Matrix-based Constructive Theorem Proving

Christoph Kreitz   Jens Otten   Stephan Schmitt   Brigitte Pientka

Department of Computer Science, Cornell-University
Ithaca, NY 14853-7501, U.S.A.
{kreitz,jeotten,steph,pientka}@cs.cornell.edu

Abstract. Bibel's connection method, originally developed for classical logic, has been extended to a variety of non-classical logics and used to control the creation of sequent proofs for interactive proof assistants. Matrix methods for intuitionistic logic also are the central inference engine for program synthesis and verification. We present a coherent account of matrix methods for constructive theorem proving and show how to extend them to inductive theorem proving by integrating rippling techniques into the unification process.

1 Introduction

Formal methods for program verification, optimization, and synthesis rely on complex mathematical proofs, which often involve reasoning about computations. Because of that there is no single automated proof procedure that can handle all the reasoning problems occurring during a program derivation or verification. Instead, one usually relies on proof assistants like NuPRL [9], Coq [10], Alf [1] etc., which are based on very expressive logical calculi and support interactive and tactic controlled proof and program development. Proof assistants, however, suffer from a very low degree of automation, since all their inferences must eventually be based on sequent or natural deduction rules. Even proof parts that rely entirely on predicate logic can seldomly be found automatically, as there are no complete proof search procedures embedded into these systems. It is therefore desirable to extend the reasoning power of proof assistants by integrating well-understood techniques from automated theorem proving.

Matrix-based proof search procedures [3, 4] can be understood as compact representations of tableaux or sequent proof techniques. They avoid the usual redundancies contained in these calculi and are driven by *complementary connections*, i.e. pairs of atomic formulae that may become leaves in a sequent proof, instead of the logical connectives of a proof goal. Although originally developed for classical logic, the *connection method* has recently been extended to a variety of non-classical logics such as intuitionistic logic [18], modal logics [20], and fragments of linear logic [14, 17]. Furthermore, algorithms for converting matrix proofs into sequent proofs have been developed [23, 24], which makes it possible to view matrix proofs as plans for predicate logic proofs that can be executed within a proof assistant [6, 15].

Viewing matrix proofs as proof plans also suggests the integration of additional proof planning techniques into the connection method. Rewrite techniques such as rippling [8], for instance, have successfully been used as proof planners for inductive theorem proving but are relatively weak as far as predicate logic reasoning is concerned. A recent extension [21] has demonstrated that rippling techniques and logical proof search methods can be combined and used successfully for constructive theorem proving and the synthesis of inductive programs.

In this paper we will present a coherent account of matrix methods for constructive theorem proving and show how to extend them further by integrating rippling techniques into the unification process. We will first present a non-clausal extension of Bibel's original connection method in Section 2 and adapt it to constructive logic in Section 3. Section 4 describes the conversion of matrix proofs into sequent proofs. In Section 5 we discuss the integration of rippling techniques into matrix methods. We conclude with a discussion of possible applications of our work in program synthesis and verification.

2 The Connection Method for Non-normal Form

The connection method [3, 4] was originally designed as proof search method for formulae in clause normal form. But as normalization of formulae is often costly and as many non-classical logics do not have normal forms, it is necessary to develop matrix methods for formulae in non-normal form. Bibel [4] already describes a non-clausal version of his connection method. The version that we will present here is more general. It is based on Wallen's matrix characterizations of logical validity [25] and can easily be adapted to a variety of logics.

Since matrix proofs can be viewed as compact representations of analytic tableaux, many notions from tableaux calculi carry over to matrix methods. The main difference is that tableaux proofs are based on rules that decompose a formula, generate subformulae, and eventually close off proof branches, while matrix methods operate directly on the formula tree and search for connections, i.e. pairs of identical literals with different polarities that could close a branch in a tableaux proof. In this section we will first introduce the basic concepts used in matrix methods, characterize logical validity in terms of these concepts, and then develop the proof procedure on the basis of the matrix characterization.

A *formula tree* is the representation of a formula F as a tree. Each node corresponds to exactly one subformula F_s of F and is marked with a unique name $a_0, a_1, \ldots$, its *position*. The *label* of a position u denotes the major connective of the F_s or the formula F_s, if it is atomic. In the latter case, u is called an *atomic position* and can also be identified by its label. The *tree ordering* $<$ of F is the partial ordering on the positions in the formula tree.

Each position in a formula tree is associated with a label, a polarity and a principal type. The *polarity* (0 or 1) of a position is determined by the label and polarity of its parent. The root position has polarity 0. The *principal type* of a position is determined by its polarity and its label. Atomic positions have no principal type. Polarities and types of positions are defined in the table below.

For example, a position labeled with $\Rightarrow$ and polarity 1 has principal type β and its successor positions have polarity 0 and 1, respectively. For a given formula we denote the sets of positions of type α, β, γ, and δ by α, β, Γ, and Δ.

principal type α	$(A \wedge B)^1$	$(A \vee B)^0$	$(A \Rightarrow B)^0$	$(\neg A)^1$	$(\neg A)^0$
successor polarity	A^1, B^1	A^0, B^0	A^1, B^0	A^0	A^1
principal type β	$(A \wedge B)^0$	$(A \vee B)^1$	$(A \Rightarrow B)^1$		
successor polarity	A^0, B^0	A^1, B^1	A^0, B^1		
principal type γ	$(\forall x A)^1$	$(\exists x A)^0$			
successor polarity	A^1	A^0			
principal type δ	$(\forall x A)^0$	$(\exists x A)^1$			
successor polarity	A^0	A^1			

A quantifier *multiplicity* $\mu_Q{:}\Gamma \to \mathbb{N}$ (briefly μ) encodes the number of distinct instances of γ-subformulae that need to be considered during the proof search. By F^μ we denote an *indexed formula*, i.e. a formula and its multiplicity. We consider multiple instances of subformulae according to the multiplicity of its corresponding position in the formula tree and extend the tree ordering accordingly. For technical reasons we use quantifier positions, i.e. γ- and δ-positions, as names for free variables in atomic formulae. Figure 1 shows the formula tree, marked with polarities, principal types, and a multiplicity, for the formula $F_1 \equiv \forall x Sx \wedge \forall y(\neg(Ty \Rightarrow Ry) \Rightarrow Py) \;\Rightarrow\; \neg\exists z((Pz \Rightarrow Qz) \wedge (Tz \Rightarrow Rz)) \;\Rightarrow\; \neg\neg Pa \wedge Sa \wedge Sb$

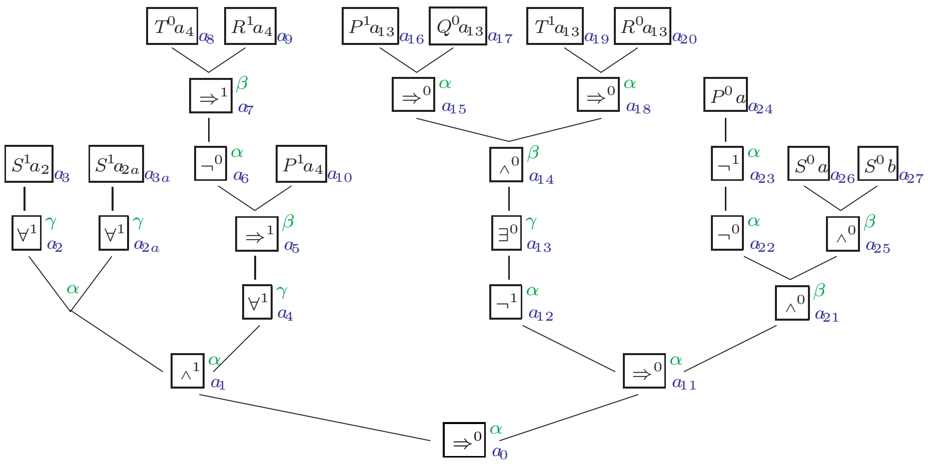


Fig. 1. Formula tree for F_1^μ with $\mu(a_2)$=2

The *matrix(-representation)* of a formula F is a two-dimensional representation of the atomic formulae in F without connectives and quantifiers and better suited for illustration purposes. In a matrix-representation α-related positions appear side by side and β-related positions appear on top of each other, where two positions u and v are *α-related* (or *β-related*), denoted $u{\sim_\alpha}v$ ($u{\sim_\beta}v$), iff $u{\neq}v$ and the greatest common ancestor of u and v, w.r.t. the tree ordering $<$, is of principal type α (or β). u is *α-related* (*β-related*) to a set S of positions if $u{\sim_\alpha}v$ ($u{\sim_\beta}v$) for all $v \in S$. The matrix representation of F_1^μ is given in Figure 2.

$$\left[\begin{array}{cccc} S^1a_2 & S^1a_{2a} & \begin{bmatrix} T^0a_4 \\ R^1a_4 \\ P^1a_4 \end{bmatrix} & \left[\begin{bmatrix} \begin{bmatrix} P^1a_{13} & Q^0a_{13} \end{bmatrix} \\ \begin{bmatrix} T^1a_{13} & R^0a_{13} \end{bmatrix} \end{bmatrix} \begin{bmatrix} S^0b \\ S^0a \\ P^0a \end{bmatrix}\right] \end{array}\right]$$

Fig. 2. Matrix of the formula F_1

The matrix-characterizations of logical validity [3, 4, 25] depend on the concepts of paths, connections, and complementarity. A *path* through a formula F is a set of mutually α-related atomic positions of its formula tree that describes a maximal horizontal path through the matrix representation of F. A *connection* is a pair of atomic positions labeled with the same predicate symbol but with different polarities. A connection is *complementary*, if its atomic formulae are unifiable by an admissible substitution.

The precise definition of complementarity depends on the logic under consideration. For classical logic, we only need to consider quantifier- or first-order substitutions. A *(first-oder) substitution* σ_Q (briefly σ) is a mapping from positions of type γ to terms. It induces a relation $\sqsubset_Q \subseteq \Delta \times \Gamma$ in the following way: if $\sigma_Q(u) = t$, then $v \sqsubset_Q u$ for all $v \in \Delta$ occurring in t.

Definition 1 (Complementarity in Classical Logic).

- A first-oder substitution σ_Q is *admissible* with respect to F^μ iff the induced *reduction ordering* $\lhd := (<^\mu \cup \sqsubset_Q)^+$ is irreflexive.
- A connection $\{u, v\}$ is σ_Q*-complementary* iff $\sigma_Q(\mathit{label}(u)) = \sigma_Q(\mathit{label}(v))$.

As paths through matrices correspond to branches of tableaux proofs and complementary connections to closing branches, a formula F is valid if every path through some F^μ contains a complementary connection.

Theorem 1 (Matrix Characterization for Classical Logic [4]). *A formula F is (classically) valid iff there is a multiplicity μ, an admissible substitution σ and a set of σ-complementary connections such that every path through F^μ contains a connection from this set.*

Example 1. In the matrix for F_1^μ in Figure 2 there are 18 paths (note that P^1a_{13} and R^0a_{13} are *not* α-related). Each path contains one of the connections $\{S^1a_2, S^0b\}$, $\{S^1a_{2a}, S^0a\}$, $\{T^0a_4, T^1a_{13}\}$, $\{R^1a_4, R^0a_{13}\}$, $\{P^1a_4, P^0a\}$, and $\{P^1a_{13}, P^0a\}$. The six connections are complementary under the first-order substitution $\sigma_Q = \{a_2 \backslash b, a_{2a} \backslash a, a_4 \backslash a, a_{13} \backslash a\}$. As no δ-positions occur in σ_Q, the induced reduction ordering $\lhd$ is the tree ordering $<$ and irreflexive. Therefore σ_Q is admissible and F_1 is valid.

According to the above characterization the validity of a formula F can be proven by showing that all paths through the matrix representation of F^μ contain a complementary connection. Obviously it is not very efficient to check all possible paths for complementary connections. Instead, a path checking algorithm should be driven by the connections: once a complementary connection has been identified all paths containing this connection can be eliminated from further consideration. This technique is similar to Bibel's connection method for classical logic [4], but our algorithm is more general and useful for proof search in various non-classical logics.

The key notions of our path checking algorithm are active paths, active subgoals, and open subgoals. The *active path* $\mathcal{P}$ specifies those paths that are currently investigated for complementarity. All paths that contain $\mathcal{P}$ and an element u of the *active subgoal* $\mathcal{C}$ have already been proven to contain a complementary connection. All paths that contain $\mathcal{P}$ and an element v of the *open subgoal* $\mathcal{C}'$ must still be tested for complementarity. If the latter can be proven complementary as well then all paths containing the active path are complementary. The algorithm will recursively check whether all paths containing the empty active path are complementary.

Let $\mathcal{A}$ denote the set of all atomic positions in the formula F. A *subpath* $\mathcal{P}$ is a set of mutually α-related atomic positions, i.e. a (not necessarily maximal) horizontal path through a matrix. A subpath $\mathcal{P}$ is a path iff there is no $u \in \mathcal{A}$ with $u \sim_\alpha \mathcal{P}$. A *subgoal* $\mathcal{C}$ is a set[1] of mutually β-related atomic positions. During proof search certain tuples $(\mathcal{P},\mathcal{C})$ consisting of a subpath $\mathcal{P}$ and a subgoal $\mathcal{C}$ with $u \sim_\alpha \mathcal{P}$ for all $u \in \mathcal{C}$ will be called *active goals*. $\mathcal{P}$ will be the *active path* and $\mathcal{C}$ the *active subgoal*. An *open subgoal* $\mathcal{C}' \subseteq \mathcal{A}$ with respect to an active goal $(\mathcal{P},\mathcal{C})$ is a maximal set of atomic positions, such that $u \sim_\alpha \mathcal{P}$ and $u \sim_\beta \mathcal{C}$ for all $u \in \mathcal{C}'$.

Example 2. In Figure 2 the sets $\mathcal{P}_1 = \{S^1 a_2, S^1 a_{2a}, T^1 a_{13}, P^0 a\}$, $\mathcal{P}_2 = \{S^1 a_2, S^1 a_{2a}, R^1 a_4, S^0 b\}$, and $\mathcal{P}_3 = \{T^0 a_4, T^1 a_{13}, R^0 a_{13}\}$ are subpaths for the formula F_1. $\mathcal{C}_1 = \{T^0 a_4\}, \mathcal{C}_2 = \{Q^0 a_{13}, T^1 a_{13}\}$, and $\mathcal{C}_3 = \{S^1 a_2\}$ are subgoals. $(\mathcal{P}_1,\mathcal{C}_1)$, $(\mathcal{P}_2,\mathcal{C}_2)$, and $(\mathcal{P}_3,\mathcal{C}_3)$ are active goals. $\{R^1 a_4, P^1 a_4\}$ is an open subgoal w.r.t. the active goal $(\mathcal{P}_1,\mathcal{C}_1)$; the empty set $\emptyset$ is the only subgoal w.r.t. $(\mathcal{P}_2,\mathcal{C}_2)$ or $(\mathcal{P}_3,\mathcal{C}_3)$.

We call an active goal $(\mathcal{P},\mathcal{C})$ *provable* with respect to a formula F if there is an open subgoal $\mathcal{C}'$ w.r.t. $(\mathcal{P},\mathcal{C})$, such that for all $u \in \mathcal{C}'$ all paths $\mathcal{P}'$ through F with $P \cup \{u\} \subseteq \mathcal{P}'$ are complementary. This definition leads to a more algorithmic characterization of validity.

Theorem 2. *A formula F is valid iff there is a multiplicity μ and an admissible substitution σ such that the active goal $(\emptyset,\emptyset)$ w.r.t. F^μ is provable.*

The above characterization holds uniformly for a variety of logics and leads to a general path-checking algorithm that, coupled with an appropriate definition of complementarity, can be used as proof search procedure for all these logics. The path checking algorithm is implicitly described by the following theorem, which gives sufficient and necessary conditions for provability.

Theorem 3. *An active goal $(\mathcal{P},\mathcal{C})$ is provable iff*

1. *there is no atomic position u with $u \sim_\alpha \mathcal{P}$ and $u \sim_\beta \mathcal{C}$, or*
2. *there is a complementary connection $\{A, \bar{A}\}$ with $A \sim_\alpha \mathcal{P}$ and $A \sim_\beta \mathcal{C}$, such that the active goal $(\mathcal{P}, \mathcal{C} \cup \{A\})$ is provable and*
   $\bar{A} \in \mathcal{P}$ or $\bar{A} \sim_\alpha (\mathcal{P} \cup \{A\})$ and the active goal $(\mathcal{P} \cup \{A\}, \{\bar{A}\})$ is provable.

[1] To us, a subgoal either describes *all* the positions that have already been investigated by the algorithm or all the positions that still need to be investigated.

```
letrec prove(F,n) =
  let σ,μ = initialize(F,n)  in
   let CON = connections(F^μ)  in
    letrec provable(P,C,σ) =
       letrec check-connections(D, A, σ) =
         let Ā∈D  in                                        (* may fail *)
           (let σ1 = unify-check(A, Ā, F^μ, σ)  in
              let σ2 = provable(P, C∪{A}, σ1)  in
                if Ā∈P  then σ2   else provable(P∪{A}, {Ā}, σ2)
           ) ? check-connections(D−{Ā}, A, σ)                (* next Ā *)
       in
       letrec check-extension(E, σ) =
         let A∈E  in                                        (* may fail *)
          (let D = {Ā∈A | {A, Ā}∈CON ∧ (Ā∈P ∨ Ā∼α(P∪{A}))}  in
              check-connections(D, A, σ)
          ) ? check-extension({u∈E|u∼αA}, σ)                (* next A *)
       in
       let E = {A∈A | A∼αP ∧ A∼βC}  in
         if E = ∅  then σ   else check-extension(E, σ)
    in
       provable(∅, ∅, σ)  ?  prove(F,n+1)
 in
   prove(F,1)
```

Fig. 3. Uniform path checking algorithm

Figure 3 describes a simple uniform path-checking algorithm based on the above theorems. The function `provable` checks the provability of an active goal $(\mathcal{P},\mathcal{C})$ under an already computed substitution σ. It returns a new substitution, if it succeeds, and fails otherwise (`?` denotes failure catching). The variable $\mathcal{E}$ describes the positions A which may be used to extend the active path. If $\mathcal{E}$ is empty, then `provable` succeeds because of Theorem 3.1. Otherwise `provable` recursively checks all possible values for A (in $\mathcal{E}$) and $\bar{A}$ (in $\mathcal{D}$) according to Theorem 3.2. The function `prove` iterates the multiplicity μ, computes all possible connections in F^μ and checks provability according to Theorem 2. It is initialized with multiplicity 1.

The algorithm is parameterized with two functions, which express the specific properties of the logic under consideration. The function `initialize` determines the initial value for the substitution σ and the multiplicity μ while `unify-check`$(A, \bar{A}, F^\mu, \sigma)$ tries compute a substitution that unifies A and $\bar{A}$, extends σ, and leads to an acyclic reduction ordering in F^μ.

For classical logic `initialize`(F,n) computes a pair (σ, μ) with $\sigma = \emptyset$ and $\mu(u)=n$ for all $u \in \Gamma$. `unify-check`$((A, \bar{A}, F^\mu, \sigma)$ computes a most general (term) unifier σ_Q of $\sigma(\mathit{label}(A))$ and $\sigma(\mathit{label}(\bar{A}))$, as well as the induced reduction ordering $\lhd := (< \cup \sqsubset_Q)^+$. It returns σ_Q if $\lhd$ is irreflexive and fails otherwise or if the two atoms cannot be unified.

Because of the stepwise increase of the multiplicity the path checking mechanism described in Figure 3 is obviously not very efficient and also not able to decide that a given first-order formula is invalid. In an efficient implementation the multiplicity for each suitable position is determined dynamically *during* the path-checking process. Other techniques that were used in theorem provers based on the usual connection method can be applied as well.

3 Proving Theorems in Constructive Logic

As program synthesis and verification often relies on constructive arguments proof assistants should be supported by automatic proof search procedures for intuitionistic logic. Independently from the philosophical differences between classical and intuitionistic logic the main distinction between these two logics can be expressed by a different treatment of $\forall$, $\Rightarrow$, and $\neg$. Whereas in the classical sequent calculus all rules are permutable (i.e. they do not destroy information), the intuitionistic sequent rules for $\forall$, $\Rightarrow$, and $\neg$ in the succedent are not permutable.

A matrix method for intuitionistic logic must therefore not only check if two connected atomic formulae can be unified but also if they can both be reached by applying the same sequence of sequent rules. Only then they form a leaf in a sequent proof. In the matrix characterization this is reflected by an additional *intuitionistic substitution* σ_J, which has to make the *prefixes* of the connected positions identical, where a prefix of a position u is a string consisting of variables and constants which essentially describes the location of u in the formula tree.

For this purpose we introduce an *intuitionistic type* of a position labelled with atoms, $\forall$, $\Rightarrow$, and $\neg$ according to the following table.

intuitionistic type ϕ	$(\neg A)^1$	$(A \Rightarrow B)^1$	$(\forall x A)^1$	P^1 (P atomic)
intuitionistic type ψ	$(\neg A)^0$	$(A \Rightarrow B)^0$	$(\forall x A)^0$	P^0 (P atomic)

Positions of type ψ correspond to the application of non-permutable sequent rules and are viewed as constants in a prefix string while ϕ-positions are variables. This makes it possible to use unification to determine the ψ-positions that must be reduced before a ϕ position in a valid sequent proof[2] and to develop a matrix-characterization for intuitionistic validity whose formulation is almost identical to the one for classical logic.

The *prefix* $\mathrm{pre}(u)$ of an atomic position u is a string $u_1 u_2 \ldots u_n$ where $u_1 {<} u_2 {<} \ldots {<} u_n {=} u$ are the elements of $\Psi \cup \Phi$ that dominate u in the formula tree. An *intuitionistic substitution* σ_J is a mapping from positions of type ϕ to (possibly empty) strings over Ψ. It induces a relation $\sqsubset_J \subseteq \Psi \times \Phi$ in the following way: if $\sigma_J(u){=}p$, then $v \sqsubset_J u$ for all characters v occurring in p.

Definition 2 (Complementarity in Intuitionistic Logic).
Let $\sigma{:=}(\sigma_Q, \sigma_J)$ be a *combined substitution* consisting of a first-order substitution σ_Q and an intuitionistic substitution σ_J.

1. σ is *J-admissible* iff the induced *reduction ordering* $\lhd{:=}({<} \cup \sqsubset_Q \cup \sqsubset_J)^+$ is irreflexive and $|\sigma_J(\mathrm{pre}(v))| \leq |\sigma_J(\mathrm{pre}(u))|$ holds for all $v \in \Delta$ with $v \in \sigma_Q(u)$ and all $u \in \Gamma$.
2. A connection $\{u, v\}$ is σ*-complementary* iff $\sigma_Q(\mathit{label}(u)){=}\sigma_Q(\mathit{label}(v))$ and $\sigma_J(\mathrm{pre}(u)){=}\sigma_J(\mathrm{pre}(v))$.

[2] This methodology is inspired by the admissibility condition for first-order substitutions in Definition 1, where $\sqsubset_Q$ determines which δ-positions must be reduced before certain γ-positions in order to satisfy the eigenvariable-condition.

Formulae of type ϕ can be used in a proof several times in different ways. An *intuitionistic multiplicity* μ_J:$\Phi \rightarrow \mathbb{N}$ encodes the number of distinct instances of ϕ-subformulae that need to be considered during the proof search. It can be combined with a quantifier multiplicity μ_Q and leads to an indexed formula F^μ.

Theorem 4 (Matrix Characterization for Intuitionistic Logic [25]). *A formula F is intuitionistically valid iff there is a multiplicity $\mu := (\mu_Q, \mu_J)$, a J-admissible combined substitution $\sigma = (\sigma_Q, \sigma_J)$, and a set of σ-complementary connections such that every path through F^μ contains a connection from this set.*

Example 3. For our formula F_1^μ from Figure 1 we have $\mu := (\mu_Q, \mu_J)$ with $\mu_Q(a_2) = 2$, $\mu_Q(u) = 1$ otherwise and $\mu_J \equiv 1$.

The ψ-positions are $\{a_0, a_6, a_7, a_{11}, a_{15}, a_{17}, a_{18}, a_{20}, a_{22}, a_{24}, a_{26}, a_{27}\}$, while $\{a_2, a_{2a}, a_3, a_{3a}, a_4, a_5, a_9, a_{10}, a_{12}, a_{16}, a_{19}, a_{23}\}$ is the set of ϕ-positions, which in the following will be indicated by capital letters. ε denotes the empty string.

For the six connections which are used to show the classical validity of F_1 in Example 1 we have $\mathrm{pre}(A_3){=}a_0A_2A_3$, $\mathrm{pre}(a_{27}){=}a_0a_{11}a_{27}$, $\mathrm{pre}(A_{3a}){=}a_0A_{2a}A_{3a}$, $\mathrm{pre}(a_{26}){=}a_0a_{11}a_{26}$, $\mathrm{pre}(a_8){=}a_0A_4A_5a_6A_7a_8$, $\mathrm{pre}(A_{19}){=}a_0a_{11}A_{12}a_{18}A_{19}$, $\mathrm{pre}(A_9){=}a_0A_4A_5a_6A_7A_9$, $\mathrm{pre}(a_{20}){=}a_0a_{11}A_{12}a_{18}a_{20}$, $\mathrm{pre}(A_{10}){=}a_0A_4A_5A_{10}$, $\mathrm{pre}(a_{24}){=}a_0a_{11}a_{22}A_{23}a_{24}$, and $\mathrm{pre}(A_{16}){=}a_0a_{11}A_{12}a_{15}A_{16}$.

The prefixes of the six connections can be unified with $\sigma_J = \{A_2\backslash\varepsilon,\ A_{2a}\backslash\varepsilon,\ A_3\backslash a_{11}a_{27},\ A_{3a}\backslash a_{11}a_{26},\ A_4\backslash\varepsilon,\ A_5\backslash a_{11}a_{22},\ A_7\backslash a_{18}a_{20},\ A_9\backslash\varepsilon,\ A_{10}\backslash a_6a_{15}a_{24},\ A_{12}\backslash a_{22}a_6,\ A_{16}\backslash a_{24},\ A_{19}\backslash a_{20}a_8,\ A_{23}\backslash a_6a_{15}\}$ while the terms can be unified with $\sigma_Q = \{a_2\backslash b,\ a_{2a}\backslash a,\ a_4\backslash a,\ a_{13}\backslash a\}$.

The combined substitution $\sigma = (\sigma_Q, \sigma_J)$ is J-admissible, as the induced reduction ordering $\lhd{:=}(<\cup\sqsubset_Q\cup\sqsubset_J)^+$ is irreflexive and no δ-positions occur in σ_Q. Thus F_1 is intuitionistically valid.

Theorems 2 and 3 hold accordingly with the intuitionistic definitions of complementarity and multiplicity. Therefore our path checking algorithm presented in Figure 3 can be used for intuitionistic logic as well. We only have to provide the logic-specific functions `initialize` and `unify-check`.

For intuitionistic logic `initialize`(F,n) computes a pair (σ, μ) where $\sigma = (\emptyset, \emptyset)$ is a combined substitution and $\mu(u){=}n$ for all $u \in \Gamma \cup \Phi$. The function `unify-check`$((A, \bar{A}, F^\mu, \sigma)$ with $\sigma = (\sigma_Q, \sigma_J)$ computes a most general term unifier σ_{Q_1} of $\sigma_Q(\mathit{label}(A))$ and $\sigma_Q(\mathit{label}(\bar{A}))$ as well as a most general prefix-unifier σ_{J1} of $\sigma_J(\mathrm{pre}(u))$ and $\sigma_J(\mathrm{pre}(v))$. It returns $\sigma_1 := (\sigma_{Q_1}, \sigma_{J1})$ if σ_1 is *J-admissible* and fails otherwise or if either of the two unifications fails.

To compute the intuitionistic substitution we apply a specialized string unification algorithm [19]. String unification in general is quite complicated but prefixes are a very restricted class of strings. Prefixes are strings without duplicates. In two prefixes p and q, corresponding to atoms of the same formula, equal symbols can only occur within a common substring at the beginning of p and q. These restrictions enable us to use a much more efficient algorithm for computing a minimal set of most general unifiers.

The unification algorithm is based on a series of transformation rules that are repeatedly applied to a singleton set of prefix equations $\mathcal{EQ} = \{p{=}q\}$ and an empty substitution $\sigma_J = \emptyset$. The procedure stops if $\mathcal{EQ}$ is empty and returns the resulting substitution σ_J as most general unifier. As the transformation rules can be applied nondeterministically, the set of most general unifiers consists of the results of all successfully finished transformations.

Our unification algorithm is parameterized by a set of transformation rules in order to be adaptable to a variety of logics. For intuitionistic prefix unification the peculiarities of the logic are expressed by a set of 10 transformation rules (see [19]). These rules enable us to compute a single most general unifier in linear time. Furthermore, a parallel application of the rules to a set of prefix equations allows us to decide in quadratic time if there is *no* general unifier.

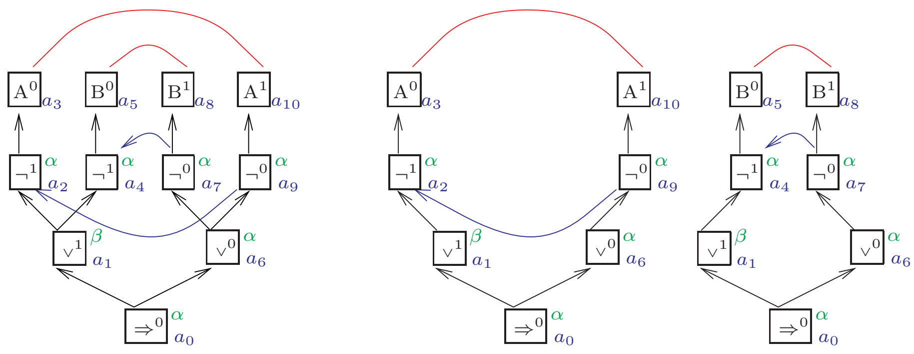
4 Reconstructing Sequent Proofs from Matrix Proofs

While matrix methods are very efficient for proving the validity of a given formula they cannot directly be used within sequent or natural-deduction based proof assistants. Although in principle it would be possible to embed the connection method as *trusted external refiner*, a technique supported by the upcoming release 5 of the NuPRL proof development system [9], a matrix proof can only be used to establish the *truth* of a given formula. If, however, the formula is a part of a program derivation, one must be able to extract a piece of code from the proof, which according to the *proofs-as-program paradigm* [2], is essentially the same as providing a constructive sequent proof. In order to integrate matrix methods into proof assistants it is therefore necessary to *reconstruct* a sequent proof from a given matrix proof.

As matrix-proofs are compact representations of sequent proofs, converting matrix methods into sequent proofs means re-introducing the redundancies that had been avoided during proof search. Obviously, the conversion should be performed without additional search since otherwise reconstructing the sequent proof would be as difficult as finding the proof in the first place.

The conversion algorithm that we will describe in this section complements the matrix-based proof procedures presented above. It is equally uniform in its design and consults tables whose data represent the peculiarities of various classical and non-classical logics with respect to matrix and sequent calculi. Basically, it traverses the formula tree of the formula F^μ in an order that respects the induced reduction ordering $\lhd{=}(<\cup\sqsubset_Q\cup\sqsubset_J)^+$ generated during proof search. It selects the appropriate sequent rule for each visited node and instantiates quantifiers according to the substitution σ_Q.

As $\lhd$ does not completely encode all the non-permutabilities of sequent rules, we have to add *wait*-labels dynamically during the conversion process. These labels prevent non-invertible sequent rules from being applied too early, which means that no proof relevant formulae are deleted prematurely. At β-nodes, which cause a sequent proof to branch, the reduction ordering has to be divided appropriately and certain redundancies (e.g. *wait*-labels) need to be eliminated in order to ensure completeness. We explain our method by a small example.



$\lhd$ $\qquad$ $\lhd_1$ $\qquad$ $\lhd_2$

Consider the formula $F_2 \equiv \neg A \vee \neg B \Rightarrow \neg B \vee \neg A$ and its matrix proof represented by the reduction ordering $\lhd$, which consists of the formula tree of F_2 (straight arrows), the induced relation $\sqsubset_J$ (curved arrows), and the connections between atoms (curved lines). After reducing the β-position a_1 the reduction ordering is split into two suborderings $\lhd_1$, $\lhd_2$ and the conversion continues separately on each of the suborderings. To guarantee completeness, the operation *split* not only splits the reduction ordering $\lhd$ but also removes positions and arrows from each $\lhd_i$ (i.e. $\{a_7, a_8\}$ from $\lhd_1$ and $\{a_9.a_{10}\}$ from $\lhd_2$) which are no longer relevant for the corresponding branch of the sequent proof. The result of the splitting process is shown on the right hand of the above diagram.

Proof reconstruction traverses the formula tree of F as follows

1. Select a position u from the set P_o of *open positions* that is not "blocked" by some arrow in $\lhd$.
2. Select a sequent rule according to the polarity and the label of u. If necessary, instantiate a variable according to σ_Q.
3. Mark u as visited, remove it from P_o and add instead its immediate successor position(s) to P_o.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,ax.}{\neg A, A \vdash}\,\neg l}{\neg A \vdash \neg B, \neg A}\,\neg r \qquad \dfrac{\dfrac{\dfrac{}{B \vdash B}\,ax.}{\neg B, B \vdash}\,\neg l}{\neg B \vdash \neg B, \neg A}\,\neg r}{\neg A \vee \neg B \vdash \neg B, \neg A}\,\vee l}{\neg A \vee \neg B \vdash \neg B \vee \neg A}\,\vee r}{\vdash \neg A \vee \neg B \Rightarrow \neg B \vee \neg A}\,\Rightarrow r
$$

	P_o	u	rule	applied to
$\lhd$	$\{a_0\}$	a_0	$\Rightarrow r$	$(\neg A \vee \neg B \Rightarrow \neg B \vee \neg A)^0$
	$\{a_1, a_6\}$	a_6	$\vee r$	$(\neg B \vee \neg A)^0$
	$\{a_1, a_7, a_9\}$	a_1	$\vee l$	$(\neg A \vee \neg B)^1$
$\lhd_1$	$\{a_2, a_9\}$	a_9	$\neg r$	$\neg A^0$
	$\{a_2, a_{10}\}$	a_{10}	—	A^1
	$\{a_2\}$	a_2	$\neg l$	$\neg A^1$
	$\{a_3\}$	a_3	$ax.$	A^0, A^1
$\lhd_2$	$\{a_4, a_7\}$	a_7	$\neg r$	$\neg B^0$
	$\{a_4, a_8\}$	a_8	—	B^1
	$\{a_4\}$	a_4	$\neg l$	$\neg B^1$
	$\{a_5\}$	a_5	$ax.$	B^0, B^1

Fig. 4. Sequent proof for F_2 and corresponding traversal steps of $\lhd$

The traversal process and the resulting sequent proof for F_2 are depicted in Figure 4. Note that starting the traversal with a_0, a_6, a_7 instead of a_0, a_6, a_1 is not prevented by “blocking” arrows in $\lhd$. This choice, however, would lead to a partial sequent proof that cannot be completed. Reducing a_7, i.e. applying $\neg r$ on $\neg B^0$, deletes the relevant formula $\neg A^0$ (position a_9). For this reason, our conversion algorithm adds two *wait*-labels dynamically to a_9 and a_7 and avoids a deadlock during traversal.

The technical details of our conversion procedure and the efficient elimination of redundancies after β-splits are quite subtle. An extensive discussion and algorithmic description can be found in [24, 22].

5 Integrating Induction Techniques

The procedure for converting matrix proofs into sequent proofs suggests viewing the connection method as a proof planner for first-order logic that can be used to extend the reasoning capabilities of proof assistants by fully automated proof procedures. As formal reasoning about programs often requires inductive arguments the same methodology should be applied to integrate techniques from inductive theorem proving as well. Here, an annotated rewrite technique, called rippling [8], has been used successfully to plan the reasoning steps from the induction hypothesis to the induction conclusion. This technique, however, shows certain weaknesses when dealing with program synthesis through inductive proofs, since in this case existentially quantified variables need to be instantiated. In fact, many program derivations require both first-order and inductive proof methods to be applied simultaneously, since both techniques are not strong enough to solve the problem independently.

Example 4. The formula $\forall x\, \exists y\; y^2{\le}x \wedge x{<}(y{+}1)^2$ specifies an algorithm for computing the integer square root of a natural number x. A (top-down) proof of this formula will proceed by induction and lead to the following two subgoals

(1) $\vdash \exists y\; y^2{\le}0 \wedge 0{<}(y{+}1)^2$, and

(2) $\exists y\; y^2{\le}x \wedge x{<}(y{+}1)^2 \;\vdash\; \exists y\; y^2{\le}x{+}1 \wedge x{+}1{<}(y{+}1)^2$

While the base case of the induction (1) can be solved by standard arithmetical reasoning about zero, no single rippling sequence will be able to rewrite the conclusion of the step case (2) into the induction hypothesis, as the choice of y in the conclusion (y_c) strongly depends on the properties of the y in the hypothesis (y_h). If $(y_h{+}1)^2{\le}x{+}1$ then y_c must be $y_h{+}1$ in order to make the first conjunct valid, while otherwise y_c must be y_h in order to satisfy the second conjunct. Rippling would be able to rewrite the conclusion into the hypothesis in each of these two cases but it requires logical inferences to create the case analysis and to instantiate the existentially quantified variable.

On the other hand, logical proof methods alone would not be able to detect the case distinction either, as they would have to prove the non-trivial lemmata $x{<}(y{+}1)^2 \Rightarrow x{+}1{<}(y_h{+}1+1)^2$ for the first case and $y_h^2{\le}x \Rightarrow y_h^2{\le}x{+}1$ for the second. Of course, it is easy to prove the step case of the induction, if one already provides the crucial lemmata $\forall z\, \forall t\; z{\le}t \Rightarrow z{\le}t{+}1$ and $\forall s\, \forall r\; s{<}r^2 \Rightarrow s{+}1{<}(r{+}1)^2$

as well as the case analysis[3] $\forall u \forall v\ u{\leq}v \vee v{<}u$, as the following matrix proof with multiplicity 2 for the conclusion shows.

$$\left[\begin{array}{cccccccc}
 & Z{\leq}^1 T{+}1 & Y_{c1}^2{\leq}^0 x{+}1 & \boxed{\begin{array}{c} U{\leq}^1 V \\[2em] V{<}^1 U \end{array}} & Y_{c2}^2{\leq}^0 x{+}1 & S{<}^0 R^2 & \\
y_h^2{\leq}^1 x & & & & & & x{<}^1 (y_h{+}1)^2 \\
 & Z{\leq}^0 T & x{+}1{<}^0 (Y_{c1}{+}1)^2 & & x{+}1{<}^0 (Y_{c2}{+}1)^2 & S{+}1{<}^1 (R{+}1)^2 &
\end{array}\right]$$

$$\sigma_Q = \{Z\backslash y_h^2, T\backslash x, Y_{c1}\backslash y_h, V\backslash x{+}1, U\backslash (y_h{+}1)^2, Y_{c2}\backslash y_h{+}1, S\backslash x, R\backslash y_h\}$$

But without the guidance of techniques that try to rewrite subformulae in the conclusion into the corresponding subformulae of the hypothesis one would have to search thousands of lemmata about arithmetics to find the ones that complete the proof, which would make the proof procedure far from being efficient.

It is therefore necessary to combine logical proof search techniques with controlled rewrite techniques such as rippling in order to improve the degree of automation when reasoning inductively about program specifications.

A first step in this direction [21] has shown that a combination rippling, *reverse rippling*, and simple first-order techniques can be used to generate the case analysis and to solve synthesis problems like the above automatically. The approach first decomposes the induction hypothesis by logical rules; uses sinks and wave-fronts to identify the corresponding subformulae in the conclusion; applies (forward) rippling rules to rewrite the components of the hypothesis; uses reverse rippling and matching to find a substitution and a rippling sequence that connects the conclusion with the result of rippling; and finally performs a consistency check to merge the individual substitutions into a conditional substitution for the conclusion as a whole. The search for a rippling sequence is based on the *rippling-distance* strategy introduced in [16, 7]. The resulting rippling proof together with the case analysis is translated back into a sequent proof that can be executed by the NuPRL proof development system [9].

Since the only weakness of the above approach lies in the sequent-based logical proof search, it suggests an integration of the above techniques into a matrix based theorem prover. Essentially this will lead to an even more compact matrix characterization of validity and thus to further reductions of the search space in inductive theorem proving. In the following we will outline the fundamental steps towards a combination of rippling and matrix-based theorem proving.

The techniques described in [21] enable us to generate *conditional substitutions* as a solution for a given synthesis problem. The conditions generated in the process will lead to a case analysis in the first step of a top-down sequent proof while each of the resulting subgoals can be proven by "conventional" proof techniques. A justification of this approach comes from the observation that a formula F is valid if and only if there is a set $\{c_1,..,c_n\}$ of logical constraints such that $C = c_1 \vee .. \vee c_n$ and each of the formulae $F_i = c_i \Rightarrow F$ are logically

[3] $a{\leq}b$ is usually just an abbreviation for $\neg(b{<}a)$.

valid. This observation, which can be proven by the cut rule, is the basis of the following modified characterization theorem.

Theorem 5. *A formula F is (intuitionistically) valid iff there is a set $\{c_1,..,c_n\}$ of constraints such that $C = c_1 \vee .. \vee c_n$ is valid and for all i there is*

- *a multiplicity $\mu_i := (\mu_{Q_i}, \mu_{J_i})$,*
- *an admissible combined substitution $\sigma_i := (\sigma_{Q_i}, \sigma_{J_i})$, and*
- *a set $\mathcal{C}_i$ of σ_i-complementary connections*

such that every path through $F_i^\mu = c_i \Rightarrow F^\mu$ contains a connection from $\mathcal{C}_i$.

Another compactification comes from the observation that a pair $\{A^1, \bar{A}^0\}$ of connected atomic formulae does not necessarily have to be equal under a given substitution σ. As complementary connections correspond to leaves in a sequent proof we only have to require that the negative atom A, which would occur on the left side of the sequent, *implies* $\bar{A}$ under σ. The implication, however, is not a general logical implication, but one that can be proven by standard decision procedures or by rewriting $\bar{A}$ into A. Thus a reasonable extension of the notion of complementarity is to consider *directed connections* $(A^1, \bar{A}^0)$ and to require A to imply $\bar{A}$ with respect to a given theory[4] $\mathcal{T}$, denoted by $A \Rightarrow_{\mathcal{T}} \bar{A}$.

Definition 3.
A directed connection (u^1, v^0) is *complementary* with respect to a theory $\mathcal{T}$ under a given combined substitution $\sigma{:=}(\sigma_Q, \sigma_J)$ if and only if $\sigma_J(\mathrm{pre}(u)){=}\sigma_J(\mathrm{pre}(v))$ and either $\sigma_Q(\mathit{label}(u)) = \sigma_Q(\mathit{label}(v))$ or $\sigma_Q(\mathit{label}(u)) \Rightarrow_{\mathcal{T}} \sigma_Q(\mathit{label}(v))$.

Basing a proof procedure on the above definition and the characterization given in Theorem 5 enables us to prove inductive specification theorems with existential quantifiers without having to provide domain lemmata and case-splits beforehand. Instead we extend the usual term-unification process by rippling and other theory-based reasoning techniques, which guarantee that the complementarity conditions are satisfied whenever the extended unification succeeds.

Another efficiency improvement during proof search comes from the observation that sinks and wave-fronts establish a strong relation between individual subformulae of the induction hypothesis and the conclusion. They help to identify connections that will be relevant for the proof and should be investigated first by the path-checking algorithm. By this we can reduce the search space, which is particularly helpful when searching for connections involving rippling.

We conclude this section with a small example that shows the advantages of the extensions discussed above.

Example 5. Consider $\exists y\ \neg(x{<}y^2) \wedge x{<}(y{+}1)^2 \vdash \exists y\ \neg(x{+}1{<}y^2) \wedge x{+}1{<}(y{+}1)^2$ from example 4 (after unfolding the abbreviation $\leq$) and its matrix below.

[4] This concept, which resembles the theory connections discussed in [5], could also be extended to unary or n-ary connections w.r.t some theory $\mathcal{T}$.

$$\begin{bmatrix} x<^1(y_h{+}1)^2 & x<^0 y_h^2 & \begin{matrix} x{+}1<^1 Y_c^2 \\ x{+}1<^0(Y_c{+}1)^2 \end{matrix} \end{bmatrix}$$

As connections should only run between atoms of the induction hypothesis and the corresponding axioms of the conclusion, the matrix contains only two useful connections $(x<^1(y_h{+}1)^2, x{+}1<^0(y_c{+}1)^2)$ and $(y_c^2<^1 x{+}1, y_h^2<^0 x)$. Since x and y_h are constants neither of the two connections can be unified and we have to use the rippling techniques described in [21] to show their complementarity. In the first case $\boxed{1}$ this eventually leads to the substitution $\sigma_{Q_1}=\{Y_c\backslash y_h{+}1\}$. As the instantiated second connection does not describe an arithmetically valid implication, which can easily be checked by arithmetical decision procedures, we add the open subgoal $(x{+}1<(y_h{+}1)^2$ as prerequisite c_1 (i.e. with opposite polarity 0) to the matrix and have thus established the validity of the sequent under this condition.

To complete the proof we now try to establish the validity of the sequent under the negated condition $c_2 = (x{+}1<^1(y_h{+}1)^2$. By connecting this condition to $x{+}1<^0(y_c{+}1)^2$ we get $\sigma_{Q_2}=\{Y_c\backslash y_h\}$ which also makes the connection $\boxed{2}$ an arithmetically valid implication. According to theorem 5 we have thus shown the intuitionistic validity of the given sequent.

From a technical point of view, the above proof has synthesized a case distinction, which is added to the matrix, and has increased the multiplicity of the the conclusion to 2, as illustrated in the indexed matrix below.

$$\begin{bmatrix} x<^1(y_h{+}1)^2 & x<^0 y_h^2 & \begin{matrix} x{+}1<^1 Y_c^2 \\ x{+}1<^0(Y_c{+}1)^2 \end{matrix} & \begin{matrix} \boxed{x{+}1<^0(y_h{+}1)^2} \\ \boxed{x{+}1<^1(y_h{+}1)^2} \end{matrix} \end{bmatrix}$$

Under the substitution $\sigma_Q = \{Y_{c1}\backslash y_h{+}1, Y_{c2}\backslash y_h\}$ the indicated four connections are complementary w.r.t the theory of arithmetic and span the indexed matrix in the conventional way.

6 Conclusion

We have presented a coherent account of matrix-based constructive theorem proving that combines proof search procedures for first-order intuitionistic logic with induction techniques. Besides proving the validity of a given formulae our combined method can also guide the development of proofs and programs in interactively controlled proof assistants. It enables us to greatly increase the degree of automation in these systems without having to sacrifice their elegance, the expressiveness of their underlying logics, or their interactive capabilities.

Matrix methods can thus be used as the central inference engine during the verification, synthesis, or optimization of programs.

Our key concept is to view matrix-based proof methods as proof planners that do not underlie the typical limitations of sequent or natural deduction calculi when searching for a solution to a given problem. During a derivation they can be called either as *trusted refiner* or as mechanism which generates a proof plan that will later be executed by the proof assistant.

Obviously, this concept is not restricted to first-order or inductive theorem proving. In a similar way we can also integrate proof procedures for other important logics, such as modal logics [20] or linear logic [14], or higher-level strategies for program synthesis [12, 13]. In many of these cases we can rely on already known successful techniques that were originally implemented independently and view their results as plans for the actual derivation. By executing this plan within a proof assistant like NuPRL, Coq, Alf, etc., we will eventually gain the amount of trust that is needed in the development of safety-critical applications.

In the future we will strengthen the integration of proof planning techniques into matrix-based theorem proving and investigate to what extent general planning and search techniques can be used to generate proof plans explicitly. We also intend to embed known synthesis, verification, and optimization techniques into proof assistants by using the same methodology. Our ultimate goal is to combine our experience with reasoning about group communication systems [11] and the above techniques into a highly automated proof environment for the development of safety-critical systems.

References

1. T. Altenkirch, V. Gaspes, B. Nordström, and B. von Sydow. *A user's guide to ALF*. University of Göteborg, 1994.
2. J. Bates and R. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, 1985.
3. W. Bibel. On matrices with connections. *Journal of the Association for Computing Machinery*, 28(633–645), 1981.
4. W. Bibel. *Automated Theorem Proving*. Vieweg, 1987.
5. W. Bibel. *Deduktion – Automatisierung der Logik*, volume 6.2 of *Handbuch der Informatik*. R. Oldenbourg, 1992.
6. W. Bibel, D. Korn, C. Kreitz, and S. Schmitt. Problem-oriented applications of automated theorem proving. In J. Calmet & C. Limongelli, eds., *Design and Implementation of Symbolic Computation Systems*, LNCS 1126, pp. 1–21, Springer, 1996.
7. W. Bibel, D. Korn, C. Kreitz, F. Kurucz, J. Otten, S. Schmitt, and G. Stolpmann. A multi-level approach to program synthesis. In N.E. Fuchs, ed.,*Seventh International Workshop on Logic Program Synthesis and Transformation*, LNAI 1463, pp. 1–25, Springer, 1998.
8. A. Bundy, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence*, 1992.
9. R. Constable, S. Allen, M. Bromley, et. al. *Implementing Mathematics with the NuPRL proof development system*. Prentice Hall, 1986.

10. G. Dowek et. al. *The Coq proof assistant user's guide.* Institut National de Recherche en Informatique et en Automatique, Report RR 134, 1991.
11. C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. In C. & H. Kirchner, ed., *15th International Conference on Automated Deduction*, LNAI 1421, pp. 317–332, Springer, 1998.
12. C. Kreitz. Formal mathematics for verifiably correct program synthesis. *Journal of the IGPL*, 4(1):75–94, 1996.
13. C. Kreitz. Program synthesis. In W. Bibel and P. Schmitt, eds., *Automated Deduction – A Basis for Applications*, chapter III.2.5, pp. 105–134, Kluwer, 1998.
14. C. Kreitz, H. Mantel, J. Otten, and S. Schmitt. Connection-Based Proof Construction in Linear Logic. In W. McCune, ed., *14th Conference on Automated Deduction*, LNAI 1249, pp. 207–221. Springer, 1997.
15. C. Kreitz, J. Otten, and S. Schmitt. Guiding Program Development Systems by a Connection Based Proof Strategy. In M. Proietti, ed., *Fifth International Workshop on Logic Program Synthesis and Transformation*, LNCS 1048, pp. 137–151. Springer, 1996.
16. F. Kurucz. Realisierung verschiedender Induktionsstrategien basierend auf dem Rippling-Kalkül. Diplomarbeit, Technische Universität Darmstadt, 1997.
17. H. Mantel and C. Kreitz. A matrix characterization for $\mathcal{MELL}$. In *6th European Workshop on Logics in AI (JELIA-98)*, LNAI, Springer, 1998.
18. J. Otten and C. Kreitz. A connection based proof method for intuitionistic logic. In P. Baumgartner, R. Hähnle & J. Posegga, eds., *4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, LNAI 918, pp. 122–137. Springer, 1995.
19. J. Otten and C. Kreitz. T-String-Unification: Unifying Prefixes in Non-Classical Proof Methods. In U. Moscato, ed., *5th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, LNAI 1071, pp. 244–260. Springer, 1996.
20. J. Otten and C. Kreitz. A Uniform Proof Procedure for Classical and Non-classical Logics. In G. Görz & S. Hölldobler, eds., *KI-96: Advances in Artificial Intelligence*, LNAI 1137, pp. 307–319. Springer, 1996.
21. B. Pientka and C. Kreitz. Instantiation of existentially quantified variables in inductive specification proofs. In *4th International Conference on Artificial Intelligence and Symbolic Computation*, LNAI 1476. Springer, 1998.
22. S. Schmitt and C. Kreitz. Deleting redundancy in proof reconstruction. In H. de Swaart, ed., *International Conference TABLEAUX-98*, LNAI 1397, pp. 262–276. Springer, 1998.
23. S. Schmitt and C. Kreitz. On transforming intuitionistic matrix proofs into standard-sequent proofs. In P. Baumgartner, R. Hähnle & J. Posegga, eds., *4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, LNAI 918, pp. 106–121. Springer, 1995.
24. S. Schmitt and C. Kreitz. Converting non-classical matrix proofs into sequent-style systems. In M. McRobbie & J. Slaney, eds, *13th Conference on Automated Deduction*, LNAI 1104, pp. 418–432. Springer, 1996.
25. L. Wallen. *Automated deduction in nonclassical logic.* MIT Press, 1990.