A Multi-Level Approach to Program Synthesis

W. Bibel[1] D. Korn[1] C. Kreitz[2] F. Kurucz[1] J. Otten[2] S. Schmitt[1] G. Stolpmann[1]

[1] Fachgebiet Intellektik, Fachbereich Informatik, Darmstadt University of Technology
Alexanderstr. 10, 64283 Darmstadt, Germany
[2] Department of Computer Science, Cornell University, Ithaca, NY 14853, USA

Abstract. We present an approach to a coherent program synthesis system which integrates a variety of interactively controlled and automated techniques from theorem proving and algorithm design at different levels of abstraction. Besides providing an overall view we summarize the individual research results achieved in the course of this development.

1 Introduction

The development of programs from formal specifications is an activity which requires logical reasoning on various levels of abstraction. The design of the program's overall structure involves reasoning about data and program structures. Inductive reasoning is necessary for determining a program's behavior on finite, but non-atomic data such as lists, arrays, queues, and sometimes even natural numbers. First-order reasoning is required to analyze the order of steps which are necessary to achieve a desired result. Propositional reasoning is used to make sure that all the formal details are correctly arranged.

Program synthesis and transformation is therefore strongly related to the concept of proofs. This has been particularly emphasized by the development of languages and tools for logic programming which use deductive techniques for the simulation of mathematical reasoning as their basic execution model.

In the field of Automated Theorem Proving (ATP) deductive systems have been developed for many of the above-mentioned areas. Each of these systems is tailored towards a particular style of reasoning but shows weaknesses outside its specific area. There is no single automated proof procedure which can handle all the reasoning problems occurring during program synthesis equally well and because of the very nature of the problem it is not very likely that there will ever be one. Instead, it is more meaningful to combine the strengths of the individual proof procedures by integrating them into a single reasoning system which can perform reasoning at all the above-mentioned levels of abstraction.

During the last few years the Intellectics Laboratory of Darmstadt Institute of Technology has been active in the development of such an integrated, application-oriented ATP-system which can serve as an inference engine of a coherent program synthesis system. For this purpose we have developed specialized proof procedures which deal with problem formalizations on the propositional, (constructive) first-order, inductive, and higher levels. On the other hand we have generated interfaces for each of these procedures which make it possible to present the generated proof in a common logical calculus. The resulting multi-level synthesis system, called MAPS, can extract individual proof tasks from a

given programming problem, delegate them to specialized proof procedures, and combine the resulting proofs into a solution of the original problem. In addition to that it will be able to proceed interactively whenever none of the proof procedures can handle the task automatically.

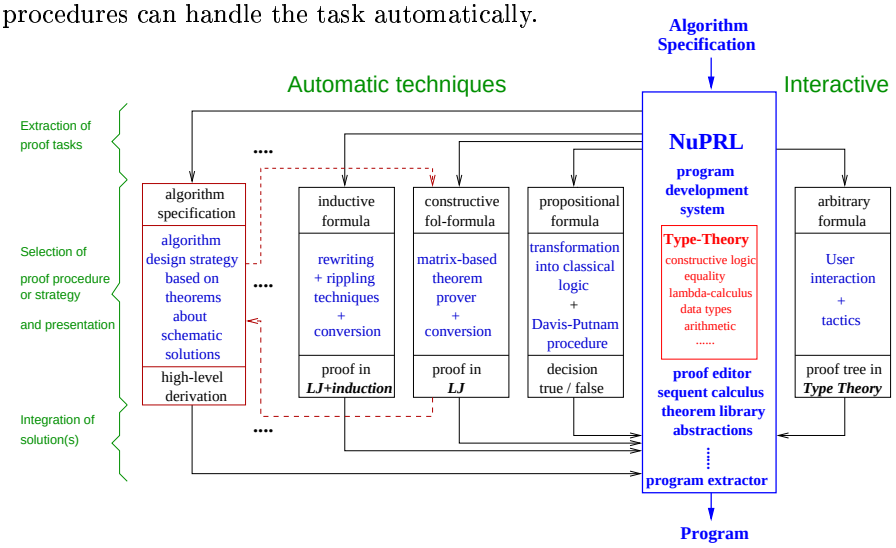


Fig. 1. Structure of the MAPS program synthesis system

The conceptual structure of MAPS is illustrated in Fig. 1. It shows on the left hand side automatic proof procedures for different levels of reasoning, viz. propositional, first-order, and inductive reasoning as well as high-level algorithm design strategies. Each of these procedures will receive proof tasks from a program development system, indicated by the horizontal arrows on top, which were extracted from a given synthesis problem. After solving their tasks the proof procedures will send their solution to a conversion module. This module will generate a representation of the solution in the common calculus and return it to the program development system (horizontal arrows on the bottom level). The dotted arrows indicate that the high-level strategies will ideally create subtasks which can be handled by the lower-level procedures immediately. If none of the available proof procedures suits the proof task to be solved the program development system will have to rely on user interaction (right hand side).

As common platform for our work we have chosen the NuPRL proof development system [10] since its underlying logical calculus can deal with a rich variety of problems from mathematics and programming and allows to formalize even high-level strategies in a natural way. Since it is based on the proofs-as-programs paradigm to program synthesis [2] it allows to treat algorithm design strategies as proof procedures and to integrate a great variety of reasoning techniques on all levels of abstraction. Finally it supports interaction with a human expert (programmer) whenever the automated strategies turn out to be too weak.

All our automated proof procedures were originally developed independently from the common platform and we had to provide techniques for integrating them into the top-down sequent proof style of NuPRL.

- Formulas from propositional intuitionistic logic will be decided by translating them into classical logic [17] and applying a non-normal form Davis-Putnam procedure [28]. This procedure will be embedded as trusted refiner which creates a sequent proof on demand.
- Matrix methods for constructive first-order logic use a non-clausal extension of the connection method [4, 30]. They have been combined with an algorithm for translating matrix proofs into sequent proofs [36] and integrated into NuPRL as a proof tactic [22].
- Inductive proofs will be generated by proof planners involving rippling [9] and rewrite techniques. Sequences of rewrite steps will be transformed into applications of cut- and substitution rules while other techniques will determine the parameters of the general induction rule [25, 23].
- High-level synthesis strategies will be integrated by verifying formal theorems about schematic program construction [18, 19]. For each strategy a theorem describing the axioms for the correctness of a particular class of algorithms will serve as derived inference rule. It will be accompanied by specialized tactics for determining and validating values for its parameters [43]. This technique heavily relies on verified domain knowledge [42] but is very effective.

The MAPS enterprise may be seen as a milestone in the long tradition of program synthesis efforts of our group which started as early as 1974 eventually leading to the program system LOPS (see [6] for a detailed exposition of this development). In lack of powerful proof systems at that time the emphasis then was laid on high-level strategies guiding the synthesis (or search for a proof) while in MAPS it is laid more on the proof obligations resulting in the synthesis task. The present paper considerably extends the preliminary outline of the concepts underlying MAPS given in [8] and presents the results achieved in the meantime.

In the following we shall describe our proof methods and their integration into the NuPRL program development system. In Section 2 we shall discuss proof procedures for intuitionistic propositional and first-order logic while Section 3 describes the integration of rewriting techniques for inductive theorem proving. Section 4 deals with higher-level synthesis strategies, particularly with algorithm design strategies based on schematic solutions for certain classes of algorithms. We conclude with an outlook to future work.

2 Integrating Theorem Provers for First Order Logic

In this section we will give a survey on automated proof search procedures we have developed for the first-order and propositional fragment of intuitionistic logic. Furthermore we shall briefly discuss how to integrate the proofs constructed by these procedures into the NuPRL environment.

2.1 Decision Procedures for Intuitionistic Propositional Logic

The intuitionistic validity of propositional formulas could in principle be investigated by first-order proof procedures. Nevertheless there are good reasons to develop methods tailored to the specific properties of propositional logic:

1. first-order methods usually fail to detect the invalidity of a propositional formula
2. the technical overhead necessary to deal with quantifiers can be skipped if the formula under consideration is propositional only
3. in many cases all that is asked about a propositional formula can essentially be answered by "yes" or "no" instead of an actual proof construction

In classical logic these insights have led to decision procedures like the Davis-Putnam procedure which currently is about the most efficient complete proof method for propositional classical logic. Unfortunately, attempting to adopt this technique into intuitionistic propositional logic leads to serious difficulties:

- the existing Davis-Putnam procedures are defined for formulas in clausal form only whereas there is no clausal form for intuitionistic formulas
- the essential idea of the Davis-Putnam procedures is a successive application of the law of the excluded middle which does not hold in intuitionistic logic

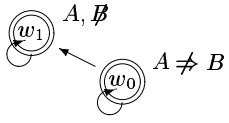
In this section we present two techniques we have developed in order to overcome both difficulties: a translation method from intuitionistic into classical propositional logic as well as a non-clausal Davis-Putnam procedure.

Translating intuitionistic into classical propositional formulas. A natural approach to deal with intuitionistic validity is to formalize the conditions for intuitionistic forcing within classical first-order logic. $A \Rightarrow B$, for instance, would be translated into $\forall v.(wRv \Rightarrow A(v) \Rightarrow B(v))$ where w denotes the current possible world. For the sake of completeness axioms encoding the properties of the accessibility relation R will be added which then must imply the translated formula. This technique is known as the *relational translation* [26, 27, 3].

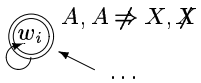
A major difficulty of this approach is the potential undecidability of the resulting classical formula. On the other hand, any intuitionistic non-theorem has a *finite* countermodel. This means that only finitely many possible worlds need to be considered and that one could use finite conjunctions instead of having to quantify over all possible worlds. Our aim therefore was to find a sufficiently effective mechanism for constructing such finite potential countermodels. To this end we have investigated the reasons which lead to infinite countermodels as described in the following.

Essentially a potential countermodel can be extended by a new possible world in two cases which both yield important conditions for adding any further possible worlds. If these conditions are not considered then infinitely many worlds without "new" properties could successively be added to the countermodel:

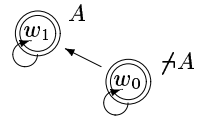
The first case occurs when an implicative formula $A \Rightarrow B$ is assumed not to be forced at a given possible world w_0. In this case we have to assume an accessible world w_1 where A is forced but B is not according to the Kripke-semantics for intuitionistic logic. This countermodel is shown in the right figure. Note, however, that A will remain forced at any world w_i accessible from w_1. Thus if we encounter the case that $A \Rightarrow X$ is assumed not to be forced at such a w_i then, in order to obtain a world accessible from w_i where A is forced but



X is not, we only need to ensure X not to be forced at w_i which is accessible from itself by reflexivity. The respective situation is shown to the right. Hence, once we have added a world to our countermodel refuting some $A \Rightarrow B$ it is not necessary anymore to add another world accessible from there in order to refute $A \Rightarrow X$ for some X.



Likewise, the second case occurs when refuting a negated formula $\neg A$ at a given world w_0. In this case we need to provide another accessible world w_1 where A is forced. This situation is again shown in the right figure. Once we have done so, however, there is no need to add *any* other accessible world from there on. To see why, we need the notion of *F-maximality* of a given world for a given propositional formula F. We consider a given possible world F-maximal iff no accessible world forces some subformula of F which is yet unforced at the given world. One can easily show that for any world w and any propositional formula F there always is an accessible F-maximal world $\max_F(w)$ (cf. [17]). Once we have (potentially) added w_1 to our countermodel as shown above we know that there must also be an F-maximal $\max_F(w_1)$ accessible from w_1, where F is the input formula of our translation. But then A is also forced at $\max_F(w_1)$ and we can well add $\max_F(w_1)$ to our countermodel instead of w_1. The main advantage of doing so is that whenever we would have to add a new world to our countermodel accessible from $\max_F(w_1)$ in order to force some subformula F' of F we know that F' must already be forced at $\max_F(w_1)$. Thus instead of actually adding this accessible world we can simply add its properties to $\max_F(w_1)$.



Obeying both restrictions for adding new worlds during the inductive construction of a potential countermodel for a propositional formula F will always lead to a finite set of possible worlds within this countermodel since F has only finitely many subformulas. To encode a potential intuitionistic countermodel for a given input formula F we first associate a unique function symbol $\mathtt{w}_i$ with each positive occurrence of an implicative or negated subformula in F. Then we construct a set W of terms by applying a proper choice of concatenations of the $\mathtt{w}_i$ to the root possible world $\mathtt{w}_0$. The order of the $\mathtt{w}_i$ within these concatenations essentially reflects the possible refutation orderings between the associated subformulas. No function symbol will occur more than once within a single concatenation and function symbols associated with negated subformulas will only occur as the outermost symbol of a concatenation.

Given a term $t \in W$, which now denotes a particular possible world within a potential countermodel, the finite set $R_F(t)$ of accessible worlds will then be the set of those terms $t' \in W$ which contain t as a subterm. We can now essentially apply the usual relational translation approach to F . However, instead of positive occurrences of universal quantifications over accessible worlds we use an appropriate such term as a representative for an arbitrary accessible world. Negatively occuring quantifications are replaced by finite conjunctions over all such terms denoting an accessible world. For any further details cf. [17].

To sum up we have achieved a morphism from intuitionistic to classical logic that maps propositional input formulas to propositional output formulas (note that no quantifiers or uninstantiated terms occur in the output formula).

A non-clausal Davis-Putnam proof procedure. The Davis-Putnam procedure [11, 12] is one of the most successful proof procedures for classical propositional logic. Its essential idea is to apply the following *splitting rule* to prove a formula F: assign the truth values *true* and *false* to a selected propositional variable X occurring in F and simplify the resulting formulas, yielding F_1 and F_2. This rule is applied recursively to the formulas F_1 and F_2 until the truth values *true* or *false* are reached. The investigated formula F is valid if all leaves of the resulting proof tree are marked with *true*, otherwise F is not valid.

Unfortunately the original formulation of the Davis-Putnam procedure and all existing implementations require the formula F in clausal form, i.e. in disjunctive normal form. The usual translation of a given formula into this form is based on the application of distributivity laws. In the worst case this will lead to an exponential increase of the resulting formula. The application of the so-called *definitional translation* [33] yields (at most) a quadratic increase of the resulting formula's size at the expense of introducing new propositional variables.

The translation of intuitionistic into classical propositional formulas described above leads to formulas which are strongly in non-normal form. Experimental results have shown that a translation to clausal form often yields formulas which are too large to obtain a proof, in particular if applying the standard translation techniques. To avoid any translation steps to clausal form we have developed a non-clausal proof procedure [29]. It is a generalization of the original clausal Davis-Putnam procedure and operates directly on arbitrary propositional formulas. To this end we represent formulas by nested matrices. A *matrix* is a very compact representation of a formula and the corresponding search space (see also section 2.2). In the clausal Davis-Putnam procedure we regard a matrix as a set of clauses where each clause is a set of literals. In our non-clausal approach a clause is a set of matrices and a matrix is either a literal or a set of clauses.

In the original Davis-Putnam procedure the above-mentioned splitting rule consists of a *clause elimination step* and a *literal deletion step*. Due to the more generalized treatment of arbitrary formulas the non-clausal splitting rule uses a *matrix elimination step* instead of the literal deletion step. In contrast to the latter it will delete a whole matrix, not only a single literal. Furthermore in the non-clausal approach an additional splitting rule, called *beta splitting rule*, is applicable. Our experimental results have shown three advantages of our non-clausal proof procedure: no translation to any clausal form is required, the application of a more general matrix elimination step is possible and an additional beta splitting rule is applicable which can shorten proofs considerably.

In practice, our translation from intuitionistic into classical logic combined with the Davis-Putnam procedure described above has turned out to be a very promising approach to deal with propositional intuitionistic logic. Already our prototypic implementations of both approaches in Prolog were able to decide the intuitionistic validity of a variety of propositional formulas with a performance competitive to any intuitionistic decision mechanism known to us.

2.2 Proof Construction in Intuitionistic First-Order Logic

The connection method is a well-known proof procedure for classical first-order logic and has successfully been realized in theorem provers like Setheo [24] or KoMeT [7]. It is based on a *matrix characterization* of logical validity: *A formula F is (classically) valid iff the matrix of F is (classically) complementary* [4, 5].

In *propositional* classical logic the matrix of a formula F is *complementary* if there is a spanning set $\mathcal{C}$ of connections for F. A *connection* is a pair of atomic formulas with the same predicate symbol but different polarities.[3] A connection corresponds to an *axiom* in the sequent calculus. A set of connections $\mathcal{C}$ *spans* a formula F if every path through F contains at least one connection from $\mathcal{C}$. With regard to a sequent calculus this means that all branches are closed by an axiom. A *path* through F contains the atoms on a horizontal path through the matrix representation of F. A *matrix* of a formula F is a compact representation of F and the corresponding search space. This characterization also applies to classical *first-order* logic if each connection in $\mathcal{C}$ is complementary, i.e. the terms of each connection in $\mathcal{C}$ can be made identical by some *first-order substitution* σ_Q in which (quantifier-)variables are replaced by terms.

Certain rules in the intuitionistic sequent calculus $\mathcal{LJ}$ differ from the classical $\mathcal{LK}$ [14]. The arising non-permutabilities between these rules need a special treatment. In the matrix characterization for *intuitionistic* logic [44] this is done by an additional *intuitionistic substitution* σ_J. This substitution has to make the prefixes of each connection identical and therewith complementary. A *prefix* of an atom is a string consisting of variables and constants which essentially describes the position of it in the tree representation of the formula to be proved.

Example 1. Consider $F_1 \equiv (Pa \Rightarrow \neg\neg\exists x Px)$. The prefixes of the atomic formulas Pa and Px are a_0A_1 and $a_0a_2A_3a_4$, respectively, where capital letters refer to variables and small letters indicate constants. The set $\{Pa, Px\}$ is a connection. It is complementary under the first-order substitution $\sigma_Q = \{x \backslash a\}$ and the intuitionistic substitution $\sigma_J = \{A_1 \backslash a_2A_3a_4\}$. Since the set $\mathcal{C} = \{\{Pa, Px\}\}$ spans F_1, the formula F_1 is intuitionistically valid.

Example 2. Let $F_2 \equiv (\neg\neg P \Rightarrow P)$. The prefixes of the two atoms $a_0A_1a_2A_3$ and a_0a_4 are not unifiable. Therefore the formula F_2 is *not* intuitionistically valid.

According to the above matrix characterization the validity of a formula F can be proved by showing that all paths through the matrix representation of F contain a complementary connection. Therefore for an automated proof search procedure based on a matrix characterization we have to (1) search for a spanning set of connections and (2) test the connections for complementarity.

Developing a proof procedure for *intuitionistic* first-order logic based on Wallen's matrix characterization means extending Bibel's connection method accordingly. It consists of an algorithm which checks the complementarity of all paths and uses an additional string-unification procedure to unify the prefixes.

[3] The *polarity* of an atomic formula is either 0 or 1 and indicates whether it would occur negated (polarity 1) in the negational normal form or not (polarity 0).

Searching for a spanning set of connections. Proof search is done by a general path checking algorithm which is driven by connections instead of logical connectives [30, 32]. Once a complementary connection has been identified all paths containing this connection are deleted. This is similar to Bibel's connection method for classical logic but without necessity for transforming the given formula to normal form. Dealing with arbitrary formulas is necessary since there is no clausal form in intuitionistic logic. The advantage of such a method is that the emphasis on *connections* drastically reduces the search space compared to calculi which are *connective*-driven such as the sequent calculus or tableau calculi. Furthermore it avoids the notational redundancy contained in these calculi.

Testing the connections for complementarity. In our path checking procedure we have to ensure that after adding a connection there are still first-order and intuitionistic substitutions which make all connections complementary. While the first-order substitution σ_Q can be computed by well-known *term*-unification algorithms we had to develop a specialized *prefix*-unification procedure for computing σ_J. This is done by a specialized algorithm for string-unification [31]. String-unification in general is quite complicated but unifying prefixes is much easier since there are certain restrictions on prefixes: prefixes are strings without duplicates and in two prefixes (corresponding to atoms of the same formula) equal characters can only occur within a common substring at the beginning of the two prefixes. This enabled us to develop a much simpler algorithm computing a *minimal* set of most general unifiers.

Our general proof procedure also allows a uniform treatment of other non-classical logics like various modal logics [32] or linear logic [21]. We only have to change the notion of complementarity (i.e. the prefix unification) while leaving the path checking algorithm unchanged.

Path checking can also be performed by using a semantic tableau [13]. The prover leanTAP [28] is based on free-variable semantic tableaux extended by the above-mentioned prefix unification. It is a very compact Prolog implementation (about 4 kilobytes) and due to the modular treatment of the different connectives it can easily be adapted to other non-classical logics.

2.3 Embedding Matrix Methods into Program Development

As long as only the matter of truth is involved, NuPRL allows to use the above techniques as *trusted external refiners*. However, whenever a piece of code shall be extracted from the proof, it is necessary to convert the proofs generated by a search procedure back into a constructive sequent proof which, according to the *proofs-as-program paradigm* [2], can be turned into a program.

In [36, 22] we have developed an embedding of connection based proof methods into NuPRL based on such conversions. The proof method described in [22] constructs a matrix proof closely related to a cut-free sequent proof in $\mathcal{LJ}_{mc}$, the multiply-conclusioned sequent calculus on which the matrix characterization for $\mathcal{J}$ is based [44]. Its integration into NuPRL basically consists of a transformation from $\mathcal{LJ}_{mc}$-proofs into sequent proofs in Gentzen's $\mathcal{LJ}$ [14], the first-order

fragment of NuPRL's calculus. To allow a *structure preserving* transformation the *cut*-rule had to be used in a restricted and regular manner. For the sake of clarity we have hidden its application within an extended sequent calculus $\mathcal{LJ}^\star$.

Converting matrix proofs into sequent proofs. Improving the efficiency of proof search in the above procedures resulted in strategies which do not support a parallel construction of matrix proofs in $\mathcal{MJ}$ and $\mathcal{LJ}_{mc}$-proofs anymore. Proof strategies such as an *extension procedure* [32] or a *tableaux* prover [28] (see also section 2.2) make it necessary to transform matrix proofs into sequent proofs *after* the proof search has been finished. Hence, the above mapping $\mathcal{LJ}_{mc} \mapsto \mathcal{LJ}^\star$ has to be extended by an additional mapping $\mathcal{MJ} \mapsto \mathcal{LJ}_{mc}$.

This two-step conversion from intuitionistic matrix proofs into $\mathcal{LJ}^\star$-sequent proofs has first been presented in [36]. The first step $\mathcal{MJ} \mapsto \mathcal{LJ}_{mc}$ turns out to be non-trivial since the compact representation of $\mathcal{MJ}$-proofs, called *reduction ordering* $\propto^\star$, does not completely encode the non-permutabilities of sequent rules in an $\mathcal{LJ}_{mc}$-proof. In order to *complete* this representation in the above sense we have extracted some conditions, called *wait*-labels, which are dynamically added during the conversion process. These conditions prevent non-invertible $\mathcal{LJ}_{mc}$-rules from being applied too early such that no proof relevant sequent formulas will be deleted. We explain our approach by an example.

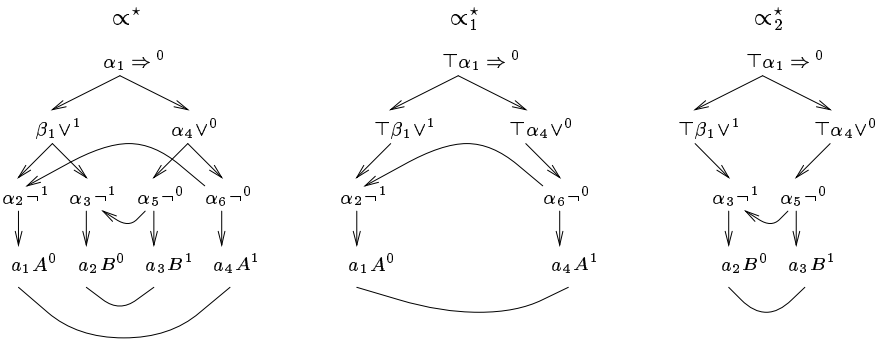


Fig. 2. Reduction ordering $\propto^\star$ of the matrix proof for F.

Consider the $\mathcal{J}$-formula $F \equiv \neg A \vee \neg B \Rightarrow \neg B \vee \neg A$ and its matrix proof $\mathcal{MJ}$ represented as a reduction ordering $\propto^\star$ (see Fig. 2, left hand side). $\propto^\star$ consists of the formula tree of F together with additional ordering constraints (curved arrows) which are extracted from the matrix proof and encode non-permutabilities of sequent rules wrt. $\mathcal{LJ}_{mc}$. Furthermore, the *connections* from the matrix proof are assigned to the atoms of F in $\propto^\star$. For unique reference to the subformulas of F each node in $\propto^\star$ contains a *position* x as well as the *main operator* $op(x)$ and *polarity* $pol(x)$ of the corresponding subformula F_x. Positions with a 'β' name denote subformulas which cause the sequent proof to split into two independent subproofs, e.g. β_1 in the example.

Proof reconstruction is based on a traversal of the reduction ordering $\propto^\star$ by visiting the positions of $\propto^\star$: (i) Select a position x from the *open position set* P_o which is not "blocked" by some arrow in $\propto^\star$, (ii) construct a unique sequent rule

from $pol(x)$ and $op(x)$ of the subformula F_x and reduce the formula $F_x^{pol(x)}$ in the sequent, (iii) update P_o with the immediate successor positions $succ(x)$ of x in $\propto^\star$. The traversal process and the resulting sequent proof for F are depicted in Fig. 3. After reducing the β-position β_1 the reduction ordering is split into two suborderings $\propto_1^\star, \propto_2^\star$ and the conversion continues separately on each of the suborderings. For this we have developed an operation $split(\propto^\star, \beta_1)$ [37] which first splits the reduction ordering $\propto^\star$. Secondly non-normal form reductions are applied to each of the $\propto_i^\star$ in order to delete redundancies from $\propto_i^\star$ which are no longer relevant for the corresponding branch of the sequent proof. The result of the splitting process is shown in Fig. 2, right hand side, where the positions already visited are marked with '$\top$'.

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

	P_o	select x	rule	applied on $F_x^{pol(x)}$
$\propto^\star$	$\{\alpha_1\}$	α_1	$\Rightarrow r$	$(\neg A \vee \neg B \Rightarrow \neg B \vee \neg A)^0$
	$\{\beta_1, \alpha_4\}$	α_4	$\vee r$	$(\neg B \vee \neg A)^0$
	$\{\beta_1, \alpha_5, \alpha_6\}$	β_1	$\vee l$	$(\neg A \vee \neg B)^1$
$\propto_1^\star$	$\{\alpha_2, \alpha_6\}$	α_6	$\neg r$	$\neg A^0$
	$\{\alpha_2, a_4\}$	a_4	—	A^1
	$\{\alpha_2\}$	α_2	$\neg l$	$\neg A^1$
	$\{a_1\}$	a_1	$ax.$	A^0, A^1
$\propto_2^\star$	$\{\alpha_3, \alpha_5\}$	α_5	$\neg r$	$\neg B^0$
	$\{\alpha_3, a_3\}$	a_3	—	B^1
	$\{\alpha_3\}$	α_3	$\neg l$	$\neg B^1$
	$\{a_2\}$	a_2	$ax.$	B^0, B^1

Fig. 3. Sequent proof for F and the corresponding traversal steps of $\propto^\star$.

The problem of completing $\propto^\star$ occurs when starting the traversal with $\alpha_1, \alpha_4, \alpha_5$, which is not prevented by "blocking" arrows in $\propto^\star$. But such a selection ordering leads to a $\mathcal{LJ}_{mc}$-derivation which could not be completed to a proof since the reduction of α_5, i.e. applying $\neg r$ on $\neg B^0$, deletes the relevant formula $\neg A^0$ (position α_6). Adding two *wait*-labels dynamically to α_6 and α_5 completes $\propto^\star$ and avoids this deadlock during traversal. For a more detailed presentation of this approach as well as for an algorithmic realization we refer to [37].

Building efficient conversion procedures. The basic problem for proof reconstruction in constructive logics lies in the deletion of redundancies after splitting at β-positions. The reason for this is that the reduction ordering together with dynamically assigned *wait*-labels could be totally blocked from further conversion steps although some of these labels are no longer needed. To avoid this kind of deadlocks and to ensure completeness of the reconstruction process we have to detect and delete these redundant subrelations from $\propto_i^\star$. One of the deletion concepts used in the operation *split* is based on a non-normal form purity reduction which is recursively applied to non-connected leaf positions in $\propto^\star$. Consider $\propto_1^\star$ in the example above. The atom a_3 is not connected after splitting at β_1. Application of the purity reduction deletes a_3 and α_5 from $\propto_1^\star$. Consequently, the *wait*-label could be removed from α_6 since α_5 does not exist any longer. If the purity reduction were not applied, both *wait*-labels would remain in $\propto_1^\star$ which would then be totally blocked for further reconstruction steps.

In [37, 38] we have shown that complete redundancy deletion after splitting at β-positions cannot be performed *efficiently* when only the *spanning mating* is given from the matrix proof. Efficiency means that the selection of proof-relevant subrelations from the $\propto_i^\star$ should avoid any additional search. If only the spanning mating is given, backtracking may be required over this selection (i.e. converting irrelevant subrelations) in order to retain completeness.

For this purpose we have developed a concept of redundancy elimination from a reduction ordering during proof reconstruction [34, 35]. The concept is based on the specification of additional *proof knowledge* from the search process in order to extract *reconstruction knowledge* for the conversion procedure. More precisely, the history of matrix proofs will be integrated into the conversion process rather than using only the spanning matings. This makes our procedure depend on a particular proof search strategy, i.e. an *extension procedure* [5, 32]. But a compact encoding of this proof knowledge into the conversion process (which can be done in polynomial time in the size of the matrix proof) allows us to derive the reconstruction knowledge in terms of a few elegant conditions. Finally, the resulting *conversion strategy* integrates these conditions into the *split* operation which efficiently extends redundancy deletions after β-splits to a maximal level. We are able to show that *all* redundancies in the resulting subrelations $\propto_1^\star, \propto_2^\star$ will be eliminated after splitting $\propto^\star$ at a β-position. This guarantees that no decisions on selecting proof-relevant subrelations have to be made and hence, additional search wrt. these decisions will be avoided.

Our approach for reconstructing $\mathcal{LJ}_{mc}$-proofs from $\mathcal{MJ}$-proofs has been uniformly extended to various non-classical logics [37, 21] for which matrix characterizations exist. A uniform representation of different logics and proofs within logical calculi as well as abstract descriptions for integrating special properties of these logics in a uniform way, e.g. the completion of reduction orderings $\propto^\star$, yields a general proof reconstruction method for all logics under consideration.

Furthermore, a technique for efficient redundancy elimination after splitting at β-positions has been developed for all of these logics [35]. The result can be seen as a general framework for building efficient and complete conversion procedures for non-classical logics when the basic proof search method is known. The theoretical concept for extracting reconstruction knowledge form the corresponding proof knowledge is invariant wrt. a special logic and hence, extends the uniformity of the underlying conversion theory.

3 Induction Techniques

Pure first-order logic theorem proving can only generate programs without loops. For deriving recursive programs induction techniques are needed during the proof process. In [23] we have developed an induction prover for "simple" induction problems which is based on rippling [9, 1]. The basic concept for integrating this external prover into the NuPRL system is similar to the first-order case: (i) separating a subgoal in the actual NuPRL sequent, (ii) searching an induction proof for the goal with the external prover, and (iii) converting the induction

proof into a NuPRL sequent proof. This integration concept has been realized with tactics and extends an earlier approach presented in [25].

3.1 Introduction to Rippling

In order to prove a goal by induction an induction scheme of the form

$$A(base) \ \wedge \ (\forall x.A(x) \Rightarrow A(step(x))) \ \Rightarrow \ \forall x.A(x)$$

has to be applied to the goal which results in the following two subgoals: a *base case* $A(base)$, which for the most part can be proved directly, and a *step case* $\forall x.A(x) \Rightarrow A(step(x)$ which needs term rewriting to derive the conclusion $A(step(x))$ from the induction hypothesis $A(x)$. To perform rewriting in a goal oriented way, a special technique called *rippling* was introduced by Bundy [9]. A more refined and formalized version has later been developed by Basin and Walsh [1] from which we take the central ideas for our presentation.

Rippling uses *annotations* on subterms to mark the differences between the conclusion and the hypothesis. It first identifies additional function symbols in the conclusion, called *wave fronts*, which will be annotated by surrounding boxes. Subterms inside a wave front which are identical to the corresponding subterms in the hypothesis are called *wave holes* and will be underlined in the depictions. Consider for example the step case for the associativity of '+'

$$(x + y) + z = x + (y + z) \ \Rightarrow \ (s(x) + y) + z = s(x) + (y + z),$$

for which the annotated conclusion is given by

$$(\boxed{s(\underline{x})}^{\uparrow} + y) + z = \boxed{s(\underline{x})}^{\uparrow} + (y + z).$$

Arrows at boxes indicate the direction to which the wave fronts will be moved (or *rippled*) in the term tree. An '$\uparrow$' means that a wave front has to be moved towards the root (*rippling-out*) whereas '$\downarrow$' permits a wave front to be moved towards the leaves (*rippling-in*). For this purpose annotated rewrite rules called *wave rules* are used, e.g.

$$\boxed{s(\underline{U})}^{\uparrow} + V \ \stackrel{R}{\longmapsto} \ \boxed{s(\underline{U + V})}^{\uparrow} \tag{1}$$

$$\boxed{s(\underline{U})}^{\uparrow} = \boxed{s(\underline{V})}^{\uparrow} \ \stackrel{R}{\longmapsto} \ U = V \tag{2}$$

A proof using the rippling-out strategy is successfully finished, if all wave fronts have been eliminated by applying wave rules. If rippling-in is used instead each universally quantified variable of the hypothesis is marked with a special *sink* symbol '$\lfloor sink \rfloor$'. All wave fronts have to be rippled towards these sink positions, which requires the application of a rule for *switching* from '$\uparrow$ to '$\downarrow$' (there is no rule for the opposite direction) and of additional wave rules for rippling-in. Afterwards a substitution has to be found which matches the sink variables in the hypothesis with the corresponding terms in the wave fronts. Backtracking may be required in order to find instances for all sink variables.

The main difference between the two strategies is that rippling-out provides a goal-oriented proof search whereas rippling-in does not. For rippling-out each step moves a wave front towards the root of the term tree and the search cannot

branch. In contrast to this, rippling-in guides a wave front only to be rippled towards the leaves without giving guarantee that there exists a sink under the actual wave front position. Backtracking is required to find a sequence of rules which ripples all wave fronts into sink positions. A *sink heuristic*, defined in [1], makes sure that rippling-in always ripples a wave front towards sink positions. The restriction on the class of provable problems caused by this heuristic is harmless compared with the gain one obtains by the reduced backtracking.

3.2 Rippling-Distance – A Uniform Rippling Strategy

Even with the sink heuristic rippling-in often has an untractable search space. In order to obtain an efficient induction strategy we have generalized rippling-out and rippling-in to a new uniform strategy, called *rippling-distance* [23]. The arrows '$\uparrow$' and '$\downarrow$' were removed from the wave fronts and each wave front is assigned to one *goal sink*. To guarantee termination a *distance measure* $\mathcal{MD}$ has been introduced which describes the distance between a wave front and its assigned goal sink in the term tree. Each application of a wave rule has to reduce this measure wrt. the selected wave front. This strategy splits the enormous search space into smaller subspaces which can be searched independently.

Rippling-distance provides a more goal-oriented proof search than rippling-in with sink heuristic since it implicitly contains the application of the switching-rule. No backtracking over the switching position in the term tree has to be done. In the worst case m^n assignments from wave fronts to goal sinks have to be tested for an annotated term with m wave fronts and n sinks. The number of possible assignments seems to be very large, but this heuristic allows us to divide the proof search into separate search tasks. Whereas a non-separated search has a complexity of about $m^{(\frac{1}{4}n \cdot m \cdot d^2)}$ steps for finding a rippling proof the divided search needs only about $m^{(n+m \cdot d)}$ steps, where d is the depth of the term. Furthermore, the assignment concept gives us control knowledge for avoiding assignments which most likely do not contribute to a successful search. An extension of dividing the proof search can be reached by separating the wave fronts into independent classes such that each class is assigned to a different goal sink. Then an independent proof search for each class reduces the complexity to about $m! \cdot d$ steps for $m = n$, and to $(m')^{(n+m' \cdot d)}$ steps for $m > n$, where $m' = m - (n-1)$ (see [23] for details).

In order to uniformly integrate rippling-out into the rippling-distance strategy the definition of *sinks* has been generalized to arbitrary term positions. Then rippling-out can be seen as a special case of rippling-in by putting a sink around the whole term on which the wave rules will be applied, e.g.

$$\lfloor \boxed{s(\underline{x})} \leq \boxed{s(\underline{x})} \cdot \boxed{s(\underline{x})} \rfloor$$

This approach can be optimized if the outermost relation of the term is equality. Then two sink positions are defined at the immediate subterms of '=', e.g.

$$\lfloor (\boxed{s(\underline{x})}^{\uparrow} + y) + z \rfloor = \lfloor \boxed{s(\underline{x})}^{\uparrow} + (y + z) \rfloor$$

The distance measure $\mathcal{MD}$ can be applied directly to this rippling-out simulation without any changes. For practical use within a rippling prover we have combined rippling-distance with *dynamically* annotated wave rules. This means that the annotations of wave rules are determined at runtime from a set of rewrite rules which do not have annotations. Since there are no direction marks '$\uparrow$', '$\downarrow$' at the wave fronts the number of possible annotations is decreased and the annotations are easier to compute [23].

The admissibility of annotated wave rules has to be tested using a well founded reduction ordering $\prec_x$ in order to avoid cyclic sequences of wave rules. From the measure $\mathcal{MD}$ we have developed a new reduction ordering $\prec_{dist}$ which can be computed more efficiently than the ordering $\prec_{comp}$, the compound reduction ordering for rippling-in presented in [1]. This advantage becomes remarkable if *multi-wave holes* are used where wave fronts may contain more then one wave hole. Furthermore, $\prec_{dist}$ has been extended with an additional weight-function, which allows the use of additional associativity and commutativity wave rules.

3.3 Integrating the Rippling-Distance Strategy into NuPRL

In [23] we have described the integration of an external rippling prover into the NuPRL system which uses rippling-distance with dynamic rule annotations for guiding a proof search. The prover is implemented in NuPRL-ML [15] and called during a NuPRL proof session via a tactic *Ripple*. This tactic prepares the proof goal for the prover by applying an appropriate induction scheme and extracting the induction step. After the prover has solved this step case the resulting rippling proof will be translated back into a NuPRL sequent.

An application of NuPRL's induction scheme for natural numbers $\mathbb{N}$ yields as step case a subgoal of the form $x{-}1 \mapsto x$. This means that an additional function symbol '$-$' occurs in the hypothesis which cannot be handled directly by the rippling calculus. We have developed a simulation of the step case $x \mapsto x{+}1$ in NuPRL which is admissible for rippling. Furthermore, NuPRL's induction scheme for list types $TList$ is also supported by our rippling prover.

Before applying an induction scheme the induction variable is moved in front of other universally quantified variables in order to maximize the number of sink variables. After the step case is proved the translation back into a sequent proof has to be done. In [25] a translation for rippling-out proofs was developed, which can be used for arbitrary sequences of rewrite rules. It is implemented as meta-tactic and uses the basic refinement rules *cut*, *substitution* and *lemma*. We have extended this approach with the following concepts [23]:

1. The (universally quantified) induction hypothesis can be instantiated.
2. Polymorphic types for integration of rewrite steps can be reconstructed.
3. Premises in a NuPRL-sequent can be used as rewrite rules.

The first improvement is necessary for completing rippling-in proofs. The hypothesis has to be instantiated with sink terms which have been rippled into the sink positions of the induction conclusion. To complete a rippling-out simulation with optimization for equality '=' (see Section 3.2) the generalized sink positions have to be unified by using the induction hypothesis as a rewrite rule.

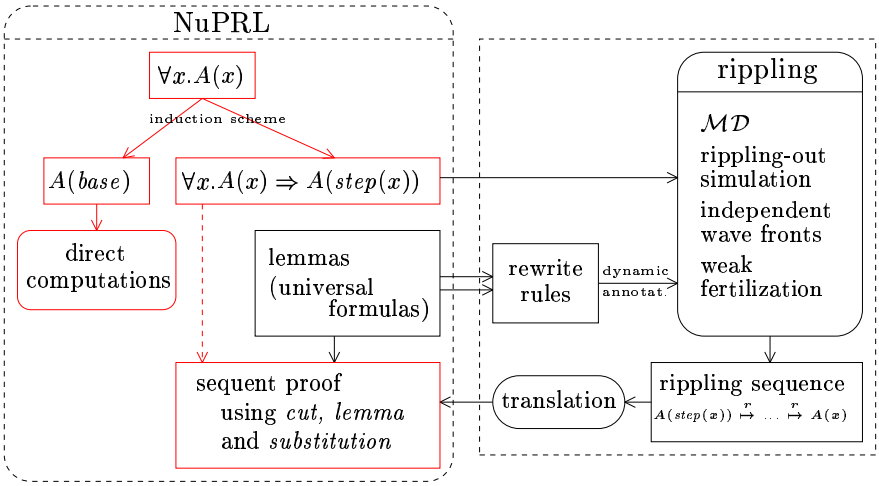


Fig. 4. Components and integration of the rippling module

The second extension determines the type and universe level for a substitution rule by analyzing the proof. This type reconstruction is necessary since the external rippling prover is untyped. A temporary proof goal will be generated in order to compute the type for a successful application of the substitution. Then the goal is forced to fail and the extracted type information will be used for the original proof.

The last improvement allows premises of the current proof goal to be used as wave rules if they are in NuPRL's *universal formula format* [16]. So second order proofs over universally quantified functions can be established by using the recursive definitions of these functions in the premises as wave rules.

Many additional improvements have been made for adapting the basic translation approach to the rippling-distance strategy. Furthermore, NuPRL's tactics *BackThruLemma* and *BackThruHyp* for backward chaining in universal formulas are applied to support a uniform translation of the rippling steps wrt. equality-, implication- and hypothesis-axioms. The components of the rippling module and its integration into the NuPRL system are summarized in Fig. 4.

In future work we will adapt NuPRL's induction scheme for integers $\mathbb{Z}$ to an admissible induction scheme for the rippling calculus. This can be realized by simply extending the presented adaption for natural numbers $\mathbb{N}$ to $\mathbb{Z}$. Furthermore, a library of measures $\mathcal{MX}$ and corresponding reduction orderings $\prec_x$ will be built for realizing special extensions. In the current implementation there are two alternative measures, one for fast proofs in large terms and the other for more complicated proofs. The latter allows us to use additional associativity and commutativity rules for normalizing wave fronts which is necessary for unblocking proof search. In the current state of the system the user has to specify the measure which should be used but this can be done automatically as soon as syntactic and functional characterizations have been developed.

4 High-Level Synthesis Strategies

The theorem proving techniques described in the previous sections operate on a rather low level of abstraction and have only little to do with the way in which a programmer would reason when developing a program. The application of these methods is therefore restricted to programming problems which are conceptually simple and can be solved completely automatically.

Techniques which are to support the synthesis of larger programs, however, will depend on a cooperation between programmer and machine. A programmer will have to control and guide the derivation process while the system will fill in the formal details and ensure the correctness of the generated algorithms. The corresponding proof techniques have to operate on a higher level of abstraction and must be based on comprehensible formalizations of application domains and programming concepts rather than on low-level inferences of the logical calculus.

Algorithm design strategies based on schematic solutions for certain classes of algorithms [40] have proved to be suited best for this purpose since they can be formulated almost entirely in programmer's terminology. It has been demonstrated [41] that algorithm schemata do not only lead to a very efficient synthesis process but can also produce competitive algorithms if properly guided.

Formally verified theorems stating the requirements for the correctness of an abstract program scheme [19] are the key for an integration of these strategies into the general framework. Such theorems can be applied like high-level inference rules which decompose the synthesis task into the task of proving instances of the given axioms. The latter can then be solved by first-order theorem provers, simple inductions, applications of additional theorems, or knowledge-base queries. The conceptually difficult problem – generating the algorithm and proving it correct – has been solved once and for all while proving the formal theorem and requires only a single step in the synthesis process. In this section we shall illustrate how this methodology is used for integrating a strategy for the design of *global search algorithms* [39] into the uniform proof system.

4.1 Formalizing the Design of Global Search Algorithms

Solving a problem by enumerating candidate solutions is a well-known concept in computer science. *Global search* is a concept that generalizes binary search, backtracking, branch-and-bound, and other methods which explore a search space by looking at whole sets of possible solutions at once.

The basic idea of global search, illustrated in Fig. 5, is to combine enumeration and elimination processes. Usually, global search has to compute the complete set of output values for a given input. Global search systematically enumerates a search space which must contain the set of all output values (a) and tests if certain elements of the search space satisfy the output-condition (b). The latter is necessary to guarantee correctness but is too fine-grained to achieve efficiency, particularly if the search space is much bigger than the set of solutions. Therefore whole regions of the search space are filtered out during the enumeration process if it can be determined that they cannot contain output values (c).

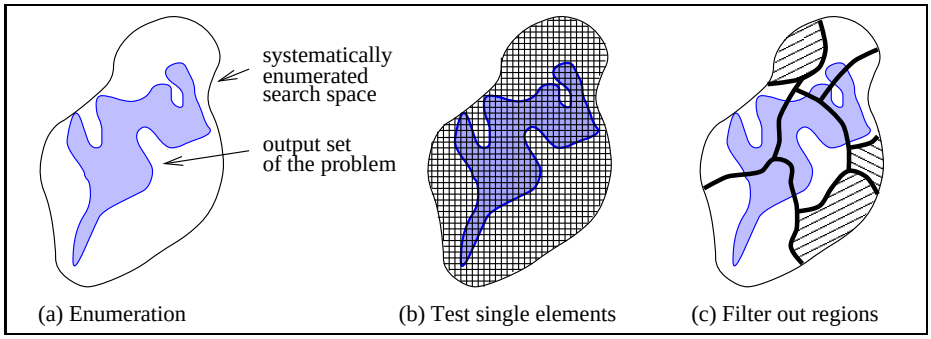


Fig. 5. Global Search as elimination process

In order to synthesize global search algorithms from specifications, we have to formalize their general structure as an abstract program scheme and to describe techniques for automatically generating appropriate enumeration and filtering processes. We begin by fixing the notation for general programming concepts.

A programming problem is usually characterized by the domain D of the desired program, its range R, a condition I on acceptable input values x, and a condition O on the returned output values z. Formally, a *specification* can be described as 4-tuple $spec = (D, R, I, O)$ where D and R are data types, I is a predicate on D, and O is a predicate on $D \times R$. A specification and a (possibly partial) computable function $body : D \rightharpoonup \mathtt{Set}(R)$ together form a *program*. A program is *correct* if it computes the complete set of output values for each acceptable input $(\forall x{:}D.\ I(x) \Rightarrow body(x) = \{z{:}R \mid O(x,z)\})$. A specification is *satisfiable* if it can be extended into a correct program. As in [40] we use a formal notation for programs which emphasizes that we are interested in computing the set of *all* solutions of a given problem (assuming that there are finitely many):

$$\mathtt{FUNCTION}\ f(x{:}\,D){:}\,\mathtt{Set}(R)\ \mathtt{WHERE}\ I(x)\ \mathtt{RETURNS}\ \{z \mid O(x,z)\} \equiv body(x).$$

The name f can be used in the body in order to describe recursive algorithms. Often we use only the left side to denote specifications in a more readable way.

All the above concepts, including an ML-like mathematical notation for computable functions, can be straightforwardly formalized in the logical language of NuPRL (see [19, section 2]) and are the formal foundation for the automated derivation of global search algorithms within the integrated synthesis system.

A careful analysis in [39] (later refined and formalized in [18, 19]) has shown that the common structure of global search algorithms can be expressed by a pair of abstract programs which is presented in Fig. 6. These programs contain placeholders D, R, I, and O for a specification and seven additional components $S, J, s_0, sat, split, ext, \Phi$ which are specific for a global search algorithm. On input x this algorithm starts investigating an initial search space $s_0(x)$ and passes it through the filter Φ which globally checks whether a search region s contains solutions. Using the auxiliary function f_{gs} the algorithm then repeatedly extracts candidate solutions ($ext(s)$) for testing and splits a search space s into a set $split(x, s)$ of subspaces which are again passed through the filter Φ. Subspaces which survive the filter contain solutions and are investigated recursively.

```
FUNCTION f(x : D):Set(R) WHERE I(x)  RETURNS {z | O(x, z)}
≡ if Φ(x, s0(x))  then fgs(x, s0(x))  else []

FUNCTION fgs(x, s : D×S):Set(R)  WHERE I(x) ∧ J(x, s) ∧ Φ(x, s)
   RETURNS {z | O(x, z) ∧ sat(z, s)}
≡ let immediate_solutions =
       let extracted_candidates = ext(s) in
         filter (λz.O(x, z)) extracted_candidates
   and recursive_solutions =
       let filtered_subspaces = filter (λt.Φ(x, t)) (split(x, s)) in
         flatten (map (λt.fgs(x, t)) filtered_subspaces)
   in append immediate_solutions recursive_solutions
```

Fig. 6. Structure of Global Search algorithms

For the sake of efficiency, search spaces are represented by *space descriptors* $s \in S$ instead of sets of values $z \in R$ and the fact that a value z belongs to the space described by s is denoted by a predicate $sat(z, s)$. The predicate $J(x, s)$ expresses that s is a *meaningful* search space descriptor for the input x. Formally S must be a data type. J and Φ must be predicates on $D \times S$ and *sat* one on $R \times S$. $s_0 : D \rightarrow S$, $split : D \times S \rightarrow S$, and $ext : S \rightarrow \mathtt{Set}(R)$ must be computable functions.

Six requirements, formalized in Fig. 7, must be satisfied to ensure the correctness of global search algorithms. The initial descriptor $s_0(x)$ must be meaningful (1) and splitting must preserve meaningfulness (2). All solutions must be contained in the initial search space (3) and be extractable after splitting finitely many times (4). Subspaces containing solutions must pass the filter (5) and filtered splitting, the combined enumeration/elimination process, must eventually terminate (6). In [39] (refined in [18, 19]) the following theorem has been proved.

Theorem 1. *If D, R, I, O, S, J, sat, s_0, split, ext, and Φ fulfill the axioms of global search then the pair of programs given in Fig. 6 is correct and satisfies the specification* $\mathtt{FUNCTION}\ f(x : D) : \mathtt{Set}(R)\ \mathtt{WHERE}\ I(x)\ \mathtt{RETURNS}\ \{z \mid O(x, z)\}$.

Thus a global search algorithm can be synthesized for a given specification by deriving seven components S, J, *sat*, s_0, *split*, *ext*, and Φ which satisfy the axioms of global search and instantiating the abstract programs accordingly.

4.2 Knowledge Based Algorithm Construction

A direct derivation of global search algorithms on the basis of theorem 1 is obviously a difficult task. The theorem does not provide information how to *find* the seven additional components and the verification of the six axioms, particularly of axioms 4 and 6 which require induction, would put a heavy load on the proof process – even if the techniques mentioned in sections 2 and 3 were already integrated into the derivation strategy. Instead, it is much more meaningful to base the construction of global search algorithms on general knowledge about algorithmic structures. For a given range type R, for instance, there are usually only a few general techniques to enumerate search spaces and each global search

$\forall$x: D. $\forall$z: R. $\forall$s: S.

1. $I(\mathrm{x}) \Rightarrow J(\mathrm{x}, s_0(\mathrm{x}))$
2. $I(\mathrm{x}) \wedge J(\mathrm{x},\mathrm{s}) \;\Rightarrow\; \forall \mathrm{t} \in \mathit{split}(\mathrm{x},\mathrm{s}).\ J(\mathrm{x},\mathrm{t})$
3. $I(\mathrm{x}) \wedge O(\mathrm{x},\mathrm{z}) \;\Rightarrow\; \mathit{sat}(\mathrm{z}, s_0(\mathrm{x}))$
4. $I(\mathrm{x}) \wedge J(\mathrm{x},\mathrm{s}) \wedge O(\mathrm{x},\mathrm{z}) \;\Rightarrow\; \mathit{sat}(\mathrm{z},\mathrm{s}) \iff \exists \mathrm{k}.\ \exists \mathrm{t} \in \mathit{split}^{\mathrm{k}}(\mathrm{x},\mathrm{s}).\ \mathrm{z} \in \mathit{ext}(\mathrm{t})$
5. $I(\mathrm{x}) \wedge J(\mathrm{x},\mathrm{s}) \;\Rightarrow\; \Phi(\mathrm{x},\mathrm{s}) \Leftarrow \exists \mathrm{z}.\ \mathit{sat}(\mathrm{z},\mathrm{s}) \wedge O(\mathrm{x},\mathrm{z})$
6. $I(\mathrm{x}) \wedge J(\mathrm{x},\mathrm{s}) \;\Rightarrow\; \exists \mathrm{k}.\ \mathit{split}^{\mathrm{k}}_{\Phi}(\mathrm{x},\mathrm{s}) = \emptyset$

where $\mathit{split}_{\Phi}(\mathrm{x},\mathrm{s}) = \{\mathrm{t} \in \mathit{split}(\mathrm{x},\mathrm{s}) \mid \Phi(\mathrm{x},\mathrm{t})\}$

and $\quad \mathit{split}^{\mathrm{k}}_{\Phi}(\mathrm{x},\mathrm{s}) = \begin{cases} \mathrm{s} & \text{if } \mathrm{k} = 0 \\ \bigcup_{\mathrm{t} \in \mathit{split}^{\mathrm{k}-1}_{\Phi}(\mathrm{x},\mathrm{s})} \mathit{split}_{\Phi}(\mathrm{x},\mathrm{t}) & \text{if } \mathrm{k} > 0 \end{cases}$

Fig. 7. Axioms of Global Search

algorithm will use a special case of such a technique. Therefore it makes sense to store information about generic enumeration processes in a knowledge base and to develop techniques for adapting them to a particular programming problem.

The investigations in [39] have shown that standard enumeration structures for some range type R can be stored in a knowledge base as objects of the form $G = ((D_G, R, I_G, O_G), S, J, s_0, \mathit{sat}, \mathit{split}, \mathit{ext})$ which are proved to satisfy axioms 1 to 4. Such objects are called *GS-theories*. A *problem reduction* mechanism will make sure that the four axioms are preserved when the enumeration structure is specialized to a given specification which involves the same range.

Specializing a standard GS-theory G works as follows. Its specification $\mathit{spec}_G = (D_G, R_G, I_G, O_G)$ characterizes a general enumeration method f_G which explores the space R_G as far as possible. Specialization simply means to avoid enumerating elements which are not needed and results in a kind of "truncated" enumeration structure for the same type. Formally, spec_G can be *specialized* to a given specification $\mathit{spec} = (D, R, I, O)$ if the the following condition is valid:

$$R{=}R_G \;\wedge\; \forall \mathrm{x}{:}\, D.\ I(\mathrm{x}) \;\Rightarrow\; \exists \mathrm{x}_G{:}\, D_G.\, I_G(\mathrm{x}_G) \;\wedge\; \forall \mathrm{z}{:}\, R.\, O(\mathrm{x},\mathrm{z}) \Rightarrow O_G(\mathrm{x}_G,\mathrm{z})$$

Thus specialization also allows to adapt the *input* of a problem since the original input x is mapped to a value x_G which serves as input for the search performed by f_G. A proof of the above condition implicitly contains a substitution $\theta{:}D{\to}D_G$ which maps x to x_G. θ can be extracted from the proof and then be used for refining f_G into a search method with inputs from D instead of D_G. On the *output* side of the problem specialization restricts f_G to the computation of values which satisfy the stronger condition O. Technically, this can be done by checking O for each computed value. Altogether problem reduction allows us to create a global search algorithm f for the specification *spec* by defining

$$f(\mathrm{x}) = \{\mathrm{z} \in f_G(\theta(\mathrm{x})) \mid O(\mathrm{x},\mathrm{z})\}.$$

For the sake of efficiency, the modifications caused by θ and O will be moved directly into the components of the algorithm. By an index θ as in J_θ or split_θ we indicate that the transformation θ is applied to all arguments expecting a domain value from D, e.g. $\mathit{split}_\theta(\mathrm{x},\mathrm{s}) = \mathit{split}(\theta(\mathrm{x}),\mathrm{s})$.

Specializing predefined GS-theories allows us to derive six components of a global search algorithm which satisfy axioms 1 to 4 with comparably little effort. In a similar way, we can avoid having to prove the sixth axiom explicitly. For each enumeration structure there are only a few standard methods to ensure termination through an elimination process. In [18, 19] it has been shown that these methods can be stored in the form of filters Φ for a GS-theory G which are proved to satisfy axiom 6. Such filters will be called *well-founded* wrt. G and this property will be preserved during specialization as well. Thus specialization reduces the proof burden to checking that, after specialization, the selected filter is *necessary* wrt. the GS-theory, i.e. that it satisfies axiom 5.

The process of adapting the search space to the specific problem can be completely formalized and expressed in a single reduction theorem.

Theorem 2. *Let* $G{=}((D_G, R, I_G, O_G), S, J, s_0, sat, split, ext)$ *be a GS-theory. Let* $spec{=}(D, R, I, O)$ *be a specification such that* $spec_G{=}(D_G, R, I_G, O_G)$ *can be specialized to spec. Let* θ *be the substitution extracted from the specialization proof. Then* $G_\theta = ((D, R, I, O), S, J_\theta, s_{0\theta}, sat, split_\theta, ext)$ *is a GS-theory. Furthermore if* Φ *is a well-founded filter wrt.* G *then* Φ_θ *is well-founded wrt.* G_θ

Adapting standard algorithmic knowledge to a given problem moves most of the proof burden into the creation of the knowledge base and keeps the synthesis process itself comparatively easy. Information retrieved from the knowledge base will provide all the basic components and guarantee that axioms 1 to 4 and 6 are satisfied. Only the specialization condition and the necessity of the specialized filter – conditions whose proofs are much easier than those of axioms 4 and 6 – need to be checked explicitly. These insights led to the following strategy for synthesizing global search algorithms from formal specifications (see [19, Section 4.4] for an application example).

Strategy 3. *Given the specification*
FUNCTION $f(\mathrm{x}{:}D){:}\mathtt{Set}(R)$ WHERE $I(\mathrm{x})$ RETURNS $\{\mathrm{z} \mid O(\mathrm{x},\mathrm{z})\}$

1. *Select a GS-theory* $G{=}((D_G, R, I_G, O_G), S, J, s_0, sat, split, ext)$ *for* R.
2. *Prove that* $spec_G{=}(D_G, R, I_G, O_G)$ *can be specialized to the specification. Derive a substitution* $\theta{:}D{\to}D_G$ *from the proof and specialize* G *with* θ.
3. *Select a well-founded filter* Φ *for* G *and specialize it with* θ.
4. *Prove that the specialized filter is necessary for the specialized GS-theory.*
5. *Instantiate the program scheme given in Fig. 6.*

It should be noted that in step 4 the specialized filter could be refined by heuristically adding further conditions which do not destroy its necessity. In some case this can drastically improve the efficiency of the generated algorithm.

4.3 Integrating the Design Strategy into Deductive Systems

So far we have described the synthesis of global search algorithms only semi-formally in order to illustrate the fundamental ideas. Integrating the strategy into a proof based system now requires a more rigorous approach. Each step in a

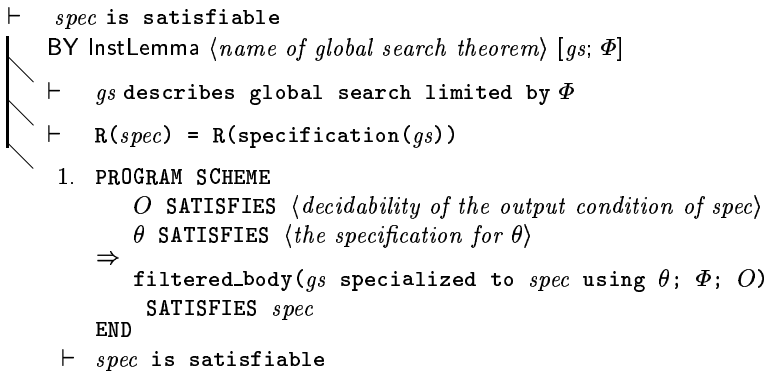


Fig. 8. Principal structure of a proof generated by the Global Search tactic

derivation must be completely formal such that it can be controlled by the proof system. On the other hand, each derivation step should remain on the high level of abstraction which we have used so far. In the following we will explain the techniques by which these two requirements could be achieved.

The *application of formally verified theorems* is one of the most important principles which make program synthesis within a formal proof system like NuPRL feasible (see [19, Section 3] for a detailed exposition). In such systems all derivations must eventually be based on primitive inference rules. Formal theorems, however, can serve as *derived inference rules* on a much higher level of abstraction. Their application corresponds to a single, conceptually large inference step which would require thousands of elementary proof steps if performed purely on the level of primitive inferences. In order to represent rule *schemes*, these theorems contain universally quantified variables which must be instantiated by concrete values *before* the theorem can be applied. Finding appropriate values is the only difficult aspect of this technique.

The kernel of our implementation of the global search strategy, for instance, is a single formal theorem which combines theorems 1 and 2. It quantifies over variables for GS-theories (*gs*), filters (Φ), transformations (θ), and specifications (*spec*) which must be instantiated by a proof tactic for synthesizing global search algorithms. The instance for *spec* is obvious. G and Φ should be provided manually since their selection from a restricted set of alternatives is a design decision rather than a deductive task. The function θ, however, should be derived fully automatically, since it does not introduce any new algorithmic idea but is determined by the specialization conditions. The value for θ is only clear after these conditions have been investigated.

The different nature of these variables had be taken into consideration while developing a NuPRL-tactic for deriving global search algorithms. In general, the design of proof tactics should correspond to the structure of the proofs they generate. As proofs are typically divided into subproofs for certain subgoals, the tactic should be organized in the same manner and provide subtactics for the different tasks. The handling of variables representing design decisions has to be moved to the beginning of the proof, since they pre-structure the rest of it. Almost all

tactics have to reflect the structure of the terms to which they are applied, since the primitive rules only allow a *structural* analysis or synthesis of terms.

This leads to a fixed anatomy on the top-level of a proof, as illustrated in Fig. 8. A typical synthesis proof begins by stating that an algorithm for a specification *spec* shall be found. It then instantiates the global search theorem by invoking the tactic InstLemma which requires the 'design parameters' *gs* and Φ, denoting the concrete GS-theory and the filter, to be provided manually. This results in three preconditions for the initial formula (NuPRL proceeds top-down). The first says that *gs* is valid, i.e. fulfills the axioms 1 to 4, and that Φ makes the search space well-founded. These assumptions are usually shown by referring to lemmas, since *gs* and Φ are selected from a few alternatives whose properties are stored as lemmas in the NuPRL library. The second subgoal states that the range types of the specification and the GS-theory are identical. The third expresses that the algorithmic scheme introduced by *gs* and Φ can be adapted to *spec*. Here the specialization to be performed is described in terms of so-called *program schemes*. By this we emphasize that θ is the missing *algorithmic* link between the schematic search space and the final program for *spec*.

Program schemes express that a complex problem can directly be reduced to simpler problems. The part after the implication symbol in Fig. 8 describes how the algorithm for the complex problem is formed by algorithms for the simpler ones. The latter may occur as variables which have to be declared before the implication symbol. The `SATISFIES` clause specifies the problem associated with the variable to be solved. Thus the final program `filtered_body(...)` can be constructed as soon as the two auxiliary algorithms O and θ are given.[4] The *specification for* θ as algorithm contains all conditions in which θ finally occurs, i.e. the conditions for specialization and necessity. Necessity is usually viewed as property of the filter Φ but, since specialization prunes the search space, we have to check whether the *combination* of Φ and θ results in a necessary filter.

The overall effect of the concept of *program schemes* is that we can formulate the final program although two auxiliary algorithms are yet unknown. Applying the basic theorem therefore corresponds to a macro reduction step transforming a complicated initial specification into one or more simpler subproblems. This results in improved comprehensibility of both the proof and the tactic.

Automatic proof methods are especially valuable for solving the third subgoal. We have already pointed out that θ can be only derived *together* with the proof showing that θ satisfies its specification. This means that we need a separate proof search phase, in which we find out how to successfully solve the goal, before we can actually construct the proof. The methods discussed in sections 2 and 3 have these capabilities, and we will investigate to what extent they can be applied to subgoals of the discussed kind. In the KIDS system, term rewriting is successfully used to prove these conditions. Term rewriting can again be realized by theorem application. But in this case the theorems have a much simpler structure and instances for the quantified variables can almost always be found

[4] When presenting the global search method on paper one easily overlooks the fact that the predicate O must be transformed into a computable function.

by first-order matching.

One should now be able to imagine what had to be done to fully implement the global search strategy on a platform like NuPRL. As logical calculi provide only primitive concepts, we first had to increase their level of abstraction by representing standard data types like lists, finite sets, etc. as well as the corresponding operations and their laws. This allows us to formulate simple programs and to prove their properties. Next, in order to reason about programs as such, we had to implement concepts like *program*, *specification* and related notions which were formalized in [18, chapter 2]. The conditions for specialization and filtering reside on a similar level. They are used in the axioms of the GS-theories which had to be implemented on the basis of the specific data structure containing the different components. Furthermore, we need the fundamental global search theorem and its proof. Finally, relevant GS-theories together with the associated well-foundedness filters had to be represented. In both cases, lemmas have to guarantee that the required properties are fulfilled. Together, these definitions and lemmas describe the formal knowledge about global search.

All the rest is tactic development. The global search strategy, as explained above, had to be built from the top-level tactic and many subtactics solving specific subtasks. Moreover, tactics had to be written for proving the laws of the data types, the global search theorem, the properties of different GS-theories and filters. Although these proofs have to be constructed only once, it does not make sense to create them without automatic support. We expect that integrating the techniques discussed in sections 2 and 3, especially induction, into our tactics will be very helpful for this purpose.

Once the global search tactic has found a proof, we can extract a correct algorithm from it using NuPRL's standard extraction mechanism. The KIDS system has shown that the efficiency of this algorithm can dramatically be improved by postprocessing steps. It needs to be investigated where to place such a step in a system which relies of derivations in a formal logic.

In our current implementation we have completed only the essential parts of the global search strategy in NuPRL. Especially, only the proofs for the relevant library theorems have been implemented, because the necessary proof methods were not yet integrated. Instead, while formalizing the various concepts, we have focused on the question whether type theory, despite the many formal details, is suitable for practical program synthesis. As it turned out, the strict semantics of type theory led to deeper insights into some of the concepts: it forced us to change their formulation since the obvious one often meant something different or was not even well-formed. Our work also led to improvements of some standard tactics of NuPRL: in the term rewriting tactic we can often infer type information automatically. By this the chances for success have been dramatically increased; an automatic invocation of such a tactic is now imaginable. This allows handling many difficulties resulting from the peculiarities of the calculus. We believe that the formal complications can be further limited by integrating additional proof methods which will make program synthesis feasible even within the strict framework of proof systems.

5 Conclusion

We have presented the design of the program synthesis system MAPS which integrates a variety of techniques from automated theorem proving and algorithm design at different levels of abstraction. We have demonstrated how proof procedures for (constructive) propositional, first-order logic, and induction as well as schema-based algorithm design strategies can be embedded into a single framework for automated proof and program development.

Because of the rigorous formal framework into which all these methods are embedded, executing the individual techniques is somewhat less efficient than separate implementations. We believe however that the integrated approach is the safest way of combining them into an automated reasoning system which can deal with many of the problems occurring during a formal program derivation.

Future work will involve a more efficient implementation of the individual techniques and support for a stronger cooperation *between* the high- and low-level methods. We are currently elaborating a method for extracting programs *directly* from matrix and induction proofs. We also intend to deepen our studies on induction techniques and to integrate additional algorithm design strategies using the same methodology. We will also work on supporting several existing functional, logical, and imperative programming languages as a target language of our derivations. Recent work on embedding the Objective Caml programming language into NuPRL's formal language [20] has shown that the practical usefulness of systems for program synthesis and transformation can be drastically increased by such efforts.

References

1. D. Basin & T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1-2), pp. 147-180, 1996.
2. J. L. Bates & R. L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, 1985.
3. J. Van Benthem Correspondence Theory In D. Gabbay & F. Guenther, eds., *Handbook of Philosophical Logic*, II, pp. 167–247, Reidel, 1984.
4. W. Bibel. On matrices with connections. *Journal of the ACM*, 28(633–645), 1981.
5. W. Bibel. *Automated Theorem Proving.* Vieweg Verlag, 1987.
6. W. Bibel. Toward predicative programming. In M. R. Lowry & R. McCartney, eds., *Automating Software Design*, pp. 405–424, AAAI Press / The MIT Press, 1991.
7. W. Bibel, S. Brüning, U. Egly, T. Rath. Komet. In *12^{th} Conference on Automated Deduction*, LNAI 814, pp. 783–787. Springer Verlag, 1994.
8. W. Bibel, D. Korn, C. Kreitz, S. Schmitt. Problem-oriented applications of automated theorem proving. In J. Calmet & C. Limongelli, eds., *Design and Implementation of Symbolic Computation Systems*, LNCS 1126, Springer Verlag, pp. 1–21, 1996.
9. A. Bundy, F. van Harmelen, A. Ireland, A. Smaill. Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence*, 1992.
10. R. L. Constable, et. al. *Implementing Mathematics with the NuPRL proof development system.* Prentice Hall, 1986.
11. M. Davis & H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
12. M. Davis, G. Logemann, D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
13. M. C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer Verlag, 1990.
14. G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

15. P. Jackson. NuPRL's Metalanguage ML. Reference Manual and User's Guide, Cornell University, 1994.
16. P. Jackson. The NuPRL Proof Development System, Version 4.1. Reference Manual and User's Guide, Cornell University, 1994.
17. D. Korn & C. Kreitz. Deciding intuitionistic propositional logic via translation into classical logic. In W. McCune, ed., *14^{th} Conference on Automated Deduction*, LNAI 1249, pp. 131–145, Springer Verlag, 1997.
18. C. Kreitz. *METASYNTHESIS: Deriving Programs that Develop Programs.* Thesis for Habilitation, TH Darmstadt, 1992. Forschungsbericht AIDA–93–03.
19. C. Kreitz. Formal mathematics for verifiably correct program synthesis. *Journal of the IGPL*, 4(1):75–94, 1996.
20. C. Kreitz. Formal reasoning about communication systems I: Embedding ML into type theory. Technical Report TR 97-1637, Cornell University, 1997.
21. C. Kreitz, H. Mantel, J. Otten, S. Schmitt. Connection-Based Proof Construction in Linear Logic. In W. McCune, ed., *14^{th} Conference on Automated Deduction*, LNAI 1249, pp. 207–221, Springer Verlag, 1997.
22. C. Kreitz, J. Otten, S. Schmitt. Guiding Program Development Systems by a Connection Based Proof Strategy. In M. Proietti, ed., *5^{th} International Workshop on Logic Program Synthesis and Transformation*, LNCS 1048, pp. 137–151. Springer Verlag, 1996.
23. F. Kurucz. Realisierung verschiedender Induktionsstrategien basierend auf dem Rippling-Kalkül. Diplomarbeit, TH Darmstadt, 1997.
24. R. Letz, J. Schumann, S. Bayerl, W. Bibel. Setheo: A high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.
25. T. van thanh Liem. Induktion im NuPRL System. Diplomarbeit, TH Darmstadt, 1996.
26. R. C. Moore. Reasoning about Knowledge and Action *IJCAI-77*, pp 223–227, 1977.
27. H. J. Ohlbach. Semantics–Based Translation Methods for Modal Logics *Journal of Logic and Computation*, 1(6), pp 691–746, 1991.
28. J. Otten. ileanTAP: An intuitionistic theorem prover. In Didier Galmiche, ed., *International Conference TABLEAUX '97*. LNAI 1227, pp. 307–312, Springer Verlag, 1997.
29. J. Otten. On the advantage of a non-clausal Davis-Putnam procedure. Forschungsbericht AIDA–97–01, TH Darmstadt, 1997.
30. J. Otten & C. Kreitz. A connection based proof method for intuitionistic logic. In P. Baumgartner, R. Hähnle, J. Posegga, eds., *4^{th} Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, LNAI 918, pp. 122–137, Springer Verlag, 1995.
31. J. Otten & C. Kreitz. T-String-Unification: Unifying Prefixes in Non-Classical Proof Methods. In U. Moscato, ed., *5^{th} Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, LNAI 1071, pp. 244–260, Springer Verlag, 1996.
32. J. Otten & 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 Verlag, 1996.
33. D. Plaisted & S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
34. S. Schmitt. Avoiding redundancy in proof reconstruction *1^{st} International Workshop on Proof Transformation and Presentation*, Schloß Dagstuhl, Germany, 1997.
35. S. Schmitt. Building Efficient Conversion Procedures using Proof Knowledge. Technical Report, TH Darmstadt, 1997.
36. S. Schmitt & C. Kreitz. On transforming intuitionistic matrix proofs into standard-sequent proofs. In P. Baumgartner, R. Hähnle, J. Posegga, eds., *4^{th} Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, LNAI 918, pp. 106–121. Springer Verlag, 1995.
37. S. Schmitt & C. Kreitz. Converting non-classical matrix proofs into sequent-style systems. In M. McRobbie & J. Slaney, eds., *13^{th} Conference on Automated Deduction*, LNAI 1104, pp. 418–432. Springer Verlag, 1996.
38. S. Schmitt & C. Kreitz. A uniform procedure for converting non-classical matrix proofs into sequent-style systems. Technical Report AIDA-96-01, TH Darmstadt 1996.
39. D. R. Smith. Structure and design of global search algorithms. Technical Report KES.U.87.12, Kestrel Institute, 1987.
40. D. R. Smith & M. R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14(2-3):305–321, 1990.
41. D. R. Smith & E. A. Parra. Transformational approach to transportation scheduling. *8^{th} Knowledge-Based Software Engineering Conference*, pp. 60–68, 1993.
42. G. Stolpmann. Datentypen und Programmsynthese. Studienarbeit, TH Darmstadt, 1996.
43. G. Stolpmann. Schematische Konstruktion von Globalsuchalgorithmen. Diplomarbeit, TH Darmstadt, 1997.
44. L. Wallen. *Automated deduction in nonclassical logic.* MIT Press, 1990.