Calculus for Modal Logics*, 9. CADE, LNCS 310, 1988.

[OpSu88] Oppacher, F., Suen, E., *HARP: A Tableau-Based Theorem Prover*, Journal of Automated Reasoning, Vol. 4, 69 - 100, 1988.

[Sch85] Schönfeld, W., *Prolog Extensions Based on Tableau Calculus*, IJCAI, 1985.

[Sha92] Shankar, N., *Proof Search in the Intuitionistic Sequent Calculus*, 11. CADE, LNCS 607, 1992.

[Wal88] Wallen, L. A., *Matrix Proof Methods for Modal Logics*, IJCAI, 1988.

[Wos90] Wos, L., *The Problem of Finding a Mapping between Clause Representation and Natural-Deduction Representation*, Journal of Automated Reasoning 6, 211-212, 1990.

[21] For the last example induction on natural numbers is necessary. The induction step was proven fully automatic.