Integrating Constraint Solving into Proof Planning

Erica Melis[1], Jürgen Zimmer[2], and Tobias Müller[3]

[1] Fachbereich Informatik, Universität des Saarlandes, D-66041 Saarbücken.
melis@ags.uni-sb.de
[2] Fachbereich Informatik, Universität des Saarlandes, D-66041 Saarbücken.
jzimmer@ags.uni-sb.de
[3] Programming Systems Lab, Postfach 15 11 50, Universität des Saarlandes, D-66041 Saarbücken.
tmueller@ps.uni-sb.de

Abstract. In proof planning mathematical objects with theory-specific properties have to be constructed. More often than not, mere unification offers little support for this task. However, the integration of constraint solvers into proof planning can sometimes help solving this problem. We present such an integration and discover certain requirements to be met in order to integrate the constraint solver's efficient activities in a way that is correct and sufficient for proof planning. We explain how the requirements can be met by an extension of the constraint solving technology and describe their implementation in the constraint solver $\mathcal{C}o\mathcal{SIE}$.

In automated theorem proving, mathematical objects satisfying theory-specific properties have to be constructed. More often than not, unification offers little support for this task and logic proofs, say of linear inequalities, can be very long and infeasible for purely logical theorem proving. This situation was a reason to develop theory reasoning approaches, e.g., in theory resolution [19], constrained resolution [6], and constraint logic programming [8] and to integrate linear arithmetic decision procedures into provers such as Nqthm [4]. Boyer and Moore, e.g., report how difficult such as integration may be.

In knowledge-based proof planning [12] 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 during the proof planning process they are represented by place holders, also called *problem variables.* In [11] we described a first hand-tailored constraint solver Lineq that incrementally restricts the possible object values. It checks for the inconsistency of constraints and thereby influences the search for a proof plan.

In Hélène Kirchner and Christophe Ringeissen, editors, *Frontiers of Combining Systems – Third International Workshop, FroCos 2000*, LNAI 1794, pages 32–46, Nancy, France, March 2000, Springer-Verlag Berlin Heidelberg 2000.

This paper presents the integration of an extended standard constraint solver into proof planning and describes several generally necessary extensions of off-the-shelf constraint solvers for their correct use in proof planning. As a result more theorems from three investigated mathematical areas (convergence of real-valued functions, convergent sequences, and continuous functions) can be proved by our proof planner.

The paper is organized as follows: First we introduce knowledge-based proof planning as it is realized in the mathematical assistant system ΩMEGA [3] and its concrete integration of constraint solving into proof planning. In section 2 we summarize the requirements that the integration into proof planning causes for constraint solving. In section 3, we discuss the essential extensions of constraint solving for proof planning. Finally, we illustrate the proof planning and particularly $\mathcal{C}o\mathcal{SIE}$'s work with a concrete proof planning example. In the following, Δ, Φ, and Ψ denote sets of formulas.

1 Integration of Constraint Solving into Proof Planning

Proof planning, introduced by A.Bundy [5], differs from traditional search-based techniques by searching for appropriate proof steps at abstract levels and by a global guidance of the proof search. Knowledge-based proof planning [12] extends this idea by allowing for domain-specific operators and heuristics, by extending the means of heuristic guidance, and by integrating domain-specific external reasoning systems.

Proof planning can be described as an application of classical AI-planning where the initial state consists of the two proof assumptions represented by sequents[1] and of the goal which is a sequent representing the theorem to be proved. 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 $a \in \mathbb{R}$ (a real number a) 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

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

After the expansion of the definiton of $\lim\limits_{x \to a}$ the resulting planning goal is

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

Proof planning searches for a sequence of operators that transfers the initial state into a state with no open planning goals. The proof plan operators represent complex inferences that correspond to mathematical proof techniques. These

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

operators are usually more abstract than the rules of the basic logic calculus. Thus, a proof of a theorem is planned at an abstract level and a plan is an outline of the proof. This plan can be recursively expanded to the calculus-level where it can be checked for correctness by a proof checker.[2]

In the following, we briefly introduce knowledge-based proof planning as it is realized in the ΩMEGA system.

1.1 Proof Planning in ΩMEGA

The operators in ΩMEGA have a frame-like representation. As a first example for planning operators, we explain `TellCS` which plays an important role in the integration of constraint solving into proof planning:

operator: `TellCS(CS)`	
premises	L1
conclusions	$\ominus$L2
appl-cond	`is-constraint`(c,CS) AND `var-in`(c) AND `tell`(L2, CS)
proof schema	L1. $\Delta_1 \quad \vdash \mathcal{C}$ () L2. $\Delta, \mathcal{C} \vdash c$ (solveCS;L1)

`TellCS` has the constraint solver `CS` as a parameter. The application of `TellCS` works on goals c that are constraints. When `TellCS` is matched with the current planning state, c is bound to this goal. This is indicated by the conclusion L2. The $\ominus$ in $\ominus$L2 indicates that the planning goal is removed from the planning state when `TellCS` is applied. The operator introduces no new subgoals because there are no $\oplus$-*premises*. An operator is applied only if the application condition, *appl-cond*, evaluates to *true*. The application condition of `TellCS` says that the operator 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)` which evaluates to true, if `CS` does not find an inconsistency of the instantiated c with the constraints accumulated so far. The constraint solver is accessed via the `tell` function.

The *proof schema* of `TellCS` contains a meta-variable $\mathcal{C}$ that is a place holder for the conjunction of all constraints accumulated (also called *answer constraint*). The instantiation of $\mathcal{C}$ is relevant for line L2 in the *proof schema* that suggests that the constraint can be logically derived from the yet unknown answer constraint.

[2] The basic calculus of the ΩMEGA system is natural deduction (ND) [17].

The control mechanism of our proof planner prefers the operator `TellCS`, if the current planning goal is an inequality or an equation.

Another planning operator is `ExistsIntro` [3] which eliminates an existential quantification in a planning goal:

operator: `ExistsIntro`	
premises	$\oplus$L1
conclusions	$\ominus$L2
appl-cond	M_x :=`new-meta-var`(x)
proof schema	L1. $\Delta \vdash \varphi[M_x/x]$ (OPEN) L2. $\Delta \vdash \exists x.\varphi$ (ExistsI;L1)

`ExistsIntro` closes an existentially quantified planning goal that matches L2 by removing the quantifier and replacing the variable x by a new problem variable M_x. The formula $\varphi[M_x/x]$ is introduced as a new subgoal which is indicated by the $\oplus$-*premise* $\oplus$L1. The function `new-meta-var` in the application condition computes a new problem variable with the type of x. The *proof schema* is introduced into the partial proof plan when the operator is expanded. `ExistsIntro` is often applied iteratively for a number of quantifiers when normalizing a goal.

Even if only one operator is applicable, there may be infinitely many branches at a choice point in proof planning. This problem occurs, for example, when existentially quantified variables have to be instantiated. In a complete proof x in $\exists x.\varphi$ has to be replaced by a term t, a *witness* for x. Since usually t is still unknown when `ExistsIntro` is applied, one solution would be to guess a witness for x and to backtrack in search, if no proof can be found with the chosen witness. This approach yields unmanageable search spaces. We have chosen the approach to introduce M_x as a place-holder for the term t and to search for the instantiation of M_x when all constraints on t are known only.

Melis [10] motivates the use of domain-specific constraint solvers to find witnesses for existentially quantified variables. The key idea is to delay the instantiations as long as possible and let the constraint solver incrementally restrict the admissible object values.

1.2 The Integration

Constraint solvers employ domain-specific data structures and algorithms. The constraint solver $\mathcal{CoSIE}$, described later, is a propagation-based real-interval solver. It is integrated as a mathematical service into the distributed architecture of the ΩMEGA system.

Fig. 1 schematically depicts the interface between the proof planner of ΩMEGA and our constraint solver. The constraint solver can be accessed directly by the

[3] `ExistsIntro` encapsulates the ND-calculus rule ExistsI which is the rule $\frac{\Delta \vdash F[t/x]}{\Delta \vdash \exists x.F}$, where t is an arbitrary term.

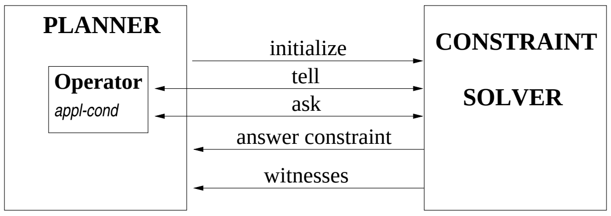

Fig. 1. Interface between constraint solving and proof planning.

proof planner and by interface functions that are called in the application conditions of certain planning operators. The proof planner's application of the operator `InitializeCS` initializes the constraint solver at the beginning of each planning process. During proof planning the main interface is provided via the planning operators `TellCS` and `AskCS`. `TellCS` sends new constraints to the solver by calling the `tell` function and `AskCS` tests entailment of constraints from the constraints collected so far by calling the `ask` function. At the end of the planning process, the proof planner directly calls the constraint solver to compute an answer constraint formula and to search for witnesses for problem variables.

A constraint solver can help to reduce the search during planning because it checks the validity of the application conditions of certain operators by checking for the inconsistency of constraints. When such an inconsistency is detected, the proof planner backtracks rather than continuing the search at that point in the search space.

2 Requirements of Constraint Solving in Proof Planning

For an appropriate integration of constraint solving into proof planning, several requirements have to be satisfied. The most relevant ones are discussed in the following.

1. Proof planning needs to process constraints containing terms, e.g., $E_1 \leq \epsilon/(2.0 * M)$. These terms may contain names of elements of a certain domain (e.g., 2.0) as well as variables (e.g., M, E_1) and symbolic constants (e.g., ϵ). So, as opposed to systems for variables constrained by purely numeric terms, the constraint representation and inference needs to include non-numeric (we say "symbolic") terms in order to be appropriate for proof planning.

In the following, we always use the notion "numeric" to indicate that a certain value or inference is related to a certain domain, although this domain does not necessarily have to contain natural, rational, or real numbers.

2. 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 operators may produce new auxiliary variables that are not contained in the original problem. Moreover, the set of constraints is incrementally growing

and typically reaches a stable state at the end of the planning process only. Therefore, dynamic constraint solving [14] is needed.

3. Since backtracking is possible in proof planning constraints that have already been added to the constraint store may be withdrawn again.

4. In proof planning a constraint occurs in a sequent $(\Delta \vdash c)$ that consists of a set Δ of hypotheses and the actual constraint formula c. The hypotheses provide the *context* of a constraint and must be taken into account while accumulating constraints, in computing the answer constraint, and in the search for instantiations of problem variables. Therefore, we refer to a sequent $\Delta \vdash c$ as a *constraint* in the rest of this paper. Importantly, certain problem variables, called *shared variables*, occur in different - possibly contradicting - contexts. For instance, the new contexts $\Delta \cup \{X = a\}$ and $\Delta \cup \{X \neq a\}$ result from introducing a case split $(X = a \dot{\vee} X \neq a)$ into a proof plan, where Δ is the set of hypotheses in the preceding plan step. When a new constraint $\Delta \cup \{X = a\} \vdash c$ is processed in the $X = a$ branch of the proof plan, its consistency has to be checked with respect to all constraints with a context Φ which is a subset of $\Delta \cup \{X = a\}$.

5. In order to yield a logically correct ND-proof when the operators are expanded, those constants that are introduced by the ND-rules $\forall$I and $\exists$E [4] have to satisfy the E*igenvariable condition*, i.e., they must not occur in other formulas beforehand. That is, they must not occur in witnesses that will be instantiated for place holders in the formulas. This condition must be satisfied by the search for witnesses of problem variables.

3 Constraint Solving for Proof Planning

Many off-the-shelf constraint solvers are designed to tackle combinatorial (optimization) problems. For them 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 [18] involves numeric constraint inference over a *constraint store* holding so-called *basic* constraints over a domain as, for example, the domain of integers, sets of integers, or 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.

[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 $\Delta \cup \{F, G\}$.

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 terminates 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 these constraint solvers are consistency check, entailment check, reflection, and search for instantiations. (In)consistency check includes the propagation of constraints combined with the actual consistency algorithm, e.g., with arc-consistency AC3 [9].

No previous solver satisfies all the above mentioned requirements and therefore we developed an extended constraint solver that can be safely integrated into proof planning. In the following, we describe the extensions of this solver and the implementation of these extensions.

3.1 Extensions of Constraint Solving

In order to meet requirement 1, a symbiosis of numeric inference techniques as well as domain specific term rewriting rules are needed. To meet the requirements 2,3, and 4, we introduce so called *context trees* which store constraints wrt. their context and enable an efficient test for subset relations between contexts. The context tree is also used to compute a logically correct answer constraint formula and to build the initial constraint store for the search for witnesses.

Constraint Inference. We employ two different kinds of constraint inference in order to detect inconsistencies as fast as possible and to symbolically solve and simplify symbolic constraints. 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.* Another 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 one which is in a unique *solved form.*

A typical constraint solver for (in)equalities in in real numbers $\mathbb{R}$ that represents constraints by numerical lower and upper bounds has to be extended because otherwise in some cases unique bounds cannot be determined. For example, if a problem variable D has two upper bounds, δ_1 and δ_2 which are symbolic constants. These bounds cannot be replaced by a unique upper bound unless a functions *min* is employed. Constraint simplification rules help to determine and to reduce the sets of upper (lower) bounds of a problem variable and to detect inconsistencies which cannot be found efficiently by purely numeric inference. For instance, the constraint $X < Y + Z \ \wedge \ Y + Z < W \ \wedge \ W < X$ is obviously inconsistent, but numeric inference cannot detect this inconsistency efficiently. This requires a constraint representation that can be handled by numeric and symbolic inference. The extension of a constraint solver needs to integrate both inference mechanisms into a single solver and benefit from the results of the respective other inference.

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. Fig. 2 shows the structure of such a context tree.

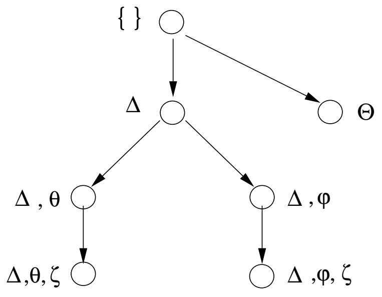

Fig. 2. A Context Tree with node annotations. Δ and Θ are sets of formulas. φ,θ, and ζ are formulas. Δ, φ stands for $\Delta \cup \{\varphi\}$.

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.

A new constraint $(\Delta \vdash c)$ must be consistent with the constraint sets S_Φ with $\Delta \subset \Phi$. The constraint solver has to check for 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. If the subtree T_Δ is the empty tree, i.e., the context Δ is new to the constraint solver, new nodes N_Δ are created and inserted into the context tree as shown in Fig. 2. This operation preserves the subset relations in the context tree.

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

The Answer Constraint. At the end of the planning process, the constraint solver uses the structure of the context tree to compute the answer constraint formula. Let $\Delta_1, \ldots, \Delta_n$ be the contexts of all nodes in the context tree and $C_1, \ldots, C_n$ be the conjunctions of all formulas which are new in $S_{\Delta_1}, \ldots, S_{\Delta_n}$ respectively, i.e., $C_i := S_{\Delta_i} - \{c \mid c \in S_{\Delta_j} \text{ with } \Delta_j \subset \Delta_i\}$. Then the answer constraint formula is $\bigwedge_i (\Delta_i \to C_i)$.

Search for Witnesses. 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 constraint solver searches for a solution for each problem variable which satisfies all constraints. In particular, the search for witnesses of *shared variables* which occur in different contexts has to take into account all constraints of these variables. Therefore, the constraint solver creates a single set with all constraints from the leaf nodes at the beginning of the search process

The search algorithm uses numeric inference and term rewriting to compute an interval constraint $max(L) \leq X \leq min(U)$ for every problem variable X, where $L(U)$ 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.

3.2 Implementation

This section describes the constraint solver $\mathcal{C}o\mathcal{SIE}$ ($\mathcal{C}$onstraint $\mathcal{S}$olver for $\mathcal{I}$nequalities and $\mathcal{E}$quations over the field of real numbers). The constraint language of $\mathcal{C}o\mathcal{SIE}$ consists of 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 $+$,$-$,$*$, $/$, min, and max. Terms may also contain *ground alien terms*, i.e. ground terms which contain function symbols unknown to $\mathcal{C}o\mathcal{SIE}$, i.e., alien. For instance, $|f'(a)|$ is a ground alien term containing the two function symbols $|.|$ and f'. $\mathcal{C}o\mathcal{SIE}$ handles these alien terms by variable abstraction, i.e., for constraint inference these terms are replaced by variables and later on instantiated again.

$\mathcal{C}o\mathcal{SIE}$ is implemented in the concurrent constraint logic programming language Mozart Oz [16]. $\mathcal{C}o\mathcal{SIE}$ builds a context tree whose nodes are *computation spaces* annotated with contexts. A computation space is a Mozart Oz data structure 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 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 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 spaces are asked for their state to detect inconsistencies. If no inconsistency is detected c is inserted into every computation space of the subtree T_Δ by merging with the space s_c.

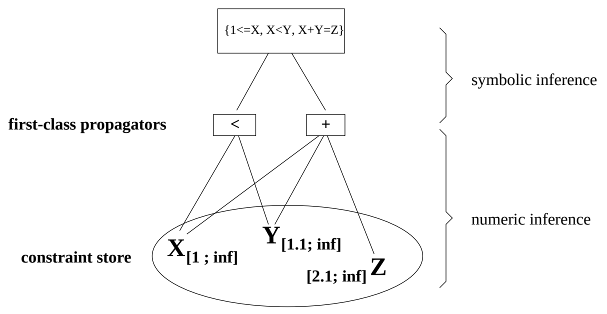

Fig. 3. Combining symbolic and numeric inference.

Symbolic and Numeric Constraint Inference. In $\mathcal{C}o\mathcal{SIE}$, numeric inference is based on the off-the-shelf Real-Interval (RI-) module coming with the Mozart Oz system. The RI-module provides *RI-variables* (constraint variables attributed with intervals of real numbers). As an extension, now the RI-module provides first-class propagators for all relations and functions from $\mathcal{C}o\mathcal{SIE}$'s constraint language. Because of being a first-class data structure these propagators can be inspected, started, and terminated, e.g., by the symbolic inference procedure and at the same time work on the constraint store in the usual way.

The symbiosis 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 $\mathcal{C}o\mathcal{SIE}$ is connected to a corresponding RI-variable. The relations and non-alien functions of a constraint are connected to the first-class propagator of those relations and functions of the RI-module.

Fig. 3 illustrates the combination of symbolic and numeric inference. It shows $\mathcal{C}o\mathcal{SIE}$'s connections of the constraint $1 \leq X \wedge X < Y \wedge X + Y = Z$ to the first-class propagators for $<$ and $+$ and to the RI-variables for X, Y, and Z in the constraint store.

The symbolic inference procedure applies (conditional) 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 $\mathcal{C}o\mathcal{SIE}$ is the following.

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

If the condition $t_1 > 0$ holds, then the rule cancels out a common factor t_1 in a fraction. 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 relations $>$ and $\leq$ 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.

The symbolic inference applies constraint simplification rules to detect inconsistencies as early as possible, e.g.,

$$(t_1 < t_2) \wedge (t_2 < t_3) \wedge (t_3 < t_1) \ \Rightarrow \ \perp \qquad (2)$$

For instance, the constraint $X < Y + Z \ \wedge \ Y + Z < W \ \wedge \ W < X$, already mentioned above, is instantly simplified to $\perp$ by the application of rule (2). With pure numeric inference it would take several minutes to detect this inconsistency.

Search. The search procedure of $\mathcal{C}o\mathcal{SIE}$ collects all constraints of the leaf nodes of the context tree in a single computation space, the *root space* of the search tree. As described below, the search may create new computation spaces. The search procedure checks recursively for each space whether it is inconsistent or contains a solution. For each computation space, propagation reduces the domains of the variables. Additionally, the symbolic inference applies term rewrite rules and constraint simplification rules to transform the constraint store into a solved form, to compute a unique symbolic smallest(greatest) upper(lower) bound for each variable, and to detect inconsistencies as early as possible. A set of constraints in solved form does not contain any redundant or trivially valid constraints, e.g., $0 < 1$. One of the simplification rules is

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

where the t_i are arithmetic terms and X is a problem variable. When propagation has reached a fix-point and no rewrite and constraint simplification rules are applicable, the space whose state is not failed is said to be stable. For a stable space with undetermined variables a distribution algorithm computes alternatives for the values of a carefully chosen variable X. The search algorithm uses these alternatives to create new disjunctive branches in the search tree, i.e., new computation spaces for every alternative for the domain of X. The new computation spaces contain exactly one of the alternatives and are submitted to recursive exploration again. The entire process is aborted as soon as a solution is found. For instance, if a variable X is constrained by $0 < X \wedge X < \epsilon$, three alternatives for X are computed, expressed by the new constraints $X = \frac{\epsilon}{2}$, $X < \frac{\epsilon}{2}$, and $X > \frac{\epsilon}{2}$.

4 Worked Example

ΩMEGA's proof planner and the integrated constraint solver $\mathcal{CoSIE}$ could find proof plans for many theorems, examples, and exercises from two chapters of the introductory analysis textbook [2]. The now extended constraint solver allows for correctly handling proofs plans that involve a case split. A case split produces alternative contexts of constraints.

A proof that requires a case split is, e.g., the proof of the theorem ContIf-Deriv. 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. In the following, we briefly describe those parts of the planning process for ContIfDeriv that are relevant for the integrated constraint solving. Let's assume a formalization of the problem that implies an initial planning state with the assumption

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

and the planning goal[5]

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

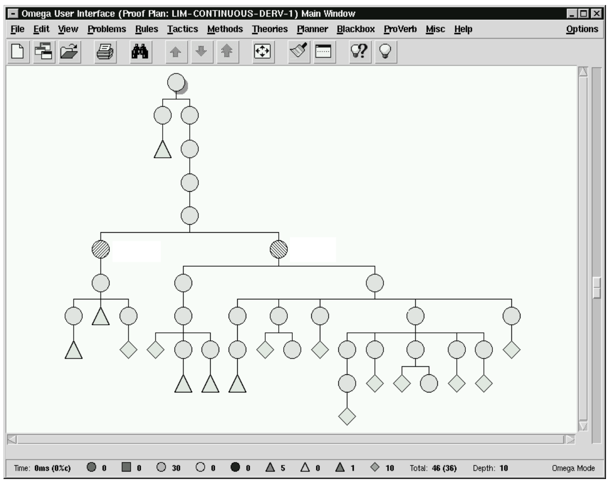
Omega User Interface (Proof Plan: LIM-CONTINUOUS-DERV-1) Main Window
File Edit View Problems Rules Tactics Methods Theories Planner Blackbox ProVerb Misc Help Options
Time: 0ms (0%c)
0
0
30
0
0
5
0
1
10
Total: 46 (36)
Depth: 10
Omega Mode

Fig. 4. The proof plan of ContIfDeriv.

The proof planner finds a proof plan for ContIfDeriv as depicted in the screen shot in Fig. 4. During proof planning, the following constraints are passed to the constraint solver $\mathcal{CoSIE}$:

[5] In this formalization the definitions of *limit* and *derivative* have already been expanded but this is not crucial for the purpose of this paper.

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

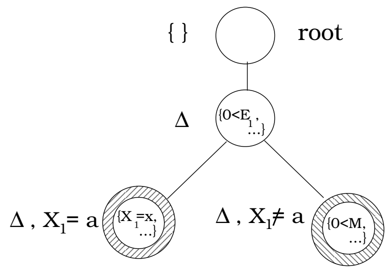

Fig. 5. The context tree for ContIfDeriv.

where Δ consists of the proof assumption (1) and the constraints $\epsilon > 0$ and $D > 0$. 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 operator.

The context tree for ContIfDeriv is shown in Fig. 5. Note that the two branches correspond to the branches of the proof plan that originate from a case split on $(X_1 = a \dot{\vee} X_1 \neq a)$. The shaded nodes correspond to the shaded plan nodes in Fig. 4.

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

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

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

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

These witnesses satisfy the Eigenvariable conditions $forbidden(E_1) = \{\delta_1\}$ and $forbidden(D) = \{x\}$.

5 Conclusion

The main theme of this paper is the integration of constraint solvers into proof planning and the nonstandard requirements caused by proof planning. Since off-the-shelf constraint solvers are typically geared towards other applications, we address generic extensions of a standard constraint solver that may also extend the potential application areas of constraint solving.

The reasons for the extensions are manifold: the constraint solver's service has to be integrated into the proof planner in a logically correct way, the constraints are usually not purely numeric, and the control of proof planning, e.g., backtracking, has to be matched on the constraint solver's side.

The programming language Oz is well-suited for the extensions reported in this paper because it provides concurrent propagation-based constraint inference encapsulated in computation spaces. The development of first-class propagators in Oz has been initiated, among others, by our need to combine numeric and symbolic constraint inference. Additionally, Oz provides the means for building new constraint systems from scratch that are as efficient as the built-in ones.

Related Work. A few theorem proving systems directly include specially designed decision procedures for constraint domains, e.g., [4], or a constraint solver [20]. All these systems tightly integrate the constraint solving into theorem proving rather than integrating an external, stand-alone constraint solver. And, of course, none of them does proof planning.

Our previous work [11] mainly dealt with interfacing and integrating the specially designed external constraint solver Lineq into proof planning by designing (Tell and Ask) operators, interface functions, and instantiation procedures. We also investigated with the merits/benefits such an integration can have for proof planning if applied appropriately and correctly [13]. We knew that additional features of the constraint solver are needed but did not elaborate on this. Now $\mathcal{C}o\mathcal{SIE}$ has been developed based on our previous experiences with symbolic constraint solving and based on the RI-module constraint solver of Mozart.

`SoleX` [15] is a general scheme for the extension of the constraint language of an existing constraint solvers preserving soundness and completeness properties. It combines symbolic and numeric inference in a sequential way. We used the `SoleX` approach to handle so-called alien terms in the constraint language of $\mathcal{C}o\mathcal{SIE}$. Constraint handling rules [7] define constraint theories and implement constraint solvers at the same time.

A context is used in the constraint logic programming language CAL [1] to handle guarded clauses. Running a CAL program results in a context tree. Therefore, context tree in CAL are conceptually different to the context trees presented in this paper.

References

1. A. Aiba and R. Hasegawa. Constraint Logic Programming System - CAL, GDCC and Their Constraint Solvers. In *Proc. of the Conference on Fifth Generation Computer Systems.*, pages 113–131. ICOT, 1992.
2. R.G. Bartle and D.R. Sherbert. *Introduction to Real Analysis.* John Wiley& Sons, New York, 1982.

3. C. Benzmueller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, A. Meier, E. Melis, W. Schaarschmidt, J. Siekmann, and V. Sorge. OMEGA: Towards a Mathematical Assistant. In W. McCune, editor, *Proc. of CADE-14*. Springer, 1997.
4. R. S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence (Logic and the Acquisition of Knowledge)*, 11, 1988.
5. A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In E. Lusk and R. Overbeek, editors, *Proc. CADE-9*, LNCS 310, Argonne, 1988. Springer.
6. H.-J. Bürckert. A Resolution Principle for Constrained Logics. *Artificial Intelligence*, 66(2), 1994.
7. T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer, 1995.
8. J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987.
9. A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.
10. E. Melis. AI-Techniques in Proof Planning. In *European Conference on Artificial Intelligence*. Kluwer Academic, 1998.
11. E. Melis. Combining Proof Planning with Constraint Solving. In *Proceedings of Calculemus and Types'98*, 1998. Electronic Proceedings http://www.win.tue.nl/math/dw/pp/calc/proceedings.html.
12. E. Melis and J.H. Siekmann. Knowledge-based Proof Planning. *Artificial Intelligence*, 115(1):65–105, 1999.
13. E. Melis and V. Sorge. Employing External Reasoners in Proof Planning. In A. Armando and T. Jebelean, editors, *Calculemus'99*, 1999.
14. S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence, AAAI-90*, pages 25–32, Boston, MA, 1990.
15. E. Monfroy and Ch. Ringeissen. SoleX: a Domain-Independent Scheme for Constraint Solver Extension. In J. Calmet and J. Plaza, editors, *Artificial Intelligence and Symbolic Computation AISC'98*, LNAI 1476. Springer, 1998.
16. The Mozart Consortium. *The Mozart Programming System*. http://www.mozart-oz.org/.
17. D. Prawitz. *Natural Deduction - A Proof Theoretical Study*. Almquist and Wiksell, Stockholm, 1965.
18. G. Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Current Trends in Computer Science*. Springer, 1995.
19. M.K. Stickel. Automated Deduction by Theory Resolution. In *Proc. of the 9th International Joint Conference on Artificial Intelligence*, 1985.
20. F. Stolzenburg. Membership Constraints and Complexity in Logic Programming with Sets. In F. Baader and U. Schulz, editors, *Frontiers in Combining Systems*. Kluwer Academic, 1996.