Structuring Deduction by Using Abstractions

Dieter Hutter
German Research Center for Artificial Intelligence (DFKI GmbH)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
hutter@dfki.de

Abstract

During the last decade a variety of industrial strength formal methods has emerged and has been applied to industrial test cases to demonstrate their adequacy and scalability. Formal techniques require a sufficient tool support especially when dealing with proof obligations. The size and the complexity of the arising problems demand for techniques to structure the deduction. In this paper we present techniques to realize a general divide-and-conquer approach in the framework of proof planning. In order to tackle different subgoals by different proof methods we propose the use of the color-calculus as an underlying constraint mechanism to resolve possible threats.

1 Introduction

The application of formal methods in an industrial setting (cf. [6]) results in an increased complexity of the specification and the correspondent verification. While various techniques have been developed to decompose specifications of large systems into modules of reasonable size (e.g. enrichment, parameterization, structuring specification), less effort has been spent to decompose the proof obligations (and their proofs) arising from the corresponding verification. In practice many proof techniques do not properly scale up with the complexity of the problem description. While some of these techniques are able to solve sophisticated problems with respect to a minimal set of axioms, the same techniques fail when a large set of redundant axioms is added to the problem description. Furthermore, dealing with inductively defined datatypes in general, proofs require induction which implies an infinite branching in each node of the search space. Since reasoning about programs typically involves deduction on well-known datatypes like integers, lists or sets we want to make use of special knowledge to guide proofs in these domains. Also in general we cannot expect that each proof obligation will be proven fully automatically which imposes the need for a user interaction allowing one to guide the prover on a strategic level.

Thus there is a need for "structuring deductions" which decomposes large proof obligations into simpler tasks and synthesizes an overall proof from the arising partial solutions. This decomposition has to enable the use of specialized methods to solve specific problems and also to ease the speculation of lemmata needed to prove the theorem.

The knowledge about how to tackle specific proof situations is encoded into a bundle of individual tactics. Like in expert systems, the accumulation of various tactics imposes an emerging functionality which is able to prove complex theorems in a goal directed way. All these tactics operate on a common representation of the actual proof state which is reflected in a proof tree annotated by additional tactical knowledge. Tactics may prune or refine this proof tree and represent the *algorithmic* part of the proof search. Proof planning [2] was proposed to combine tactics in a very flexible way by *reasoning* about their pre- and postconditions. Using these *declarative* specifications of tactics we are able to compose dynamically complex proof procedures. However, proof planning suffers from the fact that in practice tactics are highly underspecified. Hence, in order to compute sufficient planning information about the state *after* the use of a tactic, it has in fact to be executed. Thus in general we cannot elaborate a proof completely on an abstract planning level but we have to intertwine planning and plan execution.

We can overcome this tricky situation by combining two different techniques:

First, the use of abstractions allows us to plan in a hierarchical way by first searching for an abstract proof and afterwards refining this abstract solution into ground space. But the known abstractions suffer from the fact that the longer the abstract proof is, the more unlikely it is to refine this abstract solution. Either the abstraction conceals to many subtle properties which may be essential for the proof search, or the abstraction is to detailed to gain any advantages wrt. the search space. Hence in general abstractions can only be used to guide single sections or stages of the proof while the planner has to keep track of the overall progress. Then refining the abstract solutions is more likely and allows us to update the planning information of the successor states by refining the abstract proof.

Secondly, using a sequel of different abstractions to tackle a proof obligation results in the possible *threat* that a solution to a local (abstract) problem will undo some of the achievements made by previous proof steps and thus decrease the progress of the global proof which results in a need for a global measure of proof progress. We need a mechanism to pass the various kinds of information between the successive tactics in order to resolve possible threats and to gain the proposed emerging functionality.

In this paper we propose a general framework to encode proof strategic information into syntactical annotations of terms and formulas. Within this framework tactics are able to communicate their achievements to following tactics and the progress of the proof is represented by a syntactical representation of differences between the actual state and possible goal states.

2 Annotated Formulas

Annotated formulas [8] are formulas in which each occurrence of a symbol is attached by additional (strategic) information encoded into a color c which is either a color variable (like, $\alpha, \beta, \ldots$) or a color constant (like *red*, *blue*, ...). Each annotated term t corresponds to a specific term $erase(t)$ which we obtain by removing all color information from t. It is crucial for our approach that colors annotate symbol occurrences (i.e. colors are not sorts!), in particular, it is intended that different occurrences of symbols carry different colors and that symbols that carry different colors are treated differently. This observation leads to the notion of colored substitutions, a notion of substitution that takes the color information of formulae into account. A colored substitution σ is a pair $\langle \sigma^t, \sigma^c \rangle$, where the *term substitution* σ^t maps colored variables (i.e. the pair X_c of a variable X and the color c) to formulae and the *color substitution* σ^c maps color variables to colors. In order to be a legal colored substitution such a mapping σ must obey the following constraints: first, $erase(\sigma(X_c)) = erase(\sigma(X_d))$, i.e. erasing all color annotations from $\sigma(X_c)$ and $\sigma(X_d)$ results in the same (non-annotated) term, and second, if c is a color constant then all symbols occurrences of $\sigma(X_c)$ are annotated by c, i.e. free variables with constant colors stand for monochrome subformulae, whereas color variable do not constrain the substitutions. Colored equations $q = r$ are pairs of annotated terms and can be used to rewrite an annotated term t to $t[\pi \leftarrow \sigma(r)]$ if there exists a colored substitution σ with $t|\pi = \sigma(q)$.

We use colors to maintain strategic knowledge and annotate the formulas by the necessary semantic information. Suppose for example we want to implement a difference reduction technique, then we represent the similarity between two formulas by attaching same colors to corresponding symbol occurrences. E.g. the syntactical similarities between

$$g(h(a,b),k(c)) \quad \text{and} \quad g(a,f(c))$$

can be represented by the following annotated terms

$$\begin{array}{ll} & g_{red}(h_{ign}(a_{blue},b_{ign}),k_{ign}(c_{green})) \\ \text{and} & g_{red}(a_{blue},f_{ign}(c_{green})). \end{array}$$

While the "color" *ign* denotes the differences between both terms, the other colors like *red*, *blue* or *green* encode a mapping between occurrences of symbols in the first term to occurrences in the second term and vice versa. This mapping will be inherited throughout the deduction process when using the color calculus: equations used to manipulate terms are also enriched by additional color information which also implicitly encodes a mapping between occurrences of symbols on the left-hand side to those of the right-hand side. An example of such an annotated equation is

$$g_\alpha(X_\beta, k_{ign}(Y_\gamma))) = f_{ign}(g_\alpha(X_\beta, Y_\gamma))$$

where α, β, and γ are color variables. Matching the left-hand side of this equation with the first term results in a substitution $\{\alpha \leftarrow red\}, \{X_\beta \leftarrow h_{ign}(a_{blue}, b_{ign}), Y_\gamma \leftarrow k_{ign}(c_{green})\}$ which instantiates the equation to

$$\begin{array}{ll} & g_{red}(h_{ign}(a_{blue},b_{ign}),k_{ign}(c_{green})) \\ = & f_{ign}(g_{red}(h_{ign}(a_{blue},b_{ign}),k_{ign}(c_{green}))) \end{array}$$

Using this equation we are able to rewrite the term mentioned above into the annotated term

$$f_{ign}f_{ign}(g_{red}(h_{ign}(a_{blue},b_{ign}),k_{ign}(c_{green}))).$$

Thus, we have implicitly inherited the mapping of symbol occurrences between two terms along the rewrite process which gives us a simple mechanism to inherit this type of knowledge during a proof.

Formally, we introduce the notion of a skeleton function *skel*, mapping annotated formulas to some designated structure, to specify explicitly the information the deduction process has to maintain. In general the only practical restrictions to skeleton functions is that they should be compatible wrt. subterm-replacement and substitutions:

Subterm-Compatibility: Doing rewriting, the equality of left- and right-hand side of a used rewrite-rule should imply that the skeleton of the manipulated term remain unchanged, i.e. $skel(q|\pi) = skel(r)$ implies $skel(q) = skel(q[\pi \leftarrow r])$.

Substitution-Compatibility: Since we usually instantiate rewrite-rules in order to manipulate formulas, the equality of the skeletons of two terms should be preserved during their instantiation, i.e. $skel(q) = skel(r)$ implies $skel(\sigma(q)) = skel(\sigma(r))$.

If we restrict rewriting to annotated equations $q = r$ with $skel(q) = skel(r)$ then $skel(t) = skel(t[\pi \leftarrow \sigma(r)])$ holds in case $\sigma(q) = t|\pi$ and the skeleton stays invariant during the rewrite step.

For example in the difference reduction example the skeleton $skel_{DR}$ removes all those parts of a formula which are not labeled by a color variable or a member of a set C of *skeleton color constants.* E.g. let $C = \{blue, red, green\}$ then

$$\begin{aligned} & skel_{DR}(g_{red}(h_{ign}(a_{blue}, b_{ig}), k_{ign}(c_{green}))) \\ = \; & \{g_{red}(a_{blue}, c_{green})\} \\ = \; & skel_{DR}(f_{ign}(g_{red}(h_{ign}(a_{blue}, b_{ign}), c_{green}))) \end{aligned}$$

The impact of the mentioned properties are twofold. First, we can use annotations to attach planning information at the formula which is then automatically inherited during the deduction process. Second, using the notion of a skeleton we can protect parts of a formula from modification and thus restrict the search space under consideration if we declare them as a part of the skeleton.

3 Representation of Proof Progress

Annotated terms are a framework to maintain various planning information during an actual proof search. They provide a simple mechanism to propagate knowledge arising within the proof search.

As a simple example consider the problem of inheriting a *focus* to some substructure of the formula during the proof; i.e. we like to mark two terms of a formula, manipulate the formula and observe how the terms are changed during the deduction process. While the maintenance of explicit selector functions is a painful task during the rewrite process, the color calculus allows us simply to color the denoted terms and to let the calculus automatically maintain the denoted terms. These foci are a basic prerequisite to divide the proof search process into different tasks which operate on different parts of the formula.

In terms of proving an equation $q = r$ we can use the color calculus to make syntactical differences explicit. Then, a possible proof heuristic is to minimize the syntactical differences of q and r by using given axioms until we finally reach a goal situation $q' = q'$. The similarities of q and r can be represented by connecting corresponding (but not necessarily all) occurrences of the same function or variable symbol between q and r. As said before we may encode these connections with the help of annotated terms by attaching unique skeleton colors to corresponding symbol occurrences. All symbol occurrences which are not related to any other symbol are labeled with a non-skeleton color like *ign*. E.g. with $C = \{blue, red\}$ in

$$f_{ign}(g_{blue}(a_{red})) = g_{blue}(h_{ign}(a_{red}))$$

the two subterms $g(h(a))$ and $g(a)$, and both occurrences of a are related to each other.

While skeleton colors denote the common parts of both sides of the equation to be proven, other color constants represent their differences. Thus, instead of describing differences by identifying minimal subterms which contain them (as in the RUE-approach), the notion of annotated terms allows one to represent the differences explicitly and therefore, seems to be a more adequate representation for implementing difference reduction techniques. Also, the explicit representation of differences is an important requirement in computing a measure of how far two terms or formulas differ which is a central aid in guiding (parts of) the proof search.

Suppose, we want to prove a goal equation $q = r$ with the help of a set of axioms yielding a chain of intermediate equations $q = r, q_1 = r_1, \ldots, q_n = r_n$. We start with no explicit known similarities between both sides and obtain an empty skeleton on both sides. In order to manipulate an equation $q_i = r_i$ we allow two different kinds of modifications:

First, annotated axioms (i.e. rewrite rules) are used to rewrite an equation $q_i = r_i$ to a new goal $q_{i+1} = r_{i+1}$. Using annotations, such a step will preserve the skeleton of the equation, i.e. $skel_{DR}(q_i = r_i) = skel_{DR}(q_{i+1} = r_{i+1})$. This manipulation step corresponds to a usual rewrite step $erase(q_i = r_i) \rightarrow erase(q_{i+1} = r_{i+1})$ in the underlying first-order calculus.

Second, there are skeleton changing steps which do not change the underlying equation, i.e. $erase(q_i = r_i) = erase(q_{i+1} = r_{i+1})$, but which redye arbitrarily the equation such that still $skel_{DR}(q_{i+1}) = skel_{DR}(r_{i+1})$ holds. This step has no effect in the underlying first-order calculus since the non-annotated equations remain unchanged but is has major impacts on the strategic level. Adding for instance new symbol occurrences to the skeleton imposes additional decompositions of the problem since the prover will adapt the related subterms as far as possible in a first step before tackling the overall problem. It structures the proof search in such a way that in a first step the connected subterms have to be manipulated in such a way that they share a (as large as possible) common syntactic expression (or common skeleton, respectively).

In a typical deduction, rewrite steps and skeleton changing steps alternate with each other until we finally end up with a trivial equation $q' = q'$ in which both sides coincide and $skel_{DR}(q') = \{q'\}$. We achieve this final situation if we succeed to remove all occurring non-skeleton parts either by applying rewrite-rules or by increasing the common skeleton (i.e. insertion of common parts into the skeleton). We define a deduction step $q_i = r_i \to q_{i+1} = r_{i+1}$ to be *skeleton embedding* if for each $q' = r' \in skel_{DR}(q_i = r_i)$ there is some $q'' = r'' \in skel_{DR}(q_{i+1} = r_{i+1})$ such that $q' = r'$ is homomorphic embedded [9] in $q'' = r''$. Since rewrite steps do not alter the skeleton they are trivially skeleton embedding.

This approach gives us a general framework to implement proof procedures which are based on syntactical differences. A simple difference reduction tactic can be implemented by a heuristic which allows only skeleton embedding rewrite steps. Thus, the common parts of the two terms will monotonously increase during the rewrite process. Each deduction in the first-order calculus can be trivially simulated by a skeleton-embedding deduction since we may perform the complete deduction using an empty skeleton and add a final step in which we enlarge the skeleton to cover the total equation, i.e. we solve the problem without any problem decomposition. But from a strategic point of view we only gain advantages if we impose a non-empty skeleton as soon as possible in order to restrict the search space. Since the skeleton remains invariant, admissible deduction steps will only change the non-skeleton parts of the annotated term. Thus, declaring parts of a term to be parts of the skeleton prevents them from being modified and also structures the problem as it imposes relations between subterms of q and r.

4 Abstractions

We propose the use of abstractions in order to guide the proof search. Following [4] an abstraction is a pair of formal systems $\langle \Sigma_1, \Sigma_2 \rangle$ with languages Λ_1 and Λ_2 and a total function $f_\Lambda : \Lambda_1 \to \Lambda_2$. In our setting Λ_1 is the underlying annotated first-order calculus we use for equality reasoning while Λ_2 depends on the abstraction we choose to tackle the actual problem (see below).

$$\begin{array}{ccccccc}
\Phi(a,b) & \to_{\Lambda_2} & \Phi(a_1,b_1) & \ldots & \to_{\Lambda_2} & \Phi(a_n,b_n) \\
\uparrow_{f_\Lambda} & & \downarrow & \ldots & & \downarrow \\
q = r & \to^*_{\Lambda_1} & q_1 = r_1 & \ldots & \to^*_{\Lambda_1} & q_n = r_n
\end{array}$$

Figure 1: Abstracted proof search

Using abstractions we transform the given problem of equalizing two annotated terms q and r into a problem of manipulating two objects a and b in a formal system Λ_2 in order to obtain a so-called *solved form* $\Phi(a_n, b_n)$. Each step in the abstract space has to correspond to a sequel of steps in the ground space (Λ_1).

Once we find a solution in the abstract space, we have to transfer it to the given (first-order) calculus which imposes a set of different subtasks to be tackled. For each intermediate result $\Phi(a_{i+1}, b_{i+1})$ of the deduction in the abstract space we have to find an appropriate equation $q_{i+1} = r_{i+1}$ such that $q_i = r_i \to^*_{\Lambda_1} q_{i+1} = r_{i+1}$ holds.

Thus, abstractions are used to speculate intermediate goals during the proof search. Typically the refinement of abstract steps results in tasks to enable the use of specific bridge lemmata which are tackled by the use of difference unification (difference matching, respectively) and appropriate rippling techniques.

In the following we sketch some abstractions which we use in our approach. In order to ease notation we will use a kind of abstraction on the meta-level and omit the annotation of a symbol in case the annotated color is a skeleton color or a color variable while in case of a non-skeleton color we will shade the symbol itself. Thus, we will denote the annotated term $g_{red}(h_{ign}(a_{blue}, b_{ign}), k_{ign}(c_{green}))$ by the shaded term $g(h((a), b), k(c))$.

4.1 Wave-front Positions

Rippling [2, 8] was introduced as a basic mechanism to maintain the similarities of terms during the proof and to focus the interest of the prover to specific parts of the theorem. Rippling also denotes a kind of abstraction which allows one to formulate abstract goals. The idea is to abstract from the concrete shape of the non-skeleton parts (grey parts) and to specify the positions in which these wave-fronts may occur in the abstracted goal state.

Suppose, we want to equalize the two (annotated) terms $(a + b) \times (a - b)$ and $(a \times a) - (b \times b)$, then we abstract from the shape of the wave-fronts and obtain the following abstractions: $(\ldots a \ldots) \times (\ldots a \ldots)$ and $\ldots (a \times a) \ldots$ In their abstractions both terms disagree in the positions of the wave-fronts wrt. the skeleton. While in the first term the wave-fronts occur inside both arguments of $\times$, in the second term the only wave-front occurs at top-level. Equalizing both terms requires also to equalize the positions in which wave-fronts occur. Thus, an abstract solution will fix the positions of the wave-fronts while the refinement of the abstract solution will ripple the wave-fronts from their current positions to the desired target positions. Consider our example: if we commit ourselves to the top-level as target position for wave-fronts we have to ripple out all wave-fronts in the first term while the second term already meets the constraints of the abstract solution.

4.2 Paths to Common Parts of the Skeleton

The next abstraction [1] suggests to equalize the paths in both terms from the top-level to a specific part of the skeleton. In case of success the symbols on this path will be added to the skeleton in the next step. This heuristic produces a proof sketch which suggests the manipulation of "paths" to specific occurrences of terms. Expanding this proof sketch to a first order proof will create a chain of bridge lemmas to be applied on the goal.

Consider the following example of proving

$$\forall a, b\; f(a, g(a), b) = b \tag{1}$$

with the help of the following axioms:

$$\begin{aligned} f(f(V, W, X), Y, f(V, W, Z)) &= \\ f(V, W, f(X, Y, Z)) & \qquad (2) \\ f(W, Z, Z) &= Z \qquad (3) \\ f(W, W, Z) &= W \qquad (4) \\ f(Z, W, Z) &= Z \qquad (5) \\ f(g(X), X, Y) &= Y \qquad (6) \end{aligned}$$

Since only b occurs on both sides of (1), we consider all the *paths* from top-level to the occurrences of b. A path to a term t is the list of function symbols occurring from the root to a specific occurrence of t. Additionally we attach to each function symbol the argument position i into which we have to descend in order to visit t. Thus $[f_3]$ specifies the path to b on the left-hand side of (1) while $[]$ denotes the empty path to b on the right-hand side.

In order to search for an abstract proof of (1) we compute the abstraction of the equations (2) - (6) which results for instance in the following abstract equations:

$$\begin{aligned} \{[f_1], [f_2]\} &= \{[]\} \qquad (7) \\ \{[f_1, f_1], [f_3, f_1]\} &= \{[f_1]\} \qquad (8) \\ \{[f_1, f_2], [f_3, f_2]\} &= \{[f_2]\} \qquad (9) \end{aligned}$$

While (7) is an abstraction of (4), (8) and (9) are both abstractions of (2) depending on the selection of the common subterm (V and W respectively).

Thus, the problem of equating terms is reduced to the problem of equalizing the strings of indexed symbols with the help of these abstracted equations on strings of indexed symbols. In the abstract space we obtain the following proof of our abstracted problem $\{[f_3]\} = \{[]\}$

$$\begin{aligned} \{[f_3]\} &= \{[]\} \\ \{[f_3, f_1], [f_3, f_2]\} &= \{[]\} \\ \{[f_1, f_3], [f_2]\} &= \{[] \\ \{[f_1, f_3], [f_2], [f_3, f_3]\} &= \{[]\} \\ \{[f_1, f_3], [f_2], [f_3, f_3]\} &= \{[f_1, f_3], [f_2], [f_3]\} \\ \{[f_1, f_3], [f_2], [f_3, f_3]\} &= \{[f_1, f_3], [f_2], [f_3, f_3]\} \end{aligned}$$

In the next step we have to refine this abstract proof into ground space. For each abstract proof step using some abstract equation E we use the original equation, which caused the generation of E, as a bridge lemma. Thus, we have to manipulate the theorem in order to enable the use of this lemma and apply it afterwards which gives us a layout of the later proof.

In this example the refinement of the abstract proof is quite simple since each of the steps in the abstract space correspond almost to one manipulation step in the ground space. Thus, transferring the abstract proof to ground space we end up with the following proof obligation related to the last (tautologous) abstract formula:

$$\begin{aligned} & f(f(a, g(a), b), b, f(a, a, X)) \qquad (10) \\ = \; & f(U, b, f(g(Z), Z, b)) \qquad (11) \end{aligned}$$

X, U, Z denote (meta-)variables which we instantiate by comparing both terms. Applying the resulting partial unifier yields the following instantiated problem:

$$f(f(a, g(a), b), b, f(a, a, X)) \quad (12)$$
$$= \quad f(f(a, g(a), b), b, f(g(a), a, b)) \quad (13)$$

It still remains a clash inside the nested occurrences of f. While on the left-hand side the term a occurs at the position $(3, 1)$, $g(a)$ occurs at the same position on the right-hand side. In order to eliminate these occurrences appropriate bridge lemmata have to be selected, the use of which will remove the denoted differences. Axioms like (3), (4), or (5) have the property that they will remove syntactical structure when applied from left to right. We make this property explicit by highlighting the common (preserved) structure. In case of (4) we obtain the following annotated equation:

$$f(W, W, Z) \quad = \quad W \quad (14)$$

To get rid of the differences in (13) we have to apply such an equation in such a way that the shaded parts of the equation match with the differences $g(a)$ and a respectively. We use difference matching to analyze possible situations in which the designated equation can be applied and annotate (13) with colors in order to store this information. This process results in a unique solution which suggests the application of (4) on both sides. The symbols related to the left-hand side of (4) are highlighted (while the rest is shaded).

$$f(f(a, g(a), b), b, f(a, a, X)) \quad (15)$$
$$= \quad f(f(a, g(a), b), b, f(g(a), a, b)) \quad (16)$$

The colors specify the designated parts of the formula we like to modify by (14). The aim is to "glue" the white parts together in order to enable the use of (14). Since we have already equalized the paths to the occurrences of b on both sides, the rewrite process is restricted by additional constrains. We have to rewrite both sides in such a way that the shaded parts always coincide on both sides. This can be done in the following way:

$$f(f(a, g(a), b), b, f(a, a, b)) \quad (17)$$
$$= \quad f(f(a, g(a), b), b, f(g(a), a, b)) \quad (18)$$
$$f(f(f(a, g(a), b), b, a), a, \ldots) \quad (19)$$
$$= \quad f(f(f(a, g(a), b), b, g(a)), a, \ldots) \quad (20)$$
$$f(f(f(a, g(a), b), b, a), a, b) \quad (21)$$
$$= \quad f(f(f(a, g(a), b), b, g(a)), a, b) \quad (22)$$
$$f(f(f(a, g(a), b), b, f(a, Y, a)), a, b) \quad (23)$$
$$= \quad f(f(f(a, g(a), b), b, f(a, Y, a)), a, b) \quad (24)$$
$$f(f(a, g(a), f(b, b, a)), a, b) \quad (25)$$
$$= \quad f(f(a, g(a), f(b, b, g(a))), a, b) \quad (26)$$

Using the designated bridge lemma on both sides yields the final tautology:

$$f(f(a, g(a), b), a, b) \quad = \quad f(f(a, g(a), b), a, b) \quad (27)$$

The example above illustrated the use of annotated formulas to maintain information necessary to implement a prover guided by syntactical differences.Due to the recursive structure of the used methods, we aim at a stack of different goals in each situation. E.g. suppose we want to prove an equation $q = r$. In order to equalize q and r a method may suggest to apply a bridge lemma $s = t$ to q and thus, another task arises to manipulate q in order to enable the use of $s = t$. Any manipulation of q has to obey the constrains of *all* nested methods. Hence, it has to preserve the common skeleton of q and r as well as it has also to preserve the other skeleton between q and s. This is done by using different annotations of the same equations in order to rewrite the different annotations of the intermediate goal. Thus, the more we speculate subgoals to explore the search space the more we collect constraints given by the hierarchy of skeletons which have to be preserved during possible rewrite steps.

5 Conclusion

The explicit notion of differences and the corresponding mechanism to maintain them allows us to divide problems more flexible into subproblems than other existing approaches (which are mainly restricted to a top-down approach). We use abstractions to cope with situations in which no sufficient control knowledge is available in order to plan a deduction in ground space.

We have implemented and successfully tested our method to both, examples arising from industrial case studies in verification and standard academic examples, like for instance in lattice ordered groups given in the TPTP problem library. In the latter category for example, SAM's lemma [3] is fully automatically proven by the system in less than a second. Also problems from the ternary Boolean algebra [10] can be solved by our approach using a path-abstraction [1]. More details about the abstractions used for our approach can be found in [1, 5, 7].

References

[1] S.Autexier, D. Hutter: *Parameterized Abstractions used for Proof Planning* Research Report RR 97-04, DFKI GmbH, Saarbrücken, 1997

[2] Bundy, A., Stevens, A., V. Harmelen, F., Ireland, A., and Smaill, A. Rippling: a heuristic for guiding inductive proofs. Journal of Artificial Intelligence, No. 62, 1993

[3] J. Guard, and F. Oglesby, and J. Bennett, and L. Settle. *Semi-automated mathematics.* Journal of the ACM, pp. 49-62, Vol. 16, 1969

[4] F. Giunchiglia, T. Walsh. A Theory of Abstraction. *Journal of Artificial Intelligence*, 56(2-3), pp. 323-390, 1992.

[5] D. Hutter *Using Rippling for Equational Reasoning* 20th KI-96, Ed. S. Hölldobler, Springer, LNAI 1137, Dresden, Germany, 1996

[6] D. Hutter, B. Langenstein, C. Sengler, J. Siekmann, W. Stephan, A. Wolpers: *Verification Support Environment (VSE)* Journal of High Integrity Systems, Oxford University Press, Vol. 1, No. 6, pp. 523-530, 1996

[7] D. Hutter *Hierarchical Proof Planning Using Abstractions* 10th FLAIRS'97, Daytona Beach, Florida, 1997

[8] D. Hutter: *Colouring Terms to control equational reasoning* Journal of Automated Reasoning, Kluwer-Publishers, Vol. 18, pp. 399-442, 1997

[9] J.B. Kruskal: *Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture.* Trans. Amer. Math. Soc. 95, pp. 210-255, May 1960

[10] E.L. Lusk, W.M. McCune. *Uniform Strategies: The CADE-11 Theorem Proving Contest.* Journal of Automated Reasoning, Vol. 11, pp. 317-331, 1993