Extensions of Constraint Solving for Proof Planning

Erica Melis[1] **Jürgen Zimmer**[1] **Tobias Müller**[2]

Abstract. The integration of constraint solvers into proof planning has pushed the problem solving horizon. Proof planning benefits from the general functionalities of a constraint solver such as consistency check, constraint inference, as well as the search for instantiations. However, off-the-shelf constraint solvers need to be extended in order to be be integrated appropriately: In particular, for correctness, the context of constraints and Eigenvariable-conditions have to be taken into account. Moreover, symbolic and numeric constraint inference are combined. This paper discusses the extensions to constraint solving for proof planning, namely the combination of symbolic and numeric inference, first-class constraints, and context trees. We also describe the implementation of these extensions in the constraint solver $\mathcal{CoSIE}$.

Keywords: theorem proving, constraint solving.

1 INTRODUCTION

The construction of many mathematical proofs requires a combination of purely logical inference steps with specialized reasoning such as computing integrals, solving inequalities, and constructing mathematical objects with specific properties.

In particular, difficulties with purely logical proofs can be caused by the need to construct mathematical objects (witnesses) with theory-specific properties.

In fact, the situation in mathematical proof construction is contrary to the original aim of logic-oriented automated theorem proving: while logic-based automated theorem provers try to construct proofs solely by derivation of rules from a specific calculus without incorporating already known facts, mathematicians try to draw as much as possible on facts that have already been derived for a particular mathematical domain. This insight gave rise to the investigation of theory reasoning (as for instance, of theory resolution [18] and constrained resolution [3]) and to knowledge-based proof planning [9].

In knowledge-based proof planning external reasoners can be integrated. In particular, a domain-specific constraint solver can help to construct mathematical objects that are elements of a specific domain. As long as these mathematical objects are still unknown in the problem solving process, they are represented by place holders called *problem variables*.

[1] Universität des Saarlandes, Fachbereich Informatik, D-66041 Saarbrücken, Germany
[2] Programming Systems Lab, Postfach 15 11 50, Universität des Saarlandes, D-66041 Saarbrücken, Germany

Proceedings of the 14th European Conference on Artificial Intelligence, pages 229–233, Berlin, August 2000. IOS Press.

The tasks of a constraint solver in proof planning are the collection and propagation of constraints as long as they are consistent, the detection of inconsistencies for the restriction of the search space, the computation of an *answer constraint* formula that expresses the collected constraints, and finally the delivery of witnesses for the problem variables.

As a fundamental extension of our description in [8] this paper presents the reasons and the realization of the extensions of off-the-shelf constraint solvers that are necessary for its use in proof planning. We shall explain these extensions in general terms and with respect to their implementation in the concurrent constraint language Oz [17].

In what follows, Δ, Φ, and Ψ denote sets of formulas. We use standard Prolog notation to name variables (in capital letters) and constants (in lower case letters).

2 CONSTRAINT SOLVING

Constraint solvers are systems that use very efficient (theory-specific) data structures and algorithms to perform consistency and entailment checks. Many off-the-shelf constraint solvers are designed to tackle combinatorial problems in which all problem variables are introduced at the beginning and the solver submits the problem to a monolithic search engine that tries to find a solution without any interference from outside.

An established model for (propagation-based) constraint solving [17] involves numeric constraint inference over a *constraint store* holding so-called *basic* constraints over a domain as, for example, the domain of real numbers. A basic constraint is of the form $X = v$ (X is bound to a value v of the domain), $X = Y$ (X is equated to another variable Y), or $X \in B$ (X takes its value in B, where B is an approximation of a value of the respective domain). Attached to the constraint store are *non-basic* constraints. Non-basic constraints, as for example $X + Y = Z$ over integers or real numbers, are more expressive than basic constraints and, hence, require more computational effort. A non-basic constraint is realized by a computational agent, a *propagator*, observing the basic constraints of its parameters which are variables in the constraint store (in the example X, Y, and Z). The purpose of a propagator is to infer new basic constraints for its parameters and add them to the store. That happens until no further basic constraints can be inferred and written to the store, i.e. until a fix-point is reached. Inference can be resumed by adding new constraints either basic or non-basic. A propagator is terminated if it is either inconsistent with the constraint store or explicitly represented by the basic constraints in the store, i.e. entailed by the store.

The common functionalities of constraint solvers are consistency check, entailment check, reflection, and search for instantiations. The (In)consistency check includes the propagation of constraints combined with the actual consistency algorithm, e.g., with arc-consistency AC3 [7].

3 PROOF PLANNING

As opposed to classical theorem proving that is based on exhaustive search at the calculus-level, proof planning employs abstract plan operators, called *methods* that encapsulate (mathematical) proof techniques such as diagonalization and induction.

The idea underlying proof planning is that of classic AI-planning [4]. A planning state is a set of sequents that is divided into *goals* and *assumptions*. A proof planning problem is defined by an initial state specified by the proof assumptions and the goal g given by the theorem to be proved. The theorem and proof assumptions are represented by sequents. [3] For instance, for proving the theorem LIM+ which states that the limit of the sum of two real-valued functions f and g at a point $a \in \mathbb{R}$ is the sum of their limits the initial planning state consists of the goal

$$\emptyset \vdash \lim_{x \to a} f(x) + g(x) = L_1 + L_2$$

and of the proof assumptions

$$\emptyset \vdash \lim_{x \to a} f(x) = L_1 \text{ and}$$
$$\emptyset \vdash \lim_{x \to a} g(x) = L_2.$$

After the expansion of the definition of $\lim_{x \to a}$ the resulting planning goal is

$$\emptyset \vdash \forall \epsilon (\epsilon > 0 \rightarrow \exists \delta (\delta > 0 \wedge \forall x ((|x - a| < \delta \wedge x \neq a) \rightarrow |(f(x) + g(x)) - (L_1 + L_2)| < \epsilon).$$

Planning methods represent inference actions and specify preconditions of the action and its effects on the planning state. The planner searches for a solution, i.e. a sequence of actions that transforms the initial state into a state with no open goals. Roughly, the planner searches backward for an instantiated method `M` whose application proves a goal g and introduces `M` into the plan. The subgoals needed for the application of `M` replace g in the planning state. The planner continues to search for methods applicable to a subgoal and terminates if no open goals are left or if no further method can be applied. Since methods are usually more abstract than the rules of the basic logic calculus, a proof of a theorem is planned at an abstract level and a plan is an outline of the proof.

Knowledge-based proof planning [9] extends proof planning by allowing for domain-specific methods and knowledge-based means to restrict the search. This includes that some domain-specific methods can call a domain-specific external reasoning system. For instance, constraint solvers for specific domains, e.g., for sets or for linear arithmetic in the real numbers $\mathbb{R}$, can be used to decide about a method's applicability.

3.1 Proof Planning in ΩMEGA

A knowledge-based proof planner forms the core of the ΩMEGA system [6]. Once a proof plan has been found in ΩMEGA, it can be refined by expanding it to a calculus-level proof, that is, a proof in ΩMEGA's basic natural deduction (ND) calculus [5]. This ND-proof can be checked for logical correctness.

In ΩMEGA the methods have a frame-like representation. An example is the `TellCS` method which plays an important role in the integration of constraint solving into proof planning:

method: `TellCS(CS)`	
premises	
conclusions	$\ominus$ L2
appl.cond	`is-constraint`(c,`CS`) AND `var-in`(c) AND `tell`(L2, `CS`)
proof schema	L2. $\Delta, \mathcal{C} \vdash c$ (solveCS;L1)

`TellCS` has the constraint solver `CS` as a parameter. The application of `TellCS` works on a goal c that is a constraints. When the conclusion of `TellCS` is matched with the current planning state, c is bound to this goal. This is indicated by the $\ominus$-annotation of the conclusion L2 which indicates that the planning goal is removed from the planning state when `TellCS` is applied. The method introduces no new subgoals since is has no $\oplus$-premises. An method is applied only if the application condition, *appl-cond*, evaluates to *true*. The application condition of `TellCS` says that the method is applicable, if the following conditions are fulfilled. Firstly, the open goal that is matched with the c in line L2 of `TellCS` has to be a constraint, i.e. a formula of the constraint language of the constraint solver that instantiates `CS`. Secondly, the goal should contain at least one problem variable whose value is restricted by c. Last but not least, the constraint goal must be consistent with the constraints accumulated by `CS` so far. The latter is checked by `tell(L2,CS)`. The constraint solver is accessed via the `tell` function which evaluates to *true* if `CS` does not find an inconsistency of the instantiated c with the constraints accumulated so far.

The *proof schema* of `TellCS` is introduced into the partial proof plan when `TellCS` is expanded. Further expansion constructs a logical derivation of the constraint c.

In order to restrict the search space, in proof planning the instantiation of existentially quantified variables by a term t is delayed as long as possible. A problem variable is introduced as a place holder instead and a constraint solver incrementally restricts the admissible values of t.

4 NEW REQUIREMENTS

For a safe and appropriate integration of constraint solving into proof planning several requirements have to be satisfied. We shall describe below the extensions that are caused by the following peculiarities in proof planning:

Synchronization with proof planning When backtracking takes place in proof planning, constraints might have to be withdrawn from the collection of constraints, i.e. the set of constraints is reduced again. Therefore, a synchronization between the proof planner and constraint solver is needed.

Since in the planning process not every variable occurs in the sequents of the initial state, the set of problem variables may be growing. In particular, proof planning methods may introduce new auxiliary variables that are not contained in the original problem. Typically, the set of constraints is incrementally growing and reaches a stable state at the end of the

[3] A sequent $(\Delta \vdash F)$ consists of a set of formulae Δ (the hypotheses) and a formula F and means that F is derivable from Δ.

planning process only. Therefore, dynamic constraint solving [12] is needed.

Logical correctness Several proof planning methods remove quantifiers from goals or assumptions. When these methods are (recursively) expanded to the ND-level, the then introduced ND-rules ∀I and ∃E [4] must obey the Eigenvariable condition of the ND-calculus in order be a correct proof steps. This condition requires that the new (Eigen)variable introduced by the rule occurs neither in any of the hypotheses of the sequent nor in the formula itself. Since proof planning uses place holders for terms that are not yet known, the Eigenvariable condition has to be extended to the occurrence in the eventual instantiations of the problem variables. Consequently, the search has to take into account the Eigenvariable constraints.

Moreover, logical correctness requires to take into consideration the hypotheses H of a constraint. In proof planning a constraint occurs in a sequent $(\Delta \vdash c)$ that consists of a set Δ of hypotheses and a constraint formula c. The hypotheses provide the *context* of a constraint and must be taken into account in the accumulation of constraints. We refer to a constraint together with its context as a *constraint sequent.* An important peculiarity of proof planning is the fact that certain problem variables might occur in different contexts. For instance, the contexts $\Delta \cup \{X = a\}$ and $\Delta \cup \{X \neq a\}$ result from introducing a case split on $(X = a \dot{\vee} X \neq a)$ into a proof plan, where Δ is the set of hypotheses in the preceding plan step. When a new constraint sequent $\Delta \cup \{X = a\} \vdash c$ is processed, the consistency of c has to be checked with respect to all constraints with a context Φ which is a subset of $\Delta \cup \{X = a\}$.

Symbolic rewriting Proof planning has to process constraints such as $E_1 \leq \epsilon/(2.0 \cdot M)$. Their terms may contain names of elements of a certain domain (e.g., 2.0) as well as variables (e.g., M and E_1), and symbolic constants (e.g., ϵ). This means, as opposed to systems that handle purely "numeric" constraints, the constraint representation and inference needs to include numeric and non-numeric ("symbolic") terms to be appropriate for proof planning. In the following, "numeric" indicates that an element of a particular domain is involved in a term, inference, or value even if the elements of the domain are not numbers.

For the constraint representation this means, e.g., that a unique lower or upper bound for a problem variable cannot always be determined, e.g., the problem variable D in a plan for LIM+ has the upper bounds δ_1 and δ_2 which cannot be replaced by a unique bound unless the function *min* is employed. The search algorithm has to be extended to find symbolic witnesses too.

5 CONSTRAINT SOLVING FOR PROOF PLANNING

No off-the-shelf constraint solver satisfies all of the above mentioned requirements and therefore we had to develop an extended constraint solver that can be safely integrated into proof planning.

[4] these are the rules $\forall\text{I}\frac{\Delta\vdash F[a/x]}{\Delta\vdash\forall x.F}$ and $\exists\text{E}\frac{\Delta\vdash\exists x.F \quad \Delta,F[a/x]\vdash G}{\Delta\vdash G}$, where a must not occur in any formula in $\Delta \cup \{F, G\}$.

5.1 Extensions of Constraint Solving

Constraint Inference The extended solver needs to integrate both, numeric and symbolic inference mechanisms. One algorithm efficiently tests a set of constraints for inconsistencies by inspecting and handling the numeric bounds of variables. We refer to this algorithm as *numeric inference.* A second algorithm, for *symbolic inference*, uses term rewrite rules to simplify the symbolic representation of constraints and constraint simplification rules to transform a set of constraints into a satisfiability equivalent which is in a *normal form.*

For the combination of numeric and symbolic inference techniques the constraint representation of a purely numeric constraint solver has to be extended because otherwise unique bounds of problem variables cannot be determined in some cases.

Context Trees Context trees consist of nodes, the *context nodes.* Each such node N_Φ consists of a set Φ of hypotheses (the context) and a set $S_\Phi = \{c \mid \Delta \vdash c \text{ is constraint and } \Delta \subseteq \Phi\}$. A context tree is a conjunctive tree representing the conjunction of all constraints stored in the nodes. Figure 1 shows

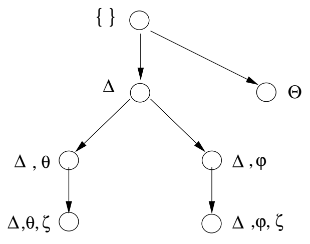

Figure 1. A Context Tree with node annotations

the structure of such a context tree. The root node is annotated with the empty context { }. A directed edge from a node N_Δ to a child N'_Φ implies $\Delta \subset \Phi$. A subtree T_Φ of a context tree consists of all nodes with a context Ψ for which $\Phi \subseteq \Psi$ holds.

The consistency of a new constraint $(\Delta \vdash c)$ has to be checked wrt. the constraint sets S_Φ with $\Delta \subset \Phi$. The constraint solver checks consistency with the sets S_Φ in the leaf nodes only because the sets of constraints grow from the root node to the leaves. In other words $\Delta \subset \Phi$ implies $S_\Delta \subset S_\Phi$. If an inconsistency occurs in at least one leaf, the constraint $(\Delta \vdash c)$ is not accepted by the constraint solver. Otherwise, c is added to all sets S_Φ in the subtree T_Δ.

When a constraint $(\Delta \vdash c)$ has to be withdrawn because of backtracking in proof planning, c is removed from all nodes in the subtree T_Δ. Empty context nodes are removed from the tree.

Search for Witnesses The constraint solver searches for a solution, i.e. witnesses for the problem variables that satisfy all constraints. Since the context tree is a conjunctive tree witnesses of the problem variables have to satisfy all constraints in the context tree if the respective context is satisfied. The search for witnesses of *shared variables*, i.e. those which occur in different contexts has to take into account all constraints of these variables. Therefore, the constraint solver creates a single search space with all constraints from the leaf nodes, i.e. the set $S = \bigcup_i S_{\Phi_i}$, at the beginning of the search process

The search algorithm uses numeric inference and symbolic inference to transform the constraints into a normal form, to detect inconsistencies as early as possible, and to compute an interval constraint $max(L) \leq X \leq min(U)$ for every problem variable X. Here, $L(U)$ is a list whose first element is the numeric lower(upper) bound $l(u)$ and the rest of $L(U)$ consists of the symbolic lower(upper) bounds. An element is dropped from a bound list as soon as it is found to be not maximal (minimal). Eventually, the maximal lower bound $max(L)$ and the minimal upper bound $min(U)$ are used to compute a witness for X. The search algorithm must not compute witnesses which contain Eigenvariables of the respective problem variable.

5.2 Implementation

The constraint solver $\mathcal{CoSIE}$ ($\mathcal{C}$onstraint $\mathcal{S}$olver for $\mathcal{I}$nequalities and $\mathcal{E}$quations over the field of real numbers) is implemented in the concurrent constraint programming language Mozart Oz [14]. $\mathcal{CoSIE}$'s constraint language consists of nonlinear arithmetic (in)equality constraints over the real numbers, i.e. constraints with one of the relations $<, \leq, =, \geq$, and $>$. Terms in formulas of this language are built from real numbers, symbolic constants and variables, and the function symbols $+, -, \cdot$, and $/$. Terms may also contain *ground terms* which contain uninterpreted (alien) function symbols. For instance, $|f'(a)|$ is a ground term containing the two uninterpreted function symbols $|.|$ and f'. $\mathcal{CoSIE}$ handles these *alien* terms by variable abstraction similar to [13]. Alien terms are temporarily replaced by constraint variables whose value cannot be restricted.

$\mathcal{CoSIE}$ builds a context tree whose nodes are *computation spaces* [16] annotated with contexts. A computation space is an abstract data type in Mozart Oz that encapsulates data, e.g., constraints and any kind of computation including constraint inference. After constraint inference has reached a fix-point, a computation space may have various states: the space is failed (i.e. the constraints are inconsistent), all propagators vanished since they are represented by the basic constraints in the constraint store, or the space contains disjunctions, i.e. constraint inference will proceed in different directions.

When a new constraint $(\Delta \vdash c)$ is sent to the solver by `TellCS`, it has to be added to certain computation spaces in the context tree. Therefore, a new computation space s_c containing c only is created and merged with all computation spaces in the leaf nodes of the subtree T_Δ. In each of these computation spaces, the symbolic inference procedure tries to simplify constraints and to detect non-trivial inconsistencies. Propagation, i.e. numeric inference, is triggered by the symbolic inference procedure as described in the next paragraph. When a fix-point is reached in numeric and symbolic inference, the resulting computation space is asked for its state. If no inconsistency is detected c is inserted into every computation space of the subtree T_Δ by merging with the space s_c.

Symbolic and Numeric Constraint Inference In $\mathcal{CoSIE}$, numeric inference is based on real-interval constraints supported by the RI-module with Mozart. Variables (for short *RI-variables*) are constrained with intervals of real numbers. As an extension, now the RI-module provides first-class propagators [15] for all relations and functions from $\mathcal{CoSIE}$'s constraint language. A first-class propagator is an abstract data type. Because of being first-class such a propagator can be inspected and discarded, e.g., by the symbolic inference procedure, and at the same time work on the constraint store in the usual way.

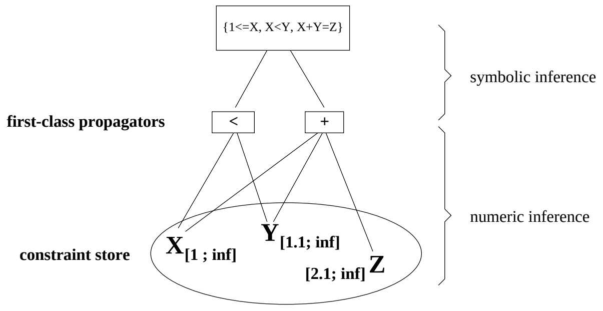

Figure 2. Combining symbolic and numeric inference

The concurrent combination of symbolic and numeric inference is based on a shared representation of constraints and by the first-class propagators. Every variable and every symbolic constant occurring in a constraint processed by $\mathcal{CoSIE}$ is connected to a corresponding RI-variable. The relation and each non-alien function of a constraint is connected to its first-class propagator in the RI-module.

Figure 2 illustrates the combination of symbolic and numeric inference. It shows $\mathcal{CoSIE}$'s connections of the constraint $1 \leq X \wedge X < Y \wedge X + Y = Z$ to the first-class propagators and the RI-variables for X, Y, and Z.

The symbolic inference procedure applies (conditional) term rewrite rules and constraint simplification rules from the theory of real numbers to (symbolic) constraints in order to transform these constraints into an equivalent normal form. Since the symbolic inference changes the term structure of constraints, it directly influences the corresponding first-class propagators. It starts or terminates first-class propagators connected to the relations and non-alien functions of the terms changed by the application of rewrite and constraint simplification rules. One of the rewrite rules used by $\mathcal{CoSIE}$ is the following:

$$(t_1 \cdot t_2)/(t_1 \cdot t_3) \quad [t_1 > 0] \quad \Rightarrow \quad t_2/t_3 \qquad (1)$$

When the symbolic inference procedure receives, for instance, the constraint $a > 0 \wedge E \leq (a \cdot \epsilon)/(a \cdot M)$, it creates new RI-variables for E, M, ϵ, and a (in case they do not exist yet) and computes new first-class propagators for the relation $\leq$ and for all occurrences of the functions $/$ and $\cdot$. The rule (1) is applied, to the term $(a \cdot \epsilon)/(a \cdot M)$, which is transformed to the normal form ϵ/M. Thus, the first-class propagators for $\cdot$ in $(a \cdot \epsilon)$ and $(a \cdot M)$ are terminated.

Search The search procedure starts with an initial computation space S and checks recursively for S and every new computation space whether it is failed or contains a solution. In each computation space, propagation restricts the domains of the variables. Additionally, the symbolic inference applies term rewrite rules and constraint simplification rules. One of the simplification rules is

$$(X \leq t_1) \wedge (X \leq t_2) \quad \Rightarrow \quad X \leq \min\{t_1, t_2\},$$

where the t_i are arithmetic terms and X is a problem variable.

6 RESULTS

ΩMEGA's proof planner together with the integrated constraint solver $\mathcal{CoSIE}$ can find proof plans for many theorems, examples, and exercises from two chapters of the introductory analysis textbook [2]. The extended constraint solver allows for a correct proof planning including planning proofs with a case split.

Example A proof that requires a case split is, e.g., the proof of the theorem ContIfDeriv. This theorem states that if a function $f\colon \mathbb{R} \to \mathbb{R}$ has a derivative $f'(a)$ at a point $a \in \mathbb{R}$, then it is continuous in a. Its formalization includes the assumption

$$\emptyset \vdash \forall \epsilon_1(\epsilon_1 > 0 \to \exists \delta_1(\delta_1 > 0 \to (\forall x_1(|x_1 - a| < \delta_1 \to ((x_1 \neq a) \to (|\tfrac{f(x_1)-f(a)}{x_1-a} - f'(a)| < \epsilon_1))))))$$

and the planning goal[5]

$$\emptyset \vdash \forall \epsilon(\epsilon > 0 \to \exists \delta(\delta > 0 \to (\forall x(|x - a| < \delta \to |f(x) - f(a)| < \epsilon))))$$

The proof planner passes the following constraint sequents to $\mathcal{CoSIE}$:

$$\begin{array}{rr}
\Delta \vdash 0 < E_1 & \Delta \vdash 0 < \delta_1 \\
\Delta \vdash D \leq \delta_1 & \Delta, (X_1 \neq a) \vdash 0 < M \\
\Delta, (X_1 \neq a) \vdash 0 < M' & \Delta, (X_1 \neq a) \vdash D \leq M \\
\Delta, (X_1 \neq a) \vdash |f'(a)| \leq M' & \Delta, (X_1 \neq a) \vdash D \leq \epsilon/(4 \cdot M') \\
\Delta, (X_1 \neq a) \vdash E_1 \leq \epsilon/(2 \cdot M) & \Delta, (X_1 = a) \vdash X_1 = x,
\end{array}$$

where Δ consists of the proof assumption and the constraints $0 < \epsilon$ and $0 < D$. The problem variables D, X_1, and E_1 correspond to δ, x_1, and ϵ_1 in the formalization of the problem. M and M' are auxiliary variables introduced by a planning method for complex estimations. The two branches of the context tree correspond to the branches of the proof plan that originate from a case split on $(X_1 = a \dot{\vee} X_1 \neq a)$.

At the end of the planning process, $\mathcal{CoSIE}$ computes the following answer constraint:

$$\begin{array}{lcl}
 & & 0 < E_1 \ \wedge \ 0 < \delta_1 \ \wedge \ D \leq \delta_1 \ \wedge \\
(X_1 \neq a & \to & 0 < M \ \wedge \ D \leq M \ \wedge \\
 & & 0 < M' \ \wedge \ |f'(a)| \leq M' \ \wedge \\
 & & E_1 \leq \epsilon/(2 \cdot M) \ \wedge \\
 & & D \leq \epsilon/(4 \cdot M'))\ ) \ \wedge \\
(X_1 = a & \to & X_1 = x\ )).
\end{array}$$

The search procedure of $\mathcal{CoSIE}$ computes the following witnesses for the problem variables of ContIfDeriv:

$$X_1 = x,\ M' = (|f'(a)| + 1),\ D = \min\{\delta_1, \tfrac{\epsilon}{4 \cdot M'}\},\ M = D,\ E_1 = \tfrac{\epsilon}{2 \cdot M}.$$

These witnesses satisfy the Eigenvariable conditions.

7 CONCLUSION AND RELATED WORK

The integration of constraint solving into proof planning causes new requirements for constraint solving that are not typically fulfilled by standard constraint solvers. Therefore, we have addressed generic extensions of a standard constraint solver. The programming language Mozart Oz is well-suited for these extensions.

Related Work `SoleX` [13] provides means for combining numerical and symbolic inference in a sequential manner. It supports the extension of the constraint language of an existing constraint solver whose soundness and completeness properties are preserved. We have adopted the `SoleX` approach to handle alien terms in the constraint language of $\mathcal{CoSIE}$. Few systems systems, for instance [19], tightly integrate constraint solving into theorem proving rather than integrating an external, stand-alone constraint solver. And, of course, none of them does proof planning.

[5] In this formalization the definitions of *limit* and *derivative* have already been expanded.

Recently the ELF-group started to integrate constraint manipulation into logical frameworks based on the results in [20]

The constraint logic programming language CAL [1] uses constraint contexts to handle guarded clauses.

Our previous work [11] mainly dealt with the interface and the actual integration of a constraint solver into proof planning. We also investigated the simplification and guidance such an integration can provide for proof planning [10]. We knew that additional features of the constraint solver would have to guarantee, e.g., the correctness of the expansion but did not elaborate on this nor implemented any extensions previously. A more detailed description of the work presented in this paper can be found in [21].

REFERENCES

[1] A. Aiba and R. Hasegawa, 'Constraint Logic Programming Systems - CAL, GDCC and Their Constraint Solvers.', in *Proc. of the Conference on 5th Generation Computer Systems.*, pp. 113–131. ICOT, (1992).

[2] R.G. Bartle and D.R. Sherbert, *Introduction to Real Analysis*, John Wiley& Sons, New York, 1982.

[3] H.-J. Bürckert, 'A resolution principle for constrained logics', *Artificial Intelligence*, **66**(2), (1994).

[4] R.R. Fikes and N.J. Nilsson, 'Strips: A new approach to the application of theorem proving to problem solving', *Logic and Computer Science*, **2**, 189–208, (1971).

[5] G. Gentzen, 'Untersuchungen über das Logische Schließen I und II', *Mathematische Zeitschrift*, **39**, (1935).

[6] The ΩMEGA Group, 'OMEGA: Towards a Mathematical Assistant', in *Proc. of CADE-14*, ed., W. McCune. Springer, (1997).

[7] A. K. Mackworth, 'Consistency in Networks of Relations', *Artificial Intelligence*, **8**, 99–118, (1977).

[8] E. Melis, 'AI-techniques in proof planning', in *European Conference on Artificial Intelligence*, pp. 494–498, (1998).

[9] E. Melis and J.H. Siekmann, 'Knowledge-based proof planning', *Artificial Intelligence*, **115**(1), 65–105, (1999).

[10] E. Melis and V. Sorge, 'Employing external reasoners in proof planning', in *Calculemus'99*, eds., A. Armando and T. Jebelean, (1999).

[11] E. Melis, J. Zimmer, and T. Müller, 'Integrating constraint solving into proof planning', in *Frontiers of Combining Systems, 3rd International Workshop, FroCoS'2000*, ed., Ch. Ringeissen, LNAI 1794, pp. 32–46. Springer, (2000).

[12] S. Mittal and B. Falkenhainer, 'Dynamic Constraint Satisfaction Problems', in *Proc. of the 10th National Conference on Artificial Intelligence, AAAI-90*, (1990).

[13] E. Monfroy and Ch. Ringeissen, 'SoleX: a domain-independent scheme for constraint solver extension', in *AISC'98*, eds., J. Calmet and J. Plaza, LNAI 1476. Springer, (1998).

[14] The Mozart Consortium, *The Mozart Programming System.* `http://www.mozart-oz.org/`.

[15] T. Müller, 'Promoting constraints to first-class status', in *Proc. of the 1st International Conference on Computational Logic*, London, (2000). to appear.

[16] C. Schulte, 'Programming constraint inference engines', in *Proc. of the 3rd International Conference on Principles and Practice of Constraint Programming*, ed., G. Smolka, LNCS 1330. Springer, (1997).

[17] G. Smolka, 'The Oz programming model', in *Current Trends in Computer Science*, ed., Jan van Leeuwen, Springer, (1995).

[18] M.K. Stickel, 'Automated deduction by theory resolution', in *Proc. of the 9th International Joint Conference on Artificial Intelligence*, (1985).

[19] F. Stolzenburg, 'Membership constraints and complexity in logic programming with sets', in *Frontiers of Combining Systems, 1st International Workshop, FroCoS'96*, eds., F. Baader and U. Schulz, (1996).

[20] R. Virga, 'Higher-order superposition for dependent types', in *Proc. of the 7th Conference on Rewriting Techniques and Applications*, ed., H. Ganzinger, LNCS 1103, New Jersey, (1996). Springer.

[21] J. Zimmer, *Constraintlösen für Beweisplanung*, Master's thesis, Fachbereich Informatik, Universität des Saarlandes, May 2000. In German.