Guiding Program Development Systems by a Connection Based Proof Strategy

Christoph Kreitz Jens Otten Stephan Schmitt

Fachgebiet Intellektik, Fachbereich Informatik
Technische Hochschule Darmstadt
Alexanderstr. 10, 64283 Darmstadt, Germany
{kreitz,jeotten,steph}@intellektik.informatik.th-darmstadt.de

Abstract. We present an automated proof method for constructive logic based on Wallen's matrix characterization for intuitionistic validity. The proof search strategy extends Bibel's connection method for classical predicate logic. It generates a matrix proof which will then be transformed into a proof within a standard sequent calculus. Thus we can use an efficient proof method to guide the development of constructive proofs in interactive proof/program development systems.

1 Introduction

According to the *proofs-as-programs* paradigm of program synthesis the development of verifiably correct software is strongly related to proving theorems about the satisfiability of a given specification. If such a theorem is proven in a constructive manner then the proof construction implicitly contains an algorithm which is guaranteed to solve the specified problem. In contrast to 'deductive' synthesis for which classical proof methods are sufficient, however, synthesis techniques based on this paradigm have rely on *constructive* logics for their adequacy. Therefore computer systems which support the development of constructive proofs and the extraction of programs from proofs are very important for program synthesis.

Such systems (e.g. NuPRL [6], Oyster [5], Isabelle [14], LEGO [15]) are usually designed as *interactive* proof editors supported by a *tactic* mechanism for programming proofs on the meta-level. Most of them are based on a very expressive constructive theory. To allow a proper interaction between the system and its users this theory is usually formulated as natural deduction or sequent calculus and includes a calculus for predicate logic similar to Gentzen's [8] calculi for intuitionistic logic. It has been demonstrated that these systems can be used quite successfully, if properly guided, but the degree of automatic support is very weak. A user often has to deal with subproblems which appear trivial to him since they depend solely on predicate logic. A formal proof, however, turns out to be rather tedious since existing tactics dealing with predicate logic are far from being complete.

On the other hand, theorem provers like Setheo [9], Otter [20], or KoMeT [2] have demonstrated that formal reasoning in *classical predicate logic* can be automated sufficiently well. It would therefore be desirable to integrate techniques from automated theorem proving into already existing program synthesis tools. This would not only liberate the users of a proof development system from having to solve problems from first order logic by hand but would also make it possible to generate simple (non-recursive) algorithms fully automatically.

M. Proietti, ed., 5^{th} *International Workshop on Logic Program Synthesis and Transformation (LoPSTR'95)*, LNCS 1048, pp. 137–151, ©Springer Verlag, 1996.

Two problems have to be solved for this purpose. Firstly, since a *constructive proof* contains much more information than a classical one many of the well known classical normal forms and equivalences are not valid constructively. Despite of the success in the classical case there is not yet an efficient proof procedure for constructive logics and therefore the existing classical proof methods need to be extended. Secondly, we have to take into account that efficient *proof search* must be based on some internal characterization of validity which avoids the usual redundancies contained in natural deduction and sequent proofs while for the sake of comprehensibility (and for an extraction of programs in the usual fashion) the proof *representation* must be based on the formal calculus underlying the program development system. This makes it necessary to convert the internal representation of proofs into a more natural one.

We have developed a complete proof procedure for constructive first-order logic and a technique for integrating it into a program development system based on a sequent calculus. As a starting point we have used a proof procedure called the *connection method* [3, 4] which has successfully been realized in theorem provers for classical predicate logic like Setheo [9] and KoMeT [2]. It is based on a characterization for the classical validity of logical formulae which recently has been extended by Wallen [19] into a matrix characterization of *intuitionistic* validity. We have considerably extended the connection method according to this characterization and developed a method for converting matrix proofs into sequent proofs. The combined procedure, whose structure is depicted in figure 1, is currently being implemented as a proof tactic of the NuPRL proof development system [6] and will thus support the efficient construction of proofs and verified routine programs within a rich constructive theory. It proceeds in three steps.

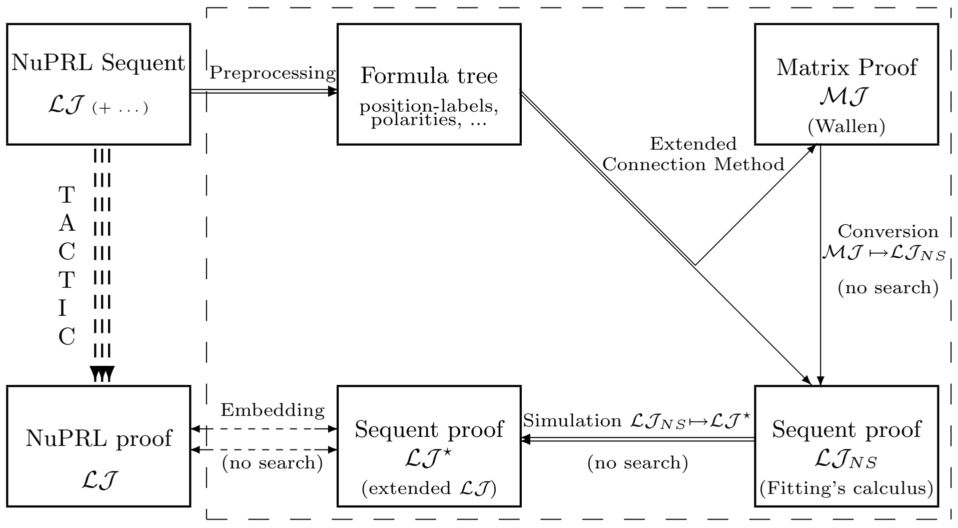


Fig. 1. Structure of the combined proof procedure

1. In a preprocessing step a NuPRL sequent will be converted into a formula tree augmented by information (such as polarities, position labels and type-labels) which will be helpful for guiding the proof search.

2. In the second step we use our extended connection method to search for a proof according to Wallen's matrix characterization. While originally we were interested only in constructing a matrix proof which should be translated into a sequent proof afterwards our investigations have shown that it is helpful to exploit the close relation between a matrix proof and a proof in Fitting's [7] sequent calculus $\mathcal{LJ}_{NS}$[1] and to consider the structure of the corresponding sequent proof already *during* the proof search and design our proof procedure as a hybrid method combining the connection method with the sequent calculus. The result of our proof search procedure will be a matrix proof which can already be considered as the skeleton of an $\mathcal{LJ}_{NS}$-proof.
3. In the final step the information gained during proof search will be used to construct a standard sequent proof which is valid in NuPRL. Since our proof search already yields the structure of a sequent proof in $\mathcal{LJ}_{NS}$ we are liberated from having to convert a matrix proof ($\mathcal{MJ}$) into an $\mathcal{LJ}_{NS}$-proof. Nevertheless the proof still needs to be converted since Fitting's *non-standard* sequent calculus $\mathcal{LJ}_{NS}$ is more complex than the standard Gentzen-like calculus $\mathcal{LJ}$ used in program development systems. For reasons which we shall discuss in section 4 our proof transformation will proceed in two smaller steps which do not involve any additional search.

In the rest of this paper we shall describe the essential components of our procedure. After explaining how to construct the augmented formula tree in section 2 we shall elaborate the extended connection method in section 3. In section 4 we shall then discuss the procedure for constructing a standard sequent proof acceptable for the NuPRL System. We conclude with a few remarks on implementation issues and future investigations.

2 Constructing the Augmented Formula Tree

In the first step of our procedure a given sequent has to be converted into a formula tree which then can be investigated by the proof search method. Except for dealing with a few peculiarities of NuPRL[2] this is can be done by a standard procedure which collects the assumptions (antecedent) of a sequent on the left hand side of an implication whose right hand side is the sequent's conclusion (succedent) and adds universal quantifiers for all the free variables. Constructing a tree representation of the resulting formula again uses a standard technique. As an example, figure 2 presents the formula tree of the formula

$$F \equiv (S \wedge (\neg(T \Rightarrow R) \Rightarrow P)) \Rightarrow (\neg((P \Rightarrow Q) \wedge (T \Rightarrow R)) \Rightarrow (S \wedge \neg\neg P))$$

which we shall later use as a running example for our proof procedure.

In this tree each node represents a sub-term of the given formula and is marked by its major logical connective. In addition to that it is associated with a polarity, a position label, and a type label. These informations are necessary for guiding the proof search and can be computed while constructing the formula tree.

[1] When developing his matrix characterization for intuitionistic validity Wallen [19] has used Fitting's formulation of the sequent calculus as theoretical framework to prove his characterization theorems.

[2] Some of the assumptions declare the type of certain variables or the wellformedness of certain expressions and may simply be ignored.

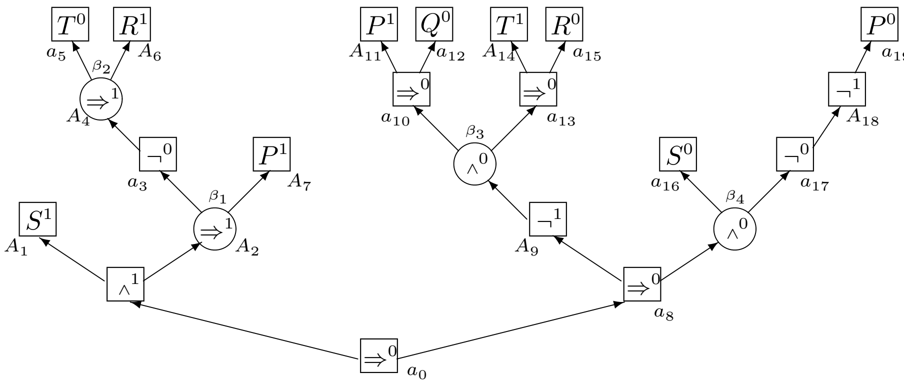


Fig. 2. Formula tree with polarities, positions, and type labels

- The *polarity* (0 or 1) of a node indicates whether the corresponding sub-formula would occur negated (1) in a normal form of the given formula or not (0). The polarity of the root is 0 and the polarity of other nodes is determined recursively by the polarity and the connective of their parents. In figure 2 the polarity of each node occurs on top of its connective.
  Pairs of atomic formulae with the same predicate symbol but different polarities (so-called *connections*) correspond to the axioms in a sequent proof and are a key notion in the matrix characterization of validity.
- While polarities are important for both classical and intuitionistic proof methods the *position labels* (*positions*) are necessary only for constructive reasoning. They encode the fact that in intuitionistic sequent calculi the order of rule applications cannot be permuted as easily as in classical logic. Special nodes ($\Rightarrow^0$, $\forall^0$, $\neg^0$, and atoms with polarity 0) correspond to rules which – when being used in a top-down fashion in order to *reduce* the node – will delete formulae from the actual sequent. In a proof these rules have to be applied as late as possible.
  By analyzing the sequence of positions on the paths from the root to two connected atomic formulae (the *prefixes* of the formulae) one can determine whether the two atoms can form an axiom which closes some subproof and in which order the corresponding rules need to be applied in order to construct this subproof. Technically this is done by assigning constants (position labels with small letters) to the special nodes and variables to the others. By trying to unify the prefixes of two connected atoms one may then check whether both atoms can be reached by a sequence of rule applications at the same time. In figure 2 we have assigned positions to nodes containing a negation, implication, universal quantifier, or an atom. All other nodes are not significant for building prefixes.
- *Type labels*[3] express a property of sequent rules which is important mostly during proof search. Nodes of type β (i.e. $\wedge^0$, $\vee^1$, and $\Rightarrow^1$) are particularly important since the corresponding rules would cause a proof to branch into two independent subproofs. In order to complete the proof each of these subproofs (encoded by a *path*) must be closed with an axiom (i.e. a connection). Thus investigating β-branches not yet closed will help identifying those connections which will contribute to a progress in the remaining proof search. In figure 2 we have marked β-nodes by circles and labels $\beta_1, \ldots \beta_4$. Nodes of other types remain unmarked.

[3] The assignment of types to nodes follows the laws of the tableaux calculus [1].

3 The Connection Method for Constructive Logics

After constructing the augmented formula tree a proof search will focus on 'connections' between atomic formulae which can be shown to be 'complementary'. According to the characterization of validity all 'paths' through the formula tree must contain such a connection. In our procedure the set of all paths (within a subproof under consideration) which are not yet investigated will be encoded by the 'active path'. This will help to guide the search efficiently. Before describing our procedure in detail we shall briefly explain the notions of connections, paths, and complementarity and resume Wallen's matrix characterization for intuitionistic validity.

3.1 A Matrix Characterization for Intuitionistic Validity

In propositional classical logic a formula F is valid if there is a spanning set of connections for F. A *connection* is a pair of atomic formulae with the same predicate symbol but different *polarities* such as P^1 and P^0. A set of connections *spans* a formula F if every *path* through F[4] contains at least one connection. This characterization also applies to predicate logic if the connected formulae can be shown to be *complementary*, i.e. if all the terms contained in connected formulae can be made identical by some (*first-order*/quantifier) substitution σ_Q.

In *sequent calculi* like Gentzen's $\mathcal{LK}$ and $\mathcal{LJ}$ [8] or Fitting's calculi [7] the difference between classical and intuitionistic reasoning is expressed by certain restrictions on the intuitionistic rules. If rules are applied in a top down fashion these restrictions cause formulae to be deleted from a sequent. Applying a rule (i.e. *reducing* a sub-formula) too early may delete a formula which later will be necessary to complete the proof. Because of this *non-permutability* of rules in intuitionistic sequent calculi the order of rule applications must be arranged appropriately. In Wallen's matrix characterization this requirement is expressed by an *intuitionistic* substitution σ_J which makes the prefixes of connected atoms identical. A *prefix* of a node A consists of the sequence of position labels on the path from the root to A and encodes the sequence of rules which have to be applied in order to isolate the sub-formula described by A. The prefix of the atom R^0 in figure 2, for instance, is $a_0a_8A_9a_{13}a_{15}$.

Both the first-order and the intuitionistic substitution can be computed by unification algorithms (see section 3.3) and put restrictions on the order of rule-applications (see [19] or [11]). $\sigma_J(B) = b_1...b_n$ for instance, means that all the special positions $b_1...b_n$ *must* have been reduced *before* position B and after its predecessor. Otherwise certain sub-formulae which are necessary assumptions for applying the sequent rule corresponding to B would not be available. Similarly $\sigma_Q(x) = t$ requires all the Eigenvariables of t to be introduced by some rule before the quantifier corresponding to x can be reduced. Together with the ordering of the formula tree the *combined substitution* $\sigma := (\sigma_Q, \sigma_J)$ determines the ordering $\lhd$ in which a given formula F has to be reduced by the rules of the sequent calculus. This ordering must be acyclic since otherwise no proof for F can be given. If $\lhd$ is acyclic then σ is called *admissible*.

During the proof search it may become necessary to create multiple instances of the same sub-formula. The number of copies generated to complete the proof is

[4] A *path* through F is a subset of the atoms of F which corresponds to a horizontal path through the (nested) *matrix representation* of F. See [19, p. 215] for a complete definition.

called *multiplicity* μ. Again, a multiplicity may be due to a quantifier or specific to intuitionistic reasoning. Altogether, the following theorem has been proven in [19].

Theorem 1 (Matrix characterization of intuitionistic validity).
A formula F is intuitionistically valid if and only if there is

- *a multiplicity μ,*
- *an admissible combined substitution $\sigma := (\sigma_Q, \sigma_J)$,*
- *a set of connections which are complementary under σ*

such that every path through the formula F contains a connection from this set.

3.2 Connection Based Proof Search

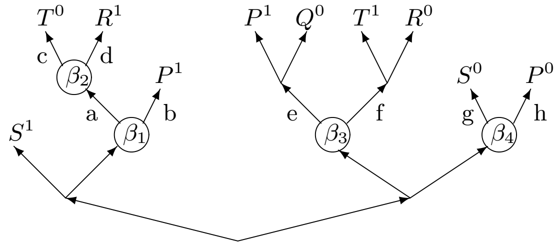
In Bibel's classical connection method [3, 4] the search for a matrix proof of a given formula proceeds by considering connections between atomic formulae whose subterms can be unified. The selection of appropriate connections is guided by the *active path* and the set of *open goals.* Developing a procedure which constructs intuitionistic proofs on the basis of theorem 1 means extending the key concepts for guiding the search procedure accordingly. This allows a formulation of our procedure which is similar to the one of the classical connection method operating on logical formulae in non-normal form.

Our investigations have shown that during the proof search one should not only consider paths and connections but also the branching structure of the corresponding (partial) sequent proof. This structure provides valuable informations about the reduction ordering $\lhd$ to be constructed and helps selecting appropriate connections guiding the search process. Furthermore, it allows to consider *local* substitutions (see [11, section 5]) instead of global ones, i.e. substitutions which can be applied independently within sub-proofs of a sequent proof. Such a local view reduces the number of copies of sub-formulae which have to be generated to find a (global) substitution and keeps the search space and the proof size smaller. Since it also simplifies the conversion of the 'abstract proof' into a humanly comprehensible sequent proof we have designed our proof search procedure as a hybrid method which generates a matrix proof and the structure of the corresponding sequent proof simultaneously.

We shall now describe our search strategy by developing a proof for the formula

$$F \equiv (S \land (\neg(T \Rightarrow R) \Rightarrow P)) \Rightarrow (\neg((P \Rightarrow Q) \land (T \Rightarrow R)) \Rightarrow (S \land \neg\neg P))$$

whose formula tree has already been presented in figure 2. To simplify our illustration we shall from now on display only a skeleton of this tree in which only positions with type label β, branches rooted at such a β-position (i.e. a,b,...,h – also called *β-branches*), and atomic formulae are marked.

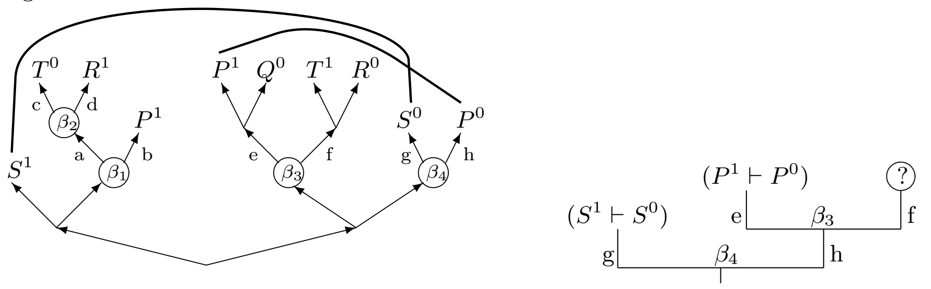


Our search procedure will consider connections between atomic formulae whose sub-terms and prefixes can be unified and build the skeleton of a sequent proof whose nodes and branches are labeled similarly to the formula tree. The search is guided by the *active β-path* and the set of *open subgoals* in the partial sequent proof. These notions extend the original concepts mentioned above and are roughly defined as follows (see [13] for precise definitions).

Definition 1.

1. *The* β-prefix *of an atom A is the set of all (labels of) β-branches that dominate A in the formula tree.*
2. *The* active β-path $\mathcal{P}_\beta$ *for a branch u in the sequent proof is the set of labels of branches on the path between u and the root of the proof tree.*
3. *The* active path $\mathcal{P}$ *for a branch u of the proof tree is the set of atoms whose* β-prefix *is a subset of $\mathcal{P}_\beta(u)$.*[5]
4. *The set of* open subgoals $\mathcal{C}_\beta$ *is the set of open branches in the sequent proof structure (i.e. branches which are not closed by an axiom).*

To prove F we begin by selecting an arbitrary atomic formula, say P^1, in branch 'e' of the formula tree and connect it with the atom P^0 in the 'h'-branch. This means that in a sequent proof we have to reduce two β-positions, namely β_3 and β_4. Unifying the prefixes of P^1 and P^0 (see section 3.3) of the two atoms leads to the substitution $\sigma_J \equiv \{A_9 \backslash a_{17}B,\ A_{18}\backslash Ba_{10}C,\ A_{11}\backslash Ca_{19}\}$ where B and C are new variables. Within an intuitionistic sequent proof the node marked by a_{17} must therefore be reduced before A_9, a_{10} before A_{18}, and a_{19} before A_{11}. Furthermore β_4 must be reduced before a_{17} and A_9 before β_3 according to the ordering of the formula tree in figure 2. Thus σ_J induces the reduction ordering $\beta_4 \lhd \beta_3$ and we have to split into the branches 'g' and 'h' (corresponding to β_4) before we split the 'h'-branch into 'e' and 'f' (corresponding to β_3). This closes the 'e'-branch in the sequent proof as shown in figure 3.



T^0 R^1 c d β_2 P^1 a b S^1 β_1
P^1 Q^0 T^1 R^0 e f β_3 S^0 P^0 g h β_4
$(P^1 \vdash P^0)$? e β_3 f
$(S^1 \vdash S^0)$ g β_4 h

Fig. 3. The first and second proof step

In the next step we choose the 'g'-branch from the set $\mathcal{C}_\beta = \{\mathrm{g}, \mathrm{f}\}$ of open subgoals. The active β-path $\mathcal{P}_\beta = \{\mathrm{g}\}$ for 'g' induces an active path $\mathcal{P} = \{S^1, S^0\}$. The only atom S^0 in the 'g'-branch of the formula tree can therefore be connected to S^1 in the active path without further reductions of β-nodes. The unification of the prefixes of S^1 and S^0 consists of a simple matching which extends σ_J by $\{A_1\backslash a_8 a_{16}\}$. This closes branch 'g' in the sequent proof.

[5] $\mathcal{P}$ is thus the set of atoms (in the *formula tree*) which can be reached from the root or any position corresponding to a element of $\mathcal{P}_\beta(u)$ without passing through a β-position.

The only open branch is now the 'f'-branch ($\mathcal{C}_\beta = \{\text{f}\}$). In the formula tree this branch contains the two atoms R^0 and T^1. We select R^0 and connect it with R^1 in branch 'd' of the formula tree. In the active path $\mathcal{P} = \{S^1, P^0, R^0, T^1\}$ for 'f' ($\mathcal{P}_\beta = \{\text{h,f}\}$) this atom is not yet included. Thus we have to reduce β_1 which splits the proof into 'a' and 'b' and β_2 which splits the 'a'-branch into 'c' and 'd'. The unification of the prefixes of R^0 and R^1 extends σ_J by $\{A_2\backslash a_8 a_{17} D,\ B\backslash D a_3 E,\ A_4\backslash E a_{13} a_{15},\ A_6\backslash\epsilon\}$. Together with the tree ordering ($\beta_4 < a_{17},\ \beta_1 = A_2 < a_3,\ A_9 < \beta_3 < a_{13},\ A_4 = \beta_2$) it induces the reduction ordering $\beta_4 \lhd \beta_1 \lhd \beta_3 \lhd \beta_2$. Therefore we have to insert the β-split into 'a' and 'b' *between* the reduction of β_4 and β_3 (leaving the rest of the partial sequent proof unchanged), split into the branches 'c' and 'd' *after* reducing β_3, and close the 'd'-branch by the axiom $R^1 \vdash R^0$. The result is shown in figure 4.

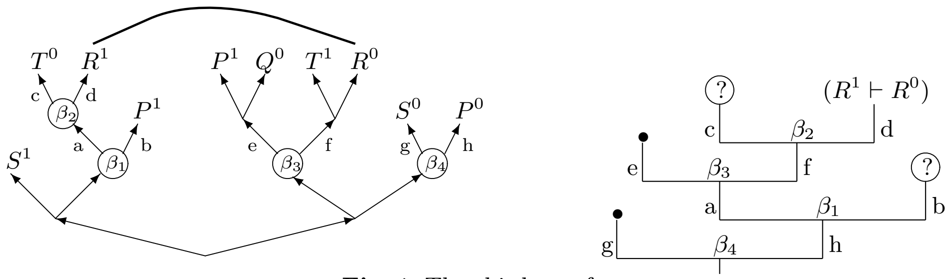


Fig. 4. The third proof step

After the third step the two branches 'b' and 'c' remain open ($\mathcal{C}_\beta = \{\text{b, c}\}$). The active β-paths for 'b' ($\mathcal{P}_\beta$= {h, b}) and for 'c' ($\mathcal{P}_\beta$= {h,a,f,c}) induce active paths $\mathcal{P} = \{S^1, P^0, P^1\}$ and $\mathcal{P} = \{S^1, P^0, T^1, R^0, T^0\}$ respectively. To close these branches we connect P^1 in the 'b'-branch of the formula tree to P^0 in the active path for 'b' and T^0 in the 'c'-branch to T^1 in the active path for 'c'. Again the unification of the prefixes consists of a simple matching. It extends σ_J by $\{A_7\backslash a_3 E a_{10} C a_{19},\ A_{14}\backslash a_{15} a_5\}$ which shows that now every branch in the sequent proof can be closed.

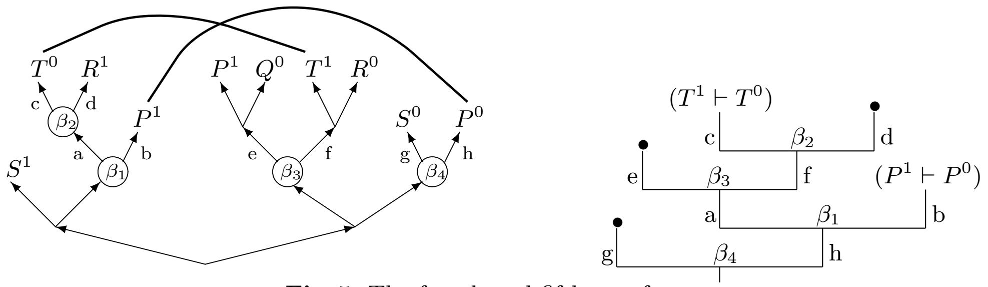
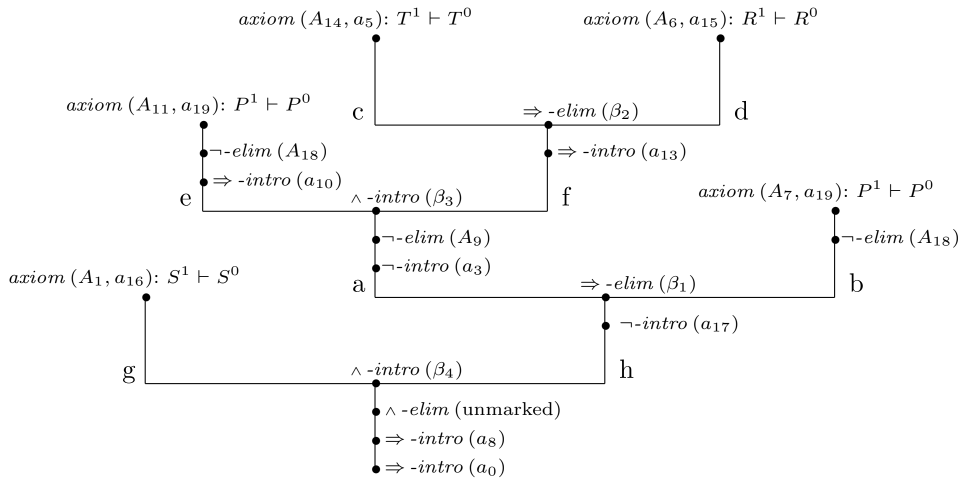


Fig. 5. The fourth and fifth proof step

This concludes the intuitionistic proof for F. All the paths through F contain a connection which is complementary under σ_J. Furthermore, a complete sequent proof can easily be constructed from the proof skeleton obtained in the process since the order in which all the other nodes have to be reduced is determined by the reduction ordering induced by σ_J and the ordering of the formula tree for F. The resulting sequent proof is presented in figure 6.

A full description of the complete proof search strategy for first-order intuitionistic logic which we just have illustrated is rather complex and shall therefore not be presented in this paper. Details can be found in [13].



$axiom\ (A_{14}, a_5)$: $T^1 \vdash T^0$
$axiom\ (A_6, a_{15})$: $R^1 \vdash R^0$
$axiom\ (A_{11}, a_{19})$: $P^1 \vdash P^0$
c
$\Rightarrow$-$elim\ (\beta_2)$
d
$\neg$-$elim\ (A_{18})$
$\Rightarrow$-$intro\ (a_{13})$
$\Rightarrow$-$intro\ (a_{10})$
e
$\wedge$-$intro\ (\beta_3)$
f
$axiom\ (A_7, a_{19})$: $P^1 \vdash P^0$
$\neg$-$elim\ (A_9)$
$\neg$-$elim\ (A_{18})$
$\neg$-$intro\ (a_3)$
$axiom\ (A_1, a_{16})$: $S^1 \vdash S^0$
a
$\Rightarrow$-$elim\ (\beta_1)$
b
$\neg$-$intro\ (a_{17})$
g
$\wedge$-$intro\ (\beta_4)$
h
$\wedge$-$elim$ (unmarked)
$\Rightarrow$-$intro\ (a_8)$
$\Rightarrow$-$intro\ (a_0)$

Fig. 6. Complete representation of the resulting sequent proof

3.3 Prefix-Unification

The efficiency of the proof search procedure described above strongly depends on the efficiency of the unification algorithms computing both the first-order and intuitionistic substitutions. While the first-order substitution σ_Q can be computed by well-known efficient unification algorithms we had to develop a specialized string-unification procedure for computing the intuitionistic substitution σ_J.

In general string-unification is very complicated and no efficient algorithm could be given so far. Fortunately, however, we do not have to unify arbitrary strings but only strings corresponding to the prefixes of two connected atoms. This enables us to put certain restrictions on the pairs of strings which have to be unified.

1. Since the prefix of a node is the sequences of position labels on the path from the root to the node we know that a position label (i.e. either a variable or a constant character) occurs at most once in the prefix string. Thus we only have to consider *strings without duplicates*.
2. Secondly prefixes correspond to branches in a formula tree. Therefore in two prefixes *equal position labels* can occur *only at the beginning*. Once the two prefixes diverge the remaining substrings are completely disjoint.

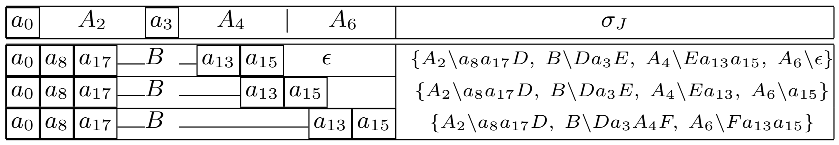
These two restrictions will cause prefix-unification to be much simpler than general string unification. We may simply skip common substrings at the beginning of the two prefixes and know that the remaining substrings will not have any variables in common. Prefix-unification can be realized as an extension of bidirectional matching where variables of one string will be matched against substrings of the other and, if necessary, vice versa.

Using bidirectional matching we can already compute a unifier for the prefixes of P^0 $(a_0a_8a_{17}A_{18}a_{19})$ and P^1 $(a_0a_8A_9a_{10}A_{11})$ in the first step of our proof search. A_9 can be instantiated by a_{17}, A_{18} by a_{10}, and A_{11} by a_{19}. In most cases, however, unification will be more complex. The prefix $a_0A_2A_7$ of the atom P^1 in the 'b'-branch, for instance, contains two adjoint variables. There are several possibilities for unifying it with the prefix of P^0: the variable A_2 may be instantiated with either $\epsilon, a_8, a_8a_{17}, a_8a_{17}A_{18}$, or $a_8a_{17}A_{18}a_{19}$ and A_7 will receive values correspondingly. Thus in general there will be more than one unifier for each pair of prefix-strings.

Furthermore we have to consider the fact that during proof search our unification procedure will be called several times and may yield different strings to be assigned to a given variable. The variable A_9, for instance will receive a value both by unifying the prefixes of P^0 and P^1 in the first step and by unifying the prefixes of R^0 and R^1 in the third. Therefore we should make sure that the strings assigned to a variable by our unification procedure are not already too specific since otherwise we will run into contradictions. The prefixes of P^0 and P^1 could also be unified by $\sigma_J = \{A_{18}\backslash a_3a_{10}, A_9\backslash a_{17}, A_{11}\backslash a_{19}\}$ which is much more compatible with the unifier required for the prefixes of R^0 and R^1. The unification algorithm should therefore create only the *most general unifiers* of two prefixes and thus leave room for specializations in the following steps of our proof search. A most general unifier (short *mgu*) takes into account that the strings assigned to variables within the two prefixes may overlap. In the case of P^0 and P^1 it would have to express that the end of (the string assigned to) A_9 may overlap with the beginning of A_{18} and that the end of A_{18} may overlap with the beginning of A_{11}. The only fixed information is that A_9 must begin with a_{17}, A_{11} must end with a_{19}, and A_{18} must contain a_{10}. All these informations can be encoded by introducing two new variables B and C and setting the most general unifier to $\{A_{18}\backslash Ba_{10}C, A_9\backslash a_{17}B, A_{11}\backslash Ca_{19}\}$.

Taking the above considerations into account our prefix-unification algorithm tries to compute the most general descriptions of all the possibilities for overlapping two prefix strings. We shall now describe this algorithm by an example unification of the prefixes $a_0A_2a_3A_4A_6$ of R^1 and $a_0a_8A_9a_{13}a_{15}$ of R^0 (which is the same as $a_0a_8a_{17}Ba_{13}a_{15}$ because of the earlier assignment $\{A_9\backslash a_{17}B\}$).

Our procedure starts by writing down the string $a_0A_2a_3A_4A_6$ in a way such that constants will receive a small slot (one character) and variables will receive slots of variable size (it suffices to reserve a slot of the size of the second string). In the rows below we write down the second string such that (except for the common substring a_0) each constant occurs in the range of a variable while the range of variables (e.g. B) may be stretched arbitrarily to make this possible. In the first row we begin by stretching the variables as little as possible. Below we will then systematically enumerate all the possibilities for extending the range of one or more variables. The most general unifier can then easily be computed: the assignment of constants is obvious and new variables will have to be generated if two variables overlap. For unifying the prefixes if R^1 and R^0 the most general unifiers are constructed according to the following diagram.


a_0 A_2 a_3 A_4 A_6 σ_J
a_0 a_8 a_{17} B a_{13} a_{15} ϵ $\{A_2\backslash a_8a_{17}D,\ B\backslash Da_3E,\ A_4\backslash Ea_{13}a_{15},\ A_6\backslash\epsilon\}$
a_0 a_8 a_{17} B a_{13} a_{15} $\{A_2\backslash a_8a_{17}D,\ B\backslash Da_3E,\ A_4\backslash Ea_{13},\ A_6\backslash a_{15}\}$
a_0 a_8 a_{17} B a_{13} a_{15} $\{A_2\backslash a_8a_{17}D,\ B\backslash Da_3A_4F,\ A_6\backslash Fa_{13}a_{15}\}$

The simple procedure illustrated above does in fact compute all the most general unifiers for two prefix-strings in a way which is considerably more efficient than the one presented in [10]. Among other advantages it generates unifiers step by step instead of computing them all at once.[6] Since it will seldomly be the case that we

[6] There may be up to $\frac{1}{2}\frac{(2n)!}{(n!)^2} \in \mathcal{O}(\frac{2^{2n}}{\sqrt{n}})$ mgu's where n is the depth of the formula tree.

will have to check all possible unifiers during a proof search[7] it leads to a very efficient proof search procedure. A complete description of the unification algorithm and its properties can be found in [12, 13].

4 Conversion into Standard Sequent Proofs

While the connection method is very efficient for *finding* proofs according to the matrix characterization of validity its results cannot directly be used for the construction of programs from the proof. Therefore it is necessary to convert matrix proofs back into sequent proofs which are closer to 'natural' mathematical reasoning. This is comparably easy for classical propositional logic but becomes rather difficult for predicate or intuitionistic logic (see e.g. [16, section 3]) since the reduction ordering $\lhd$ induced by σ_Q and σ_J has to be taken into account.

Fortunately, our proof search procedure described in the previous section already constructs a sequent proof in Fitting's [7] *non-standard* sequent calculus $\mathcal{LJ}_{NS}$. In contrast to standard sequent calculi like Gentzen's $\mathcal{LJ}$ [8] which are used in program development systems it allows the occurrence of more than one formula in the succedent of a sequent. Thus for integrating our procedure into an program development system we only have to convert this $\mathcal{LJ}_{NS}$–proof into a proof within a *standard* sequent calculus.

To understand the differences between these calculi consider the rules shown on the left half of figure 7 where Γ and Δ are *sets* of formulae. When using the two calculi in an analytic manner (i.e. reading the rules from the conclusion to the premises) the different treatment of succedents results in different non-permutabilities of the rules in a sequent proof. The $\neg$–*elim* rule in $\mathcal{LJ}$, for instance, would cause a deletion of the actual succedent formula C and a standard proof could not be finished if C is still relevant. The application of the corresponding $\mathcal{LJ}_{NS}$–rule, however, does not cause any problems. On the other hand the $\neg$–*intro* rule could stop $\mathcal{LJ}_{NS}$–proofs because relevant succedent formulae Δ which are not involved in the reduction itself would be deleted. The corresponding rule in the standard calculus is not dangerous in this sense and does not cause any non-permutabilities in the $\mathcal{LJ}$–proofs.

$$\begin{array}{lll}
\mathcal{LJ}: & \mathcal{LJ}_{NS}: & \mathcal{LJ}^{\star}: \\[6pt]
\dfrac{\Gamma, \neg A \vdash A}{\Gamma, \neg A \vdash C}\ \neg\text{–}elim & \dfrac{\Gamma, \neg A \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\ \neg\text{–}elim & \dfrac{\Gamma, \neg A \vdash A \vee \Delta_S}{\Gamma, \neg A \vdash \Delta_S}\ \neg(\vee)\text{–}elim \\[12pt]
\dfrac{\Gamma, A \vdash}{\Gamma \vdash \neg A}\ \neg\text{–}intro & \dfrac{\Gamma, A \vdash}{\Gamma \vdash \neg A, \Delta}\ \neg\text{–}intro & \dfrac{\dfrac{\Gamma, A \vdash}{\Gamma \vdash \neg A}\ \neg\text{–}intro}{\Gamma \vdash \neg A \vee \Delta_S}\ \vee\text{–}intro\ 1
\end{array}$$

Fig. 7. Example rules of $\mathcal{LJ}$ and $\mathcal{LJ}_{NS}$ and the simulation of $\mathcal{LJ}_{NS}$-rules in $\mathcal{LJ}^{\star}$.

In [17, chapter 2] we have shown that because of the strong differences between the rules of the calculi $\mathcal{LJ}_{NS}$ and $\mathcal{LJ}$ it is not possible to transform every $\mathcal{LJ}_{NS}$–proof into a corresponding $\mathcal{LJ}$–proof without changing the structural information contained in the proof. Thus a transformation of $\mathcal{LJ}_{NS}$–proofs into standard sequent proofs would require an additional search process. To solve this problem we have

[7] This will only be necessary if unification fails in some later step of the proof search.

extended Gentzen's calculus $\mathcal{LJ}$ into an extended standard sequent calculus $\mathcal{LJ}^\star$ which is compatible with $\mathcal{LJ}_{NS}$. This calculus essentially simulates a set of formulae in the succedent by a disjunction of these formulae and thus uses only sequents containing at most one succedent formula. On the other hand it is a simple extension of $\mathcal{LJ}$ since it requires only a few rules in addition to those of the calculus $\mathcal{LJ}$. For example the $\mathcal{LJ}_{NS}$–rule $\neg$–*elim* (figure 7) will be simulated by such a new rule $\neg(\vee)$–*elim* in the calculus $\mathcal{LJ}^\star$ (where Δ_S is the disjunction of the elements contained in Δ). The simulation of the $\neg$–*intro* rule of $\mathcal{LJ}_{NS}$ does not require new rules since it can be expressed by existing $\mathcal{LJ}$–rules as shown in figure 7. All additionally rules which are required to extend $\mathcal{LJ}$ are summarized in figure 8. The only structural rule $\vee$–*change* A_i ensures that it is sufficient to reduce only the leftmost formula of the succedent disjunction Δ_S. For detailed presentation of the development of these rules and its correctness we refer again to [17].

$$\frac{\Gamma \vdash A_i \vee (\Delta_S \setminus\setminus A_i)}{\Gamma \vdash \Delta_S}\ \vee\text{–}change\ A_i$$

$$\frac{\Gamma, \neg A \vdash A \vee (\Delta_S)}{\Gamma, \neg A \vdash \Delta_S}\ \neg(\vee)\text{–}elim \qquad \frac{\Gamma, A \Rightarrow B \vdash A \vee (\Delta_S) \quad \Gamma, B \vdash \Delta_S}{\Gamma, A \Rightarrow B \vdash \Delta_S}\ \Rightarrow(\vee)\text{–}elim$$

$$\frac{\Gamma \vdash A \vee (\Delta_S) \quad \Gamma \vdash B \vee (\Delta_S)}{\Gamma \vdash (A \wedge B) \vee (\Delta_S)}\ \wedge(\vee)\text{–}intro \qquad \frac{\Gamma \vdash A \vee (B \vee (\Delta_S))}{\Gamma \vdash (A \vee B) \vee (\Delta_S)}\ \vee(\vee)\text{–}intro$$

$$\frac{\Gamma \vdash A[x/t] \vee \exists x.A}{\Gamma \vdash \exists x.A}\ \exists(\vee)^*\text{–}intro\ t \qquad \frac{\Gamma \vdash A[x/t] \vee ((\exists x.A) \vee (\Delta_S))}{\Gamma \vdash (\exists x.A) \vee (\Delta_S)}\ \exists(\vee)\text{–}intro\ t$$

Fig. 8. The additionally required rules of $\mathcal{LJ}^\star$

Using the complete rule set of $\mathcal{LJ}^\star$ and the rule mapping procedure guiding the application of these rules (depending on the actual succedent of the $\mathcal{LJ}_{NS}$–proof) the non–permutabilities of each non–standard proof could be simulated exactly in an $\mathcal{LJ}^\star$–proof. Figure 9 shows an informal presentation of the transformation procedure. Its input is an $\mathcal{LJ}_{NS}$–proof of a given formula which is represented as a list of $\mathcal{LJ}_{NS}$ rules.[8] From the output of the procedure we obtain a corresponding $\mathcal{LJ}^\star$–proof, i.e. a list of $\mathcal{LJ}^\star$ rules. The *proof structure* (by which we mean the multi-set of axiom formulae) will not be violated by such a transformation.

Consider, for instance, the following application of the $\Rightarrow$–*elim* rule in $\mathcal{LJ}_{NS}$:

$$\frac{\forall x.A(x) \vee B(x)\ \vdash\ \exists y.A(y),\ \exists x.B(x) \qquad \forall x.A(x) \vee B(x),\ \exists z.\neg A(z)\ \vdash\ \exists x.B(x)}{\forall x.A(x) \vee B(x),\ \exists y.A(y) \Rightarrow \exists z.\neg A(z)\ \vdash\ \exists x.B(x)}$$

In the left subgoal two succedent formulae were generated. We have shown in [16] that these two succedent formulae cannot be simulated within an $\mathcal{LJ}$-proof. Consequently we will have fundamental differences in the resulting proof structures. In contrast to this, our transformation procedure transforms any complete $\mathcal{LJ}_{NS}$–proof of the above sequent into an $\mathcal{LJ}^\star$-proof while preserving the structure of the $\mathcal{LJ}_{NS}$-proof (i.e. using the same instances of the axiom formulae). Figure 10 gives an example of such a proof generated by our procedure.

[8] After branching into two independent subproofs the proof of "left" subgoal precedes the proof of the "right" one in this list. So we can avoid a more complicated list structure for representing the branching structure of the $\mathcal{LJ}_{NS}$–proof.

```
function transform ($\mathcal{LJ}_{NS}$-list, $\mathcal{LJ}^\star$-list)
  let $r$ the head of $\mathcal{LJ}_{NS}$-list and $t$ the tail of $\mathcal{LJ}_{NS}$-list
  let $r'$ the corresponding $\mathcal{LJ}$–rule
  if for the actual succedent [9]$|\Delta| = \emptyset$ then
    append $[r']$ to $\mathcal{LJ}^\star$-list
  else
    if for the actual succedent $|\Delta| = 1$ then append
      $l = \begin{cases} [op(\vee)\text{–}elim], & if\ r = op\text{–}elim,\ op \in \{\Rightarrow, \neg\} \\ [\exists(\vee)^*\text{–}intro], & if\ r = \exists\text{–}intro \\ [\varepsilon^{10}], & if\ r = \vee\text{–}intro \\ [r'], & otherwise \end{cases}$
    to $\mathcal{LJ}^\star$-list
    else
      if for the actual succedent $|\Delta| > 1$ then append
        $l = \begin{cases} [op(\vee)\text{–}elim], & if\ r = op\text{–}elim,\ op \in \{\Rightarrow, \neg\} \\ [\vee\text{–}change^{11}, op(\vee)\text{–}intro], & if\ r = op\text{–}intro,\ op \in \{\exists, \vee, \wedge\} \\ [\vee\text{–}change, \vee\text{–}intro1, r'], & if\ r \in \{\Rightarrow\text{–}intro, \neg\text{–}intro, \forall\text{–}intro, axiom\} \\ [r'], & otherwise \end{cases}$
      to $\mathcal{LJ}^\star$-list
  fi fi fi
  if $t \neq []$ then call transform ($t$,$\mathcal{LJ}^\star$-list) else return $\mathcal{LJ}^\star$-list fi

call transform ($\mathcal{LJ}_{NS}$-list, nil)
```

Fig. 9. The transformation procedure $\mathcal{LJ}_{NS} \longmapsto \mathcal{LJ}^\star$.

The concept of proof structure, the calculus $\mathcal{LJ}^\star$, and the transformation of $\mathcal{LJ}_{NS}$–proofs into $\mathcal{LJ}^\star$–proofs have been investigated in detail in [17, chapter 2]. Altogether we have proven the following properties.

Theorem 2.

1. *The calculus $\mathcal{LJ}^\star$ is a standard calculus which is sound and complete.*
2. *Each $\mathcal{LJ}_{NS}$–proof can be represented in $\mathcal{LJ}^\star$ in a structure-preserving way.*
3. *Each rule of $\mathcal{LJ}^\star$ can be simulated by applying a fixed set of $\mathcal{LJ}$–rules (including the cut).*

Using the proof of theorem 2 we have embedded the calculus $\mathcal{LJ}^\star$ into the NuPRL proof development system [6] by simulating its rules via proof tactics guiding the application of $\mathcal{LJ}$–rules. Furthermore we have implemented a procedure transforming $\mathcal{LJ}_{NS}$–proofs into $\mathcal{LJ}^\star$–proofs which is comparably simple (in contrast to the one presented in [18]) and keeps the size of the resulting proof small. As a consequence we can transform matrix proofs into standard sequent proofs *without any additional search* and integrate our proof search method into larger environments for reasoning about programming and many other kinds of applied mathematics.

[9] ε denotes the *empty* rule which does not affect the actual sequent in the $\mathcal{LJ}^\star$–proof.
[10] The application of the $\vee$–*change* rule additionally requires the formula which has to be changed to the leftmost succedent position. This formula is uniquely determined by the actual $\mathcal{LJ}_{NS}$–rule r and has been omitted here for simplicity.
[11] By this we mean the actual succedent Δ in the $\mathcal{LJ}_{NS}$–proof wrt. to the rule r.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A(a) \vdash A(a)}{A(a) \vdash A(a) \vee (\exists y.A(y) \vee \exists x.B(x))}\;\vee\textit{-intro 1}}{A(a) \vdash \exists y.A(y) \vee \exists x.B(x)}\;\exists(\vee)\textit{-intro a} \qquad \boxed{\textit{Subgoal 1}}}{A(a) \vee B(a) \vdash \exists y.A(y) \vee \exists x.B(x)}\;\vee\textit{-elim}}{\forall x.A(x) \vee B(x) \vdash \exists y.A(y) \vee \exists x.B(x)}\;\forall\textit{-elim a} \qquad \boxed{\textit{Subgoal 2}}}{\forall x.A(x) \vee B(x), \exists y.A(y) \Rightarrow \exists z.\neg A(z) \vdash \exists x.B(x)}\;\Rightarrow(\vee)\textit{-elim}$$

$\boxed{\textit{Subgoal 1:}}$

$$\dfrac{\dfrac{\dfrac{B(a) \vdash B(a)}{B(a) \vdash B(a) \vee (\exists x.B(x) \vee \exists y.A(y))}\;\vee\textit{-intro 1}}{B(a) \vdash \exists x.B(x) \vee \exists y.A(y)}\;\exists(\vee)\textit{-intro a}}{B(a) \vdash \exists y.A(y) \vee \exists x.B(x)}\;\vee\textit{-change } \exists x.B(x)$$

$\boxed{\textit{Subgoal 2:}}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A(a) \vdash A(a)}{A(a) \vdash A(a) \vee \exists x.B(x)}\;\vee\textit{-intro 1} \qquad \dfrac{\dfrac{\dfrac{B(a) \vdash B(a)}{B(a) \vdash B(a) \vee (\exists x.B(x) \vee A(a))}\;\vee\textit{-intro 1}}{B(a) \vdash \exists x.B(x) \vee A(a)}\;\exists(\vee)\textit{-intro a}}{B(a) \vdash A(a) \vee \exists x.B(x)}\;\vee\textit{-change } \exists x.B(x)}{A(a) \vee B(a) \vdash A(a) \vee \exists x.B(x)}\;\vee\textit{-elim}}{\forall x.A(x) \vee B(x) \vdash A(a) \vee \exists x.B(x)}\;\forall\textit{-elim a}}{\forall x.A(x) \vee B(x), \neg A(a) \vdash \exists x.B(x)}\;\neg(\vee)\textit{-elim}}{\forall x.A(x) \vee B(x), \exists z.\neg A(z) \vdash \exists x.B(x)}\;\exists\textit{-elim a}$$

Fig. 10. The resulting $\mathcal{LJ}^{\star}$–proof [12]

5 Conclusion

We have developed an automated proof procedure for intuitionistic logic and a technique for integrating it into proof/program development systems based on the sequent calculus. For this purpose we have extended Bibel's connection method for classical predicate logic [3, 4] into a procedure operating on formulae in non-normal form which is complete for first-order intuitionistic logic and developed an efficient algorithm for the unification of prefix-strings. Furthermore we have designed an extended standard sequent calculus which makes it possible to convert the abstract proof into a proof acceptable for the program development system without any additional search. Our method is currently being realized as a tactic for solving subproblems from first-order logic which arise during a program derivation.

Our work demonstrates that it is possible to make techniques from automated theorem proving directly applicable to program synthesis. By an emphasis on connections and open branches in the proof structure the search space is drastically reduced in comparison with methods based on natural deduction or sequent calculi while a very compact proof representation avoids the notational redundancies contained in them. Since we also construct the skeleton of a sequent proof already during the proof search the transformation of the resulting 'abstract proof' into a humanly comprehensible sequent proof turns out to be comparably easy. Thus our proof-technology combines the strengths of well-known proof search methods (i.e. completeness and efficiency) with those of interactive, tactics supported proof development systems (i.e. safety, flexibility, and expressivity of the underlying theory) and thus extends the deductive power of these systems in a safe and efficient way.

[12] For better overview we omit the *axiom* rules

Although we have based our implementation on the NuPRL proof development system [6] our methodology can also be used to guide other systems based on natural deduction or sequent calculi. Due to the similarity of intuitionistic logic and modal logics it could also be extended to automate reasoning in these logics. Very likely the same proof-technology will be, at least in principle, usable for some subset of linear logic and calculi which describe formal methods in software engineering (although a matrix-characterization for validity still has to be developed). Besides exploring these possibilities our future work will focus on techniques for improving the efficiency of the proof search like the preprocessing steps used in Setheo [9] and KoMeT [2] and the use of typing information during unification. Furthermore we shall investigate how inductive proof methods can be integrated into program synthesis systems by the same technology. All these steps will help a user of a program development system to focus on the key ideas in program design while being freed from having to deal with all the formal details that ensure correctness.

References

1. E. W. Beth. *The foundations of mathematics.* North–Holland, 1959.
2. W. Bibel, S. Brüning, U. Egly, T. Rath. Komet. In *Proceedings of the 12th CADE*, LNAI 814, pp. 783–787. Springer Verlag, 1994.
3. W. Bibel. On matrices with connections. *Jour. of the ACM*, 28, p. 633–645, 1981.
4. W. Bibel. *Automated Theorem Proving.* Vieweg Verlag, 1987.
5. A. Bundy, F. van Harmelen, C. Horn, A. Smaill. The Oyster-Clam system. In *Proceedings of the 10th CADE*, LNCS 449, pp. 647–648. Springer Verlag, 1990.
6. R. L. Constable et. al. *Implementing Mathematics with the NuPRL proof development system.* Prentice Hall, 1986.
7. M. C. Fitting. *Intuitionistic logic, model theory and forcing.* North–Holland, 1969.
8. G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.
9. R. Letz, J. Schumann, S. Bayerl, W. Bibel. Setheo: A high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.
10. H. J. Ohlbach. A resolution calculus for modal logics. Ph.D. Thesis (SEKI Report SR-88-08), Universität Kaiserslautern, 1988.
11. J. Otten, C. Kreitz. A connection based proof method for intuitionistic logic. In *Proceedings of the 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, LNAI 918, pp. 122–137, Springer Verlag, 1995.
12. J. Otten, C. Kreitz. T-String-Unification: Unifying Prefixes in Non-Classical Proof Methods. Report AIDA-95-09, FG Intellektik, TH Darmstadt, 1995.
13. J. Otten. Ein konnektionenorientiertes Beweisverfahren für intuitionistische Logik. Master's thesis, TH Darmstadt, 1995.
14. L. Paulson. Isabelle: The next 700 theorem provers. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pp. 361–386. Academic Press, 1990.
15. R. Pollack. *The theory of LEGO – a proof checker for the extendend calculus of constructions.* PhD thesis, University of Edinburgh, 1994.
16. S. Schmitt, C. Kreitz. On transforming intuitionistic matrix proofs into standard-sequent proofs. In *Proceedings of the 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, LNAI 918, pp. 106–121, Springer, 1995.
17. S. Schmitt. Ein erweiterter intuitionistischer Sequenzenkalkül und dessen Anwendung im intuitionistischen Konnektionsbeweisen. Master's thesis, TH Darmstadt, 1994.
18. G. Takeuti. *Proof Theory.* North–Holland, 1975.
19. L. Wallen. *Automated deduction in nonclassical logic.* MIT Press, 1990.
20. L. Wos et. al. Automated reasoning contributes to mathematics and logic. In *Proceedings of the 10th CADE*, LNCS 449, p. 485–499. Springer Verlag 1990.