Extending automatic theorem proving by planning

Inger Sonntag, Jörg Denzinger
Fachbereich Informatik
Universität Kaiserslautern
E-mail: {sonntag, denzinge}@informatik.uni-kl.de

Abstract

A general concept for combining planning with automatic theorem proving is introduced. From this a system architecture based on the notion of planning trees, methods and sensors is developed. It is illustrated by examples taken from the domain of sorting algorithms.

Contents

1 Introduction

In the developing field of automatic theorem provers we have reached a turning-point. Domain-specific systems have been developed to such an extend, that they can now efficiently prove specific problems, but only in very specific domains. However, we believe that, to further improve automatic provers, a more general approach has to be taken - an approach which primarily concerns itself with the way a proof is found. Apart from domain-specific knowledge, general techniques of human problem solving could considerably advance the process of proving.

Up to now the actions to find a proof, for a given problem, represent a random search in a widely spread search space that does not follow any particular problem-oriented plan. The search can, if at all, be influenced only by the use of restricted heuristics which mostly need to be evoked and tuned by the user. Generally speaking such an approach lacks flexibility. Consider, for the moment, the most accomplished problem solver, the human being. One of its strongest features is its ability to adapt to changing situations by adjusting the course of its action, even if this is contrary to its original plan. It possesses the ability to change, revise, generalize and specify its plan according to the current circumstances. We believe that this flexibility, which we refer to as proof planning, has to be embedded in the future development of automated reasoning. In fact the need to adopt human reasoning techniques in automatic provers has already been suggested by Newell [New81] and more recently this notion has been adopted by Bundy in the development of the inductive proof checker Oyster/Clam [Bun88].

With this in mind we have commenced an investigation into the feasibility of proof plans by analysing proof processes in connection with inductive theorem proving in equational theories based on Rewrite-methods. In order to be able to judge the creation of plans and their validity we have concentrated on a particular class of examples: the verification of sorting algorithms. This class offers not only complex proof problems but is also of general interest as sorting algorithms are common in most areas of computer science; for example as system calls in operating systems. As a starting point a publication of Gramlich [Gra90], a case study of verifying sorting algorithms with the completion-based prover UNICOM, has come in handy.

From our study of examples a system architecture has resulted which we believe to be especially suited for proof planning. After a description of the underlying basic concept concerning verification and planning we intend to present this architecture. It is then followed by an illustration of its proof process in connection with the verification of sorting algorithms.

2 System architecture

2.1 Underlying ideas

Let us now take a closer look at the general ideas underlying our proposal for an automatic theorem prover that possesses a planning component. We believe that a proof process should consist of two interleaving activities: verification and planning. The **verifier** refers to the component of a proof process in which the actual proof steps are performed. As such it represents an automatic theorem prover in the traditional sense. If a verifier automatically solves a given problem, no planning will be needed to successfully conclude a proof. However, the existence of such a powerful prover is doubtful. Keeping this in mind, we characterize **planning** as a means by which a problem is segmented into sub-problems such that the verification of the latter is more plausible and induces the correctness of the original problem description [1]. A **plan** for a specific problem can then be seen as a structure that reflects the proposed segmentations. The sub-problems themselves have to be proved to guarantee the correctness of their origin. This is again tested by verification or further segmentation, that is to say by planning the proof of the sub-problems. The resulting plan for the original problem has to record all the deduced sub-problems. Planning the proof of a sub-problem thus extends the preceding plan. This is why it is also called **refinement of a plan**. Note that this concept of planning requires the verifying process to interleave with the planning process.

Up to now planning has been described as a series of plan refinements. However, the set of sub-problems to which a problem can be reduced is, in general, not obvious. On the contrary, there usually exists quite a wide variety of correct segmentations of a problem. In addition, it can not be guaranteed that a proof for each sub-problem will be found. This explains the necessity to provide means by which the success of a segmentation, a refinement of a plan, can be evaluated. These means do not only have to be employed for the immediate decision on an refinement. It should be clear that the "predicted success" of a plan is not certain. When, at a later stage of the proof process, the success of a plan is found to be below expectations it will be necessary to backtrack to a preceding (in this new context, more promising) plan which is then alternatively refined. This switching between plausible plans is called **replanning**.

For the above concept of planning the choice of an adequate data structure for plans is of importance. We have found **planning trees** to be an appropriate representation of plans that naturally reflects the dependencies of created problems as well as the sequence of refinements. In addition it enables the stages of replanning to be easily followed. The chosen structure for plans implies the construction or expansion of a planning tree to be one planning step. We now conclude this section by presenting a somewhat more technical description of planning trees, which will be further developed in section 2.2:

[1] Note that our description assumes the correctness of a stated theorem. Appropriate mechanisms to manage incorrect statements are included in our concept but are neglected in this general presentation.

Each node of a planning tree depicts a problem to be solved, that is to say a theorem of which the correctness is of interest. Leaves are the only nodes to be further processed. Leaves, of which the problem can be verified are called **verified nodes**. In addition, nodes are referred to as verified nodes, of which every son is a verified node and of which the problem can be verified using the information provided by the sons. A leaf is expanded by attaching additional leaves depicting sub-problems of its respective problem. A planning tree that only consists of verified nodes is referred to as a **complete planning tree**, which implies that all its problems have been solved (all theorems listed in the nodes are proven correct).

2.2 Overview of an architecture

In this section an architecture of a system for automatic theorem proving featuring planning is developed. We start off by elaborating the keywords presented in the previous section. This will lead to a description of the key components of the system. Figure 1 visualizes the major components and their interplay.

Of the two activities that we have divided a proof process into, planning will be discussed in more detail than verification. There are two reasons for this. Firstly, verification in regard to automatic theorem proving has already been (thoroughly) discussed; see for example [BM79, BM88, HHRS86, DM89]. Secondly, we consider the strength of our approach to lie in its flexible and interactive planning component. It dominates the verification component. That is to say that the verifier is activated and deactivated according to decisions made by planning. It then follows that the **verifier** has to meet certain requirements by featuring an interface tailored to the planning component's need. This interface has to allow

- time duration control
  The duration of the verifier's activity can be bound by a time factor.

- control of the number of steps
  The number of steps the verifier performs can be constricted.

- readily accessible information on the state
  The acceptance or rejection of current theorems has to be made visible. In addition, further information on the state of a current proof has to be available. This is necessary to enable an evaluation of the current verification process which, in turn, provides a basis for further planning decisions and for judging current ones.

- acceptance of additional theorems
  Additional theorems can be made available to the verifier. They are to be considered as lemmata.

In later examples we will presume the existence of a verifier with the above features

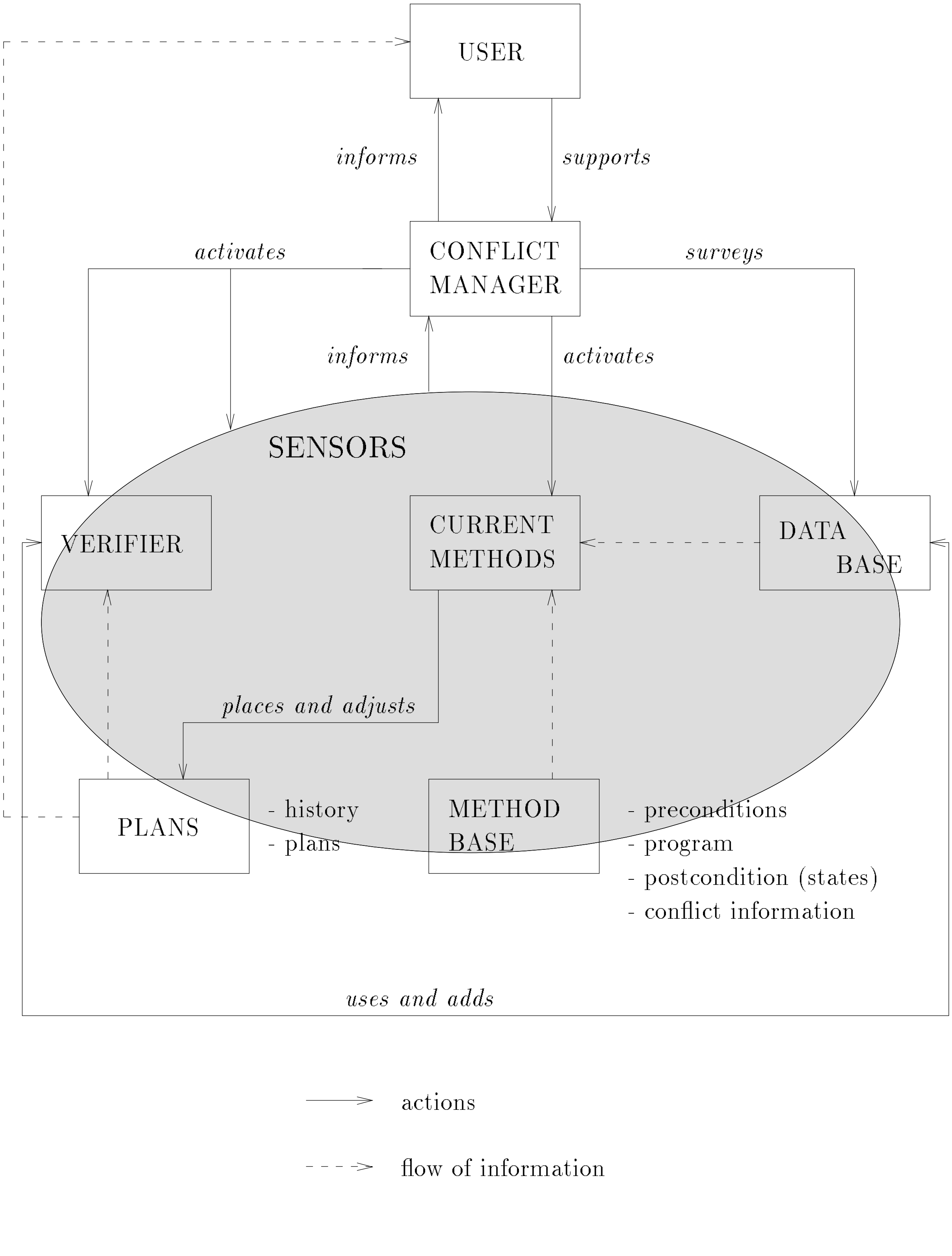


Figure 1: An automatic theorem prover featuring planning

that, in addition, presents an inductive theorem prover using rewrite methods according to Bachmair's inference rules [Bac88].

The need for the last feature of the enumeration is rooted in the understanding of a plan as a reflection of a possible problem segmentation. In the case of theorem proving segmentation implies the construction of additional theorems. Their infusion into the verification process hopefully leads to a verification of the original theorem itself. For this purpose these additional theorems have to be introduced into the verification process such that they are applied with a higher priority than theorems and lemmata from the data base. As the distinguished theorems are rooted in a plan they have to be passed to the verifier by a planning component. This component is referred to by **plan component**. It manages the plans generated in a proof process.

As the plan component is in charge of plans it has to provide an appropriate data structure, the planning trees, including respective administrative mechanisms. A planning tree has to describe a plan as well as its construction so that it can be of use to planning. A plan itself can easily be deduced from the structure and the contents of the nodes of its corresponding tree. This alone, however, does not allow reasoning on the state of a plan and thus evaluation of its quality is not possible. With this in mind the information provided by a node of a planning tree has to be extended. The resulting node structure is illustrated in figure 2. It has been mentioned that each node

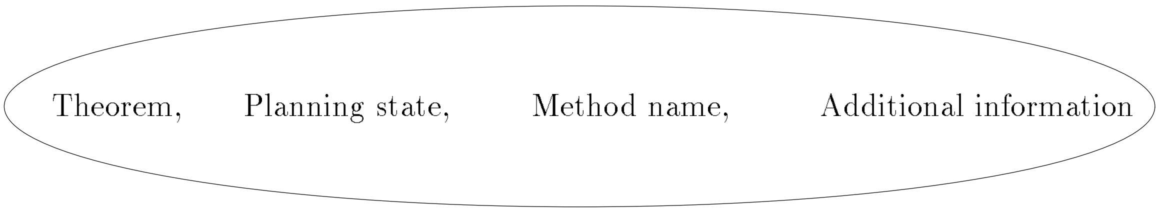


Figure 2: Node structure of planning trees

features the problem to be investigated - the theorem of which the correctness is of question. In addition, each node has to reflect its theorem's planning state depending on the node's sons. There exist four possible planning states for each node:

- verified (v)
  The node's theorem has been verified under the presumption that the theorems depicted in the sons are verifiable.

- rejected (r)
  The node's theorem has been proven incorrect.

- unsuccessful (u)
  The node describes the root of a subtree which represents a failed (sub-)plan.

- non-verified (n)
  No evaluation concerning the correctness of the node's theorem or its planning

worth has yet been possible. Note that the planning state of every newly created node is instantiated by non-verified.

It is clear that planning will mainly concentrate on non-verified nodes. A further demand in regard to the node structure lies in the support of sensible refinement and replanning decisions. In the case of refinement a tree is expanded by attaching new nodes, i. e. leaves, to an existing node. It is important to keep track of the planning step that has led to the expansion. This is to say the expanded node should feature an identification of the employed planning step leading to its creation. This information, referred to by method name, is relevant to avoid planning loops and for efficient replanning. By planning loops we refer to replanning that repeatedly leads to problem segmentations that have previously been omitted as unsuccessful. In addition, this identification enables planning to draw conclusions from unsuccessful planning steps. It has long been accepted that problem solving processes can be enhanced by analyzing their failures. The fourth information, additional information, stored in the node structure includes data provided by one of the system's features, which will later be referred to by postcondition of a method.

Concluding, the plan component does not only manage plans but also their history. This way the plan component can be seen as a foundation for a learning component that constructs plans for new, different problems by analyzing the old ones. The learning component is part of future work and has not yet been included in the suggested system's layout. Furthermore, note that learning as well as a user's insight of the proof process requires well-organized presentations of the information contained in the plan component. It is, for example, possible to keep refinement and replanning information either in separate planning trees or in one highly complex one, as long as the history is recognizable. In later examples we have opted for the second alternative.

Up to now the process, which leads to the construction of plans (that are held in the plan component and tested by the verifier) has not been considered. This process consists of a descriptive and a control part. While the descriptive part serves to express differing possibilities of plan generations, the control unit analyses a proof's state and determines the course of action. The latter chooses between the planning steps offered by the descriptive unit and controls the activation of the verifier. As there exists, in general, a wide range of possible activities the control unit's tasks is to make decisions based on conflicting information. This is why we call it **conflict manager**. It can be seen as the heart of the system's layout where all actions are decided upon. This implies that the quality of a realized system following this concept highly depends on the quality of its conflict manager. In the later stages of development the conflict manager is presumed to solve conflicting situations without much intervention of the user, that is to say more or less automatically. However, the user might be interested in strongly influencing the planning process by programming it step by step. This resembles the programming of proofs themselves and enables to form the planning process in a more deterministic way. The programming of the conflict manager will be especially useful during the development of a well-suited and flexible control unit.

Keeping the conflict manager's task in mind it is important to feed its decision procedures with appropriate information on the system's state. To such an end the concept of **sensors** is introduced. Sensors can be seen as spys that keep track of the changing system states, of which the information - processed to the need of the conflict manager - can thus be accessed at all times. From the sensors' task follows that they have to be adapted to all the concerned components: verifier, plan component, descriptive unit, data base. It is clear that the use of sensors for evaluating the stage of a verifying process highly depends on the kind of information the verifier allows to be visible to the outside. An example for the employment of sensors in regard to a data base lies in pattern recognition of given defined equational systems. In addition to information-gaining aspects, sensors present a means by which the later introduced methods as well as sensors themselves can be classified. It allows a control of the number of currently activated methods and sensors. This is of considerable importance to the feasibility of a system following this lay-out as a vast number of sensors and methods can be expected.

In addition to sensors the conflict manager needs to be supported by the user of the proof system. The user's task lies in monitoring the planning stages in form of processed information provided by the plan component. Furthermore, the user can supersede or take over decisions of the conflict manager. In case the conflict manager can not reach a decision it also relies on direction from him.

A further possibility of user interaction in the planning process involves the descriptive unit. As briefly explained this unit contains descriptions of plan generations. Such a description of a planning step is called a **method**. The user can devise his own methods or fall back on those comprised in a method base. Methods that are chosen to be employed in the current proof process are included in the current methods component. The conflict manager has to choose which one of those is activated. For this purpose a method does not only contain a precise description of a problem segmentation, a program, but also information, called conflict information, on how to evaluate its possible success in the current situation in comparison to other methods. However, a method is usually generated to be applied in specific cases, only. Preconditions, decidable functions using sensors, describe requirements for the state of a proof process that have to be met to ensure the methods adequate application. This way the amount of conflicting choices is reduced without the need of complex activities performed by the conflict manager. The conflict information then allows the remaining choices to be evaluated in connection with the current system's state. For this task it consists of decidable functions of which the results are natural numbers. The conflict manager can order the methods according to the value of the respective conflict information, activating the one with the highest value. A further means to assist the conflict manager is provided in the form of postconditions. They describe the expected change to the poof tree's state induced by the programs application. Examples for functions appearing in postconditions are: $verifiable(node)$ and $insufficient(node)$. A node is marked verifiable (in the additional information of a node) when the verifier is expected to proof the node's theorem with the exclusive help of the information provided by the data base and the node's sons. If a node's information is not considered sufficient for the proof of its father's theorem by the verifier, this node is marked as insufficient.

For the development of programs a programming language has to be provided which also includes a whole range of basic function for the syntactical manipulation and analysis of theorems. In addition, programs have to express the proposed changes to a planning tree, as for example the creation of a new node that is to be attached as a son to the currently discussed node. The objectives of such a program mark one of the differences between the concept of a method described here and the one proposed in [HKK92]. A program or "procedural content" from the latter's point of view solely serves as an interpreter for the declarative content of a method.

2.3 Elaboration of the architecture

This section serves to deepen the understanding of the planning components introduced in the previous chapter. For this purpose more precise and detailed information is presented which is illustrated by examples. The examples have been derived from the verification of sorting algorithm. The algorithms that are discussed here are: Quick Sort, Merge Sort and Insertion Sort. Their specification, as well as a description of the verification problem can be found in the appendix. Note that we assume the reader to have a basic knowledge of equational theories and inductive proving as we will be dealing with inductive proof processes in equational theories [Bac87, Ave90, Ave91]. Therefore, the functions required for the examples are described by equations. We presume the existence of a reduction ordering by which the equations of the specifications can be ordered from left to right. This way they can be seen as rewrite rules.

Furthermore, unless otherwise stated, variables used in an equation are not related to those of any other equation. To simplify the connection between variables and their respective sorts for the reader, they are named according to the following pattern: $n, m, q, n1, m1, q1, ...$ which represent natural numbers. The variables $B, b, B1, b1, ...$ refer to boolean values and $l, l1, l2, ...$ to lists. The set of boolean values, natural numbers and the set of lists will be referred to, respectively, by $BOOL$, NAT and $LIST$. Variables representing function symbols are preceded by a question mark, for example $?symb$.

The generation of a plan, that is to say the suggestion of helpful lemmata consists of a deduction of information that is implicitly given by the defining equations. To be more exact, an analysis of the syntax of equations can lead to semantically correct statements. As the examination of our examples has shown, even incorrect statements are of use to the planning process. A slight moderation of these incorrect ones has generally led to assertions that proved to be crucial to the proof process. It is clear that the syntax can not reveal all of the semantical meaning of an equation. This deficiency is acknowledged by enabling moderation of the syntactically deduced equations by the user. However, the syntax allows the user to be guided along the right path. We believe this support to be a great achievement. Based on these reflections, the following examples pay special attention to syntactical analysis. To enable a better understanding, the insights deduced in such a way are described somewhat informally. This, of course, only disguises the detailed technical description needed for a realization of the suggested system.

$Sensor_1$	
Description	$currentTheorem_{lhs}$: function ord is applied to a term of which the leading function symbol is a defined function symbol $?sort$; $?sort$ possesses only one argument of the sort $LIST$
Program	if $currentTheorem_{lhs} = ord(?sort(t))$ and **fs-sort**$(?sort) = LIST$ then $return(true)$ else $return(false)$
$endSensor_1$	

Figure 3: An example of a sensor

Let us proceed by presenting the function which expresses the part of verification we will consider here. This function, referred to by ord, will have the value $true$ if the elements of its input list l are ordered according to the relation $\leq$. Otherwise the function will return false. For a complete description of the verification, the characterization of a function $perm$ is essential, where $perm$ tests whether a list is a permutation of another one. Nevertheless, this function is neglected here as the presentation of the examples concentrates on the first feature of the verification problem, only.

$$ord : LIST \rightarrow BOOL$$

$$ord(nil) = true$$
$$ord(cons(n, l)) = and(\leq_{nl} (n, l), ord(l))$$

The questions is now whether the application of a sorting algorithm $?sort$ to a list l returns a sorted list in the above sense. This is expressed by the theorem $ord(?sort(l)) = true$ to be proven. In such a situation the conflict manager is expected to activate methods fitted to the problem. The connection of a method to the problem at hand is expressed in the precondition, either by a decidable function or a sensor. We have opted for the latter because of two reasons. Firstly, it will be seen that this precondition is not only going to be used for one but more methods. Thus a sensor avoids additional leg-work and provides a logical, easily notable connection between these methods. Secondly, sensors are meant to monitor the proof process for the conflict manager such that note-worthy incidents can promptly be reacted to. The appearance of an unproved theorem of the above kind is certainly worth noting as methods for exactly such theorems exist. This way the number of methods to be chosen from can easily be reduced. The sensor describing the above verification situation, named $Sensor_1$, is given in figure 3, followed by an example for a method, $Method_1$, in figure 4. All sensors and methods referred to in the examples are listed in the appendix.

For a first example the global variable $?sort$, used by the sensors and some methods, will be instantiated by the function $isort$, which describes an Insertion Sort algorithm. Note that all variables used in sensors are, if not otherwise stated, global variables. The evaluation of a sensor binds those variables to values, which can further be processed

$Method_1$	
Description	Generation of the left hand side of a new theorem $newTheorem$ from the recursive case of ?sort (generalize each recursive occurrences of ?sort in the right hand side of the recursive case by a new, distinctive variable)
Precondition	$Sensor_1 = true(?sort)$
Conflict information	$planning(currentNode)$

Program	$t =$ **fs-rec-rhs**$(?sort)$
	$m =$ **fs-occur**$(?sort, t)$
	For $i = 1$ to m do
	    $l_i =$ **var-gen**
	    $t_i =$ **fs-subterm**$(t, ?sort)$
	    $t =$ **fs-replace**$(t, ?sort, l_i)$
	endfor
	$newTheorem_{lhs} = ord(t)$
	$newTheorem_{rhs} = and_i^m(ord(l_i))$
	nd-attach($currentNode,$
	        nd-new($newTheorem, n,$
	                $Method_1, none)$
	        )

Postcondition	$verifiable(currentNode)$
$endMethod_1$	

Figure 4: An example of a method

by methods. To each sensor being stated in the precondition of a method a list of the influenced global variables is attached.

$$isort : LIST \rightarrow LIST$$

$$isort(nil) = nil$$
$$isort(cons(n, l)) = ins(n, isort(l))$$

Thus the main theorem to be dealt with in the proof process is of the form:

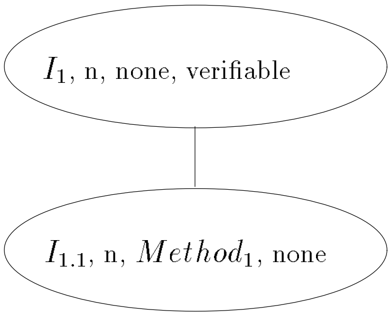
$$ord(isort(l)) = true$$

$Sensor_1$ notifies the conflict manager of the form of the theorem to be proven. The latter then chooses those methods of which the precondition demands $Sensor_1$ to be activated. However, $?sort$ being bound to $isort$ and the occurrence of its value on the left hand side of the current theorem results in $Sensor_4$ being evaluated to $true$. As a consequence two methods, $Method_1$ and $Method_2$, have to be examined for possible success.

The conflict manager then analyzes the conflict information of the chosen methods. The decidable functions chosen as conflict information in the examples consist of the following three functions: $planning(node)$, $replanning(node)$ and $refinement(node)$. Each of the three functions will yield a number $n > 0$ if the node it operates on is not yet verified and complies to the condition depicted by the function. If this is not the case, their value will be 0. The function $planning$ tests whether the node in question is a leaf while $replanning$ demands the existence of at least one son that stands for an unsuccessful subplan. This will be the case if the son's planning state is marked rejected (r) or unsuccessful (u). The third function's value will be n if the examined node possesses at least one son of which the planning state is either non-verified (n) or verified (v). A method that leads to the extension of a node such that the resulting theorems of the new sons can not be expected to exclusively lead to the verification of the node's theorem, should feature $refinement$ in the conflict information. In case the effects of a method on a plan are generally very complex and therefore only sensible whenever no other less complex and just as promising method for the current system state exists, this method's conflict information should include $replanning$. Concerning the examples it is of no importance for the conflict manager's decisions which number n stands for exactly. This is why we will not specify it any further. It is clear that the more methods are loaded into the system, the more tuned the conflict information has to be. However, specific guide-lines can only be hoped for after extensive testing of an implemented system.

In the case of $Method_1$ and $Method_2$ the evaluation of the respective conflict information according to the above description of $planning$ and $replanning$ leads to the choice of $Method_1$. The application of this method to the current node I_1 results in the following planning tree [2].

[2] $and_i^n(t_i) = t_i$ if $n = 1$; $and_i^n(t_i) = and(t_1, and(t_2, ..., and(t_{n-1}, t_n)...))$ if $n > 1$



I_1: $ord(isort(l)) = true$

$I_{1.1}$: $ord(ins(n, l)) = ord(l)$

Due to the postcondition of $Method_1$ the additional information of the node I_1 has been set to verifiable. When deciding on the next step to be taken, the conflict manager falls back on this information. It can either refine or verify the current plan. For this example the additional information of I_1 leads to a preference of its verification over immediate refinement steps. This is why the conflict manager then activates the verifier with the additional theorem $I_{1.1}$. We will show that I_1 can be straightforwardly proven by a verifier using rewrite methods. In future examples verification will not always be explicitly outlined. Nevertheless, we have, for the presented cases, tested it with the inductive theorem prover UNICOM [GL91].
Instantiation of l by nil and $cons(n, l)$ results in two equations (critical pairs) that cover theorem I_1:
$ord(isort(nil)) = true$ and $ord(isort(cons(n, l))) = true$.
$ord(isort(nil)) = true$ can easily be transformed into a trivial equation by use of definition rules, only:

$ord(\underline{isort(nil)}) = true$

$\Downarrow \qquad isort(nil) = nil$

$\underline{ord(nil)} = true$

$\Downarrow \qquad ord(nil) = true$

$true = true$

The desired transformation of the second equation requires the use of theorem $I_{1.1}$ in addition to definition rules and the induction hypothesis IH (I_1 itself):

$ord(\underline{isort(cons(n, l))}) = true$

$\Downarrow \qquad isort(cons(n, l)) = ins(n, isort(l))$

$\underline{ord(ins(n, isort(l)))} = true$

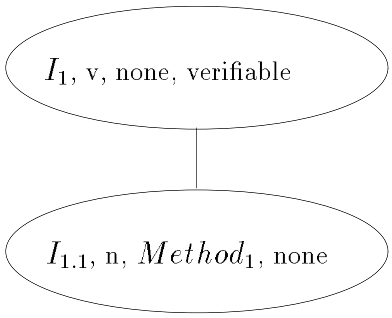
$\Downarrow \qquad ord(ins(n, l)) = ord(l) \qquad I_{1.1}$

$\underline{ord(isort(l))} = true$

$$\Downarrow \qquad ord(isort(l)) = true \qquad \text{IH, } I_1$$

$true = true$

An update of the planning tree, the new plan, marks theorem I_1 as verified.



I_1: $ord(isort(l)) = true$

$I_{1.1}$: $ord(ins(n, l)) = ord(l)$

This leaves $I_{1.1}$ to be further planned, the current plan has to be refined. Two sensors react to the changed situation, $Sensor_2$ and $Sensor_3$. Both of them are used as preconditions, $Sensor_2$ in $Method_4$ and $Sensor_3$ in $Method_3$. The identical conflict information of the two methods, $refinement$, does not result in a preference of any one of the two. However, as both feature the postcondition insufficient in connection with node $I_{1.1}$, the conflict manager does not have to choose between the two methods. Instead the plan is refined with the help of both methods. $Sensor_2$ concentrates on the function ins which fits the sensor's description of the symbol $?symb$:

$$ins : NAT\ LIST \rightarrow LIST$$

$$\begin{aligned}
&ins(n, nil) = cons(n, nil)\\
&ins(n, cons(m, l)) = if - list(\leq (n, m),\\
&\qquad\qquad\qquad\qquad cons(n, cons(m, l)),\\
&\qquad\qquad\qquad\qquad cons(m, ins(n, l)))
\end{aligned}$$

The left hand side of the new theorem $newTheorem$ is then generated by replacing the subterm $ins(n, l)$ by an if-then-else construct. The new subterm has to be adapted to the sort of the replaced function. This is why the if-then-else construct on lists is chosen for the example. The right hand side of the new theorem evolves from purely boolean transformations of the if-then-else statement. It is presented for function symbols of arity 1, but can be easily adapted to symbols with greater arities:

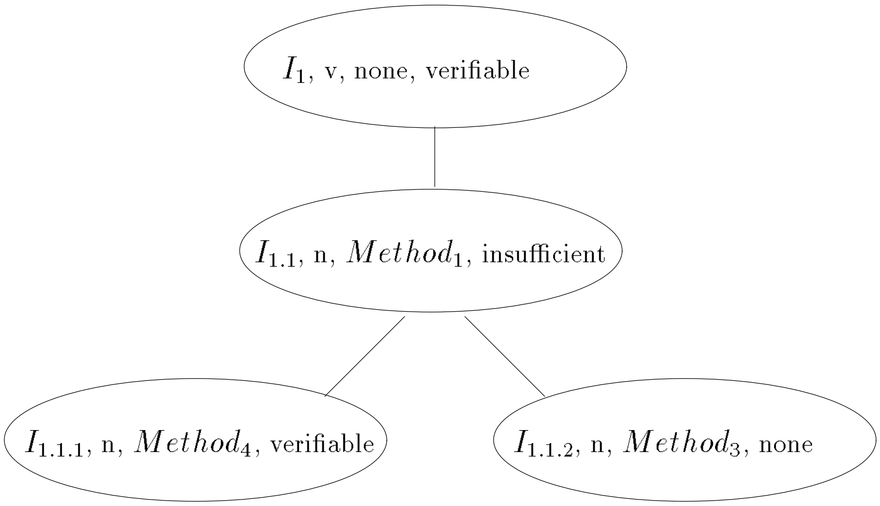
$$\begin{aligned}
?symbol(if(B, t_1, t_2)) &= if(B, ?symbol(t_1), ?symbol(t_2))\\
&= or(\ or(and(not(B), ?symbol(t_2)), and(B, ?symbol(t_1))),\\
&\qquad and(?symbol(t_1), ?symbol(t_2)))
\end{aligned}$$

Keeping this in mind $Method_4$ yields equation $I_{1.1.1}$:

$I_{1.1.1}$: $ord(if - list(B, l1, l2)) = or(\ or(and(not(B), ord(l2)), and(B, ord(l1))),$
$\qquad and(ord(l1), ord(l2)))$

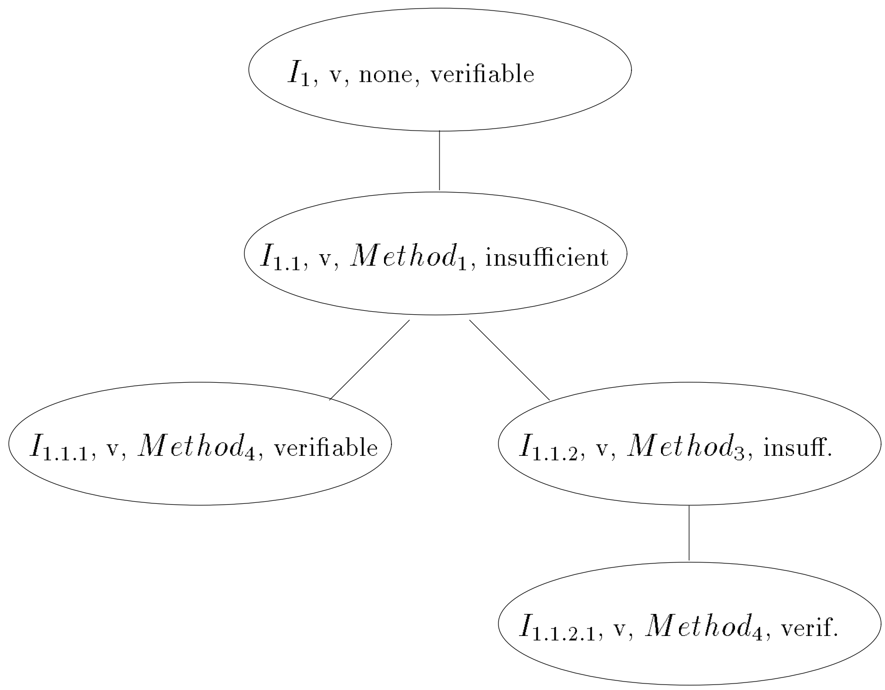
The handling of such theorems is not only important for the verification of sorting algorithms but for inductive theorem proving, in general. As such it has already been proposed by Boyer and Moore [BM79].

$Method_3$ considers the connection of ord and the list provided by the connection of the relation $\leq_{nl}$ (occurring in the recursive case of ord) and the list provided by the function to which ord is applied. For the example of Insertion Sort the connection is expressed by the term $\leq_{nl}(n, ins(m, l))$, the left hand side of the proposed theorem. The right hand side is then developed from the left hand one by a conjunction of relations of the form $\leq_{nl}(n, t_i)$ or $\leq(n, t_i)$ for every subterm t_i of the left hand side's second argument. The choice of the relation depends on the sort of the subterms. Thus $I_{1.1.2}$ and the resulting planning tree are of the following form:
$I_{1.1.2}$: $\leq_{nl}(n, ins(m, l)) = and(\leq(n, m), \leq_{nl}(n, l))$



The verification of $I_{1.1}$ with UNICOM employs both new theorems. Therefore, they still have to be proven. While the proof of $I_{1.1.2}$ requires the plan to be further refined, $I_{1.1.1}$ can be verified with the exclusive help of boolean knowledge and some obvious statements on natural numbers. This is why we consider $I_{1.1.1}$ to be verified without further planning. In the appendix the needed statements on natural numbers are added to the final planning tree for completeness sake but they are neglected here.

For the proof of theorem $I_{1.1.2}$ the additional theorem $I_{1.1.2.1}$ is generated by the planning component in the same way as $I_{1.1.1}$:
$I_{1.1.2.1}$: $\leq_{nl}(n, if\!-\!list(B, l1, l2)) = or(\; or(and(not(B), \leq_{nl}(n, l2)), and(B, \leq_{nl}(n, l1))), and(\leq_{nl}(n, l1), \leq_{nl}(n, l2)))$
This is to say $Sensor_2$ indicates the activation of $Method_4$. As this is the only matching method, the conflict manager attempts verification with the resulting additional theorem, only. This action ends successfully, the planning tree is complete.

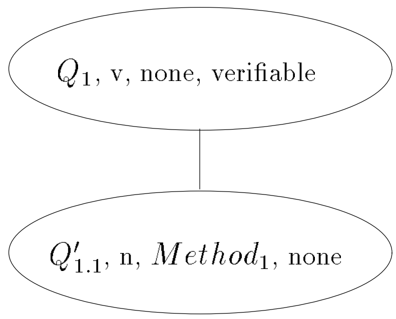


The proof of the Merge Sort algorithm specified in the appendix is planned analogously with the one of Insertion Sort. This is why it will not be further discussed here. Instead some remarks on the example of a Quick Sort algorithm will be given.

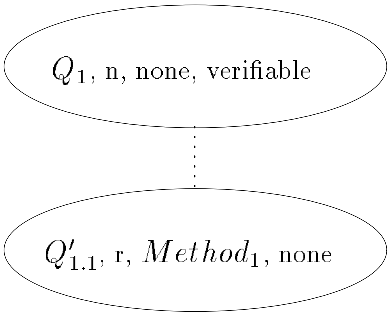
$$qsort : LIST \rightarrow LIST$$

$$\begin{aligned}
&qsort(nil) = nil \\
&qsort(cons(n, l)) = app(\ qsort(lower(n, l)), \\
&\qquad\qquad\qquad\qquad\quad\ cons(n, qsort(greater(n, l))))
\end{aligned}$$

As in the previous example, the proof process proceeds by applying $Method_1$ to a planning tree consisting of the equation Q_1: $ord(qsort(l)) = true$. This results in the generation of $Q'_{1.1}$: $ord(app(l_1, cons(n, l_2))) = and(ord(l_1), ord(l_2))$. Q_1 can be verified with the help of $Q'_{1.1}$.



The proof of $Q'_{1.1}$, however, poses certain problems. No method fits the above situation. The conflict manager either activates the verifier for $Q'_{1.1}$ (bounded by time) or asks for a user's advice. Whatever the choice, any attempt to verify this particular theorem will lead to the result that it is incorrect, it is marked as rejected. This implies that the proof of Q_1 has to be replanned. To avoid planning loops, the failed plan cannot be neglected. This way the conflict manager can avoid methods that have led to failures. Thus, the current enhanced planning tree for the example is of the following form:



Of all the methods presented $Method_2$ is chosen by the conflict manager as the only appropriate one. Not only is its precondition satisfied, $Sensor_4$, but it is also especially marked for replanning by its conflict information. The first theorem generated by $Method_2$ takes the relation between the pivot element used by Quick Sort algorithms and the two lists that are derived from the original one with the help of the pivot element into consideration. In addition to these two lists being ordered, each element of the first list is supposed to be smaller than the pivot element ($\leq_{ln} (l_1, n)$). Each element of the second list is presumed to be greater than the pivot ($\leq_{nl} (n, l_2)$). This reasoning leads to theorem $Q_{1.1}$:

$Q_{1.1}$: $ord(app(l1, cons(n, l2))) = and(\ ord(l1), and(ord(l2),$
$\qquad and(\leq_{ln} (l1, n), \leq_{nl} (n, l2))))$

From this theorem four more are derived for planning the proof of Q_1. A closer look at the verification of Q_1 with these new theorems helps to illustrate the idea of $Method_2$.

Main conjecture to be proven:
Q_1: $ord(qsort(l)) = true$
Instantiation of l by nil and $cons(n, l)$ leads to two equations that cover the conjecture: $ord(qsort(nil)) = true$ and $ord(qsort(cons(n, l))) = true$.

$ord(qsort(nil)) = true$ can easily be transformed into a trivial equation by use of definition rules, only:

$ord(\underline{qsort(nil)}) = true$

$\Downarrow \qquad qsort(nil) = nil$

$\underline{ord(nil)} = true$

$\Downarrow \qquad ord(nil) = true$

$true = true$

The desired transformation of the second equation requires the use of the five theorems listed above in addition to defined rules, the induction hypothesis (Q_1 itself) and boolean knowledge:

$$ord(\underline{qsort(cons(n,l))}) = true$$

$$\Downarrow \qquad qsort(cons(n,l)) = app(\ qsort(lower(n,l)), cons(n, qsort(greater(n,l))))$$

$$\underline{ord(app(qsort(lower(n,l)), cons(n, qsort(greater(n,l))))) = true}$$

$$\Downarrow \qquad ord(app(l1, cons(n, l2))) = and(\ ord(l1), and(ord(l2), and(\leq_{ln}(l1,n), \leq_{nl}(n,l2)))) \qquad Q_{1.1}$$

$$\begin{aligned} and(\underline{ord(qsort(lower(n,l)))},\ & and(ord(qsort(greater(n,l))), \\ & and(\ \leq_{ln}(qsort(lower(n,l)),n), \\ & \quad \leq_{nl}(n, qsort(greater(n,l)))))) = true \end{aligned}$$

$$\Downarrow \qquad ord(qsort(l)) = true \qquad \text{IH}, Q_1$$

$$\begin{aligned} & and(true,\ and(\underline{ord(qsort(greater(n,l)))}, \\ & \quad and(\leq_{ln}(qsort(lower(n,l)),n), \leq_{nl}(n, qsort(greater(n,l)))))) = true \end{aligned}$$

$$\Downarrow \qquad ord(qsort(l)) = true \qquad \text{IH}, Q_1$$

$$\begin{aligned} and(true, and(true, and(\ & \underline{\leq_{ln}(qsort(lower(n,l)),n)}, \\ & \underline{\leq_{nl}(n, qsort(greater(n,l)))}))) = true \end{aligned}$$

$$\Downarrow \qquad \leq_{ln}(qsort(l),n) = \leq_{ln}(l,n) \qquad Q_{1.2}$$

$$and(true, and(\leq_{ln}(lower(n,l),n), and(true, \underline{\leq_{nl}(n, qsort(greater(n,l)))}))) = true$$

$$\Downarrow \qquad \leq_{nl}(n, qsort(l)) = \leq_{nl}(n,l) \qquad Q_{1.3}$$

$$and(true, and(\underline{\leq_{ln}(lower(n,l),n)}, and(true, \leq_{nl}(n, greater(n,l))))) = true$$

$$\Downarrow \qquad \leq_{ln}(lower(n,l),n) = true \qquad Q_{1.4}$$

$$and(true, true, and(true, \underline{\leq_{nl}(n, greater(n,l))}))) = true$$

$$\Downarrow \qquad \leq_{nl}(n, greater(n,l)) = true \qquad Q_{1.5}$$

$$and(true, and(true, \underline{and(true, true)}))$$

$$\Downarrow \qquad and(true, b) = b$$

$$and(true, \underline{and(true, true)}) = true$$

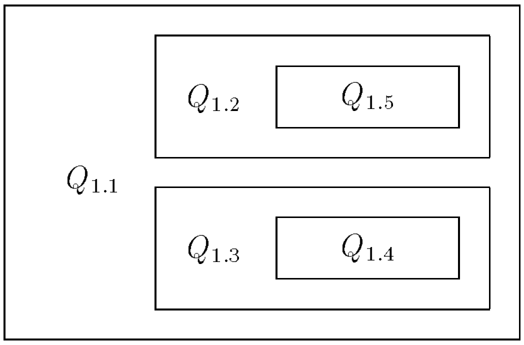
$$\Downarrow \qquad and(true, b) = b$$

$\underline{and(true, true)} = true$

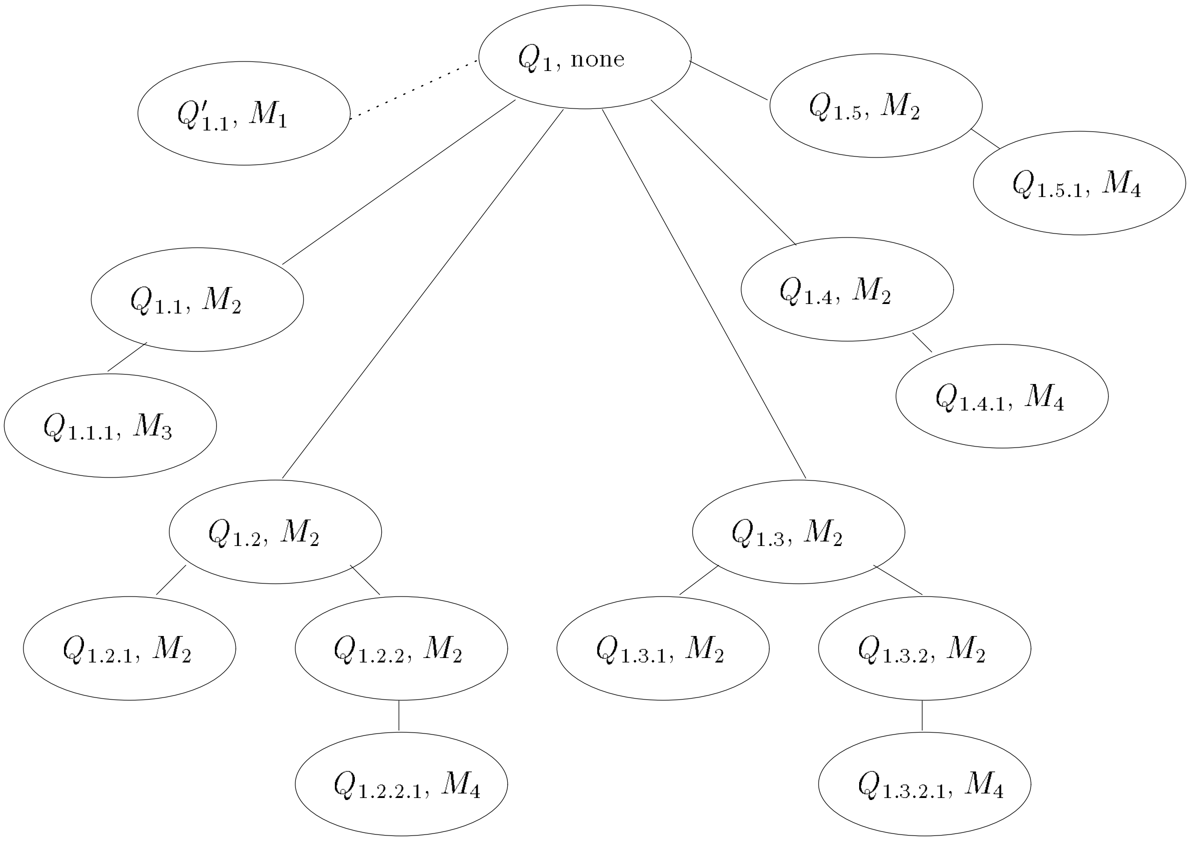
$$\Downarrow \qquad and(true, b) = b$$

$true = true$

Let us now have a closer look at the influence of the five additional theorems on the proof. $Q_{1.1}$ moves the function symbol ord into the left hand side of the conjecture, such that the induction hypothesis can be applied. This process has led to a simplification of the original equation. However, the resulting left hand side still contains components that have to be simplified, namely $\leq_{ln}$ $(qsort(lower(n, l)), n)$ and $\leq_{nl}$ $(n, qsort(greater(n, l)))$. These components result from the application of $Q_{1.1}$. Note that their subterms are either variables or terms that have been replaced for variables by $Method_2$ while constructing $Q_{1.1}$. This is to say that the generalized subterms can be of importance if they are not eliminated during verification by application of the generalized theorem in connection with the induction hypothesis. This is why the generation of $Q_{1.2}$ and $Q_{1.3}$ requires knowledge about the right hand side of $Q_{1.1}$ (especially about its subterms containing generalized variables but not as arguments to the function ord). However, their generalization is not retracted at once but step by step. This is reflected by the dependency of $Q_{1.4}$ ($Q_{1.5}$) on $Q_{1.2}$ ($Q_{1.3}$). The following figure illustrates the refinement process for this example with the help of nested boxes. The inner boxes can only be unravelled after dealing with the outer ones. Note, that the depth of nesting corresponds to the index of the variable $TeList_j$ in the program of $Method_2$.



We conclude this section by presenting the verified planning tree and a list of the theorems generated while planning the proof of Q_1. The node information is reduced to the theorem's identifier and the abbreviated name of the respective method, for example M_1 instead of $Method_1$. Once again the obvious theorems on boolean and natural functions are neglected. Note the extensive use of $Method_4$. $Method_2$ is chosen for two further refinement steps. The explanation concerning its first application can be adapted to the changed circumstances by replacing the function ord by $\leq_{nl}$ or $\leq_{ln}$, respectively.



$Q_{1.1.1}$: $\le_{nl} (n, app(l1, cons(m, l2))) = and(\le (n, m), and(\le_{nl} (n, l1), \le_{nl} (n, l2)))$

$Q_{1.2.1}$: $\le_{ln} (app(l1, cons(n, l2)), m) = and(\le (n, m), and(\le_{ln} (l1, m), \le_{ln} (l2, m)))$

$Q_{1.2.2}$: $and(\le (n, m), and(\le_{ln} (lower(n, l), m), \le_{ln} (greater(n, l), m))) = and(\ \le (n, m), \le_{ln} (l, m))$

$Q_{1.2.2.1}$: $\le_{ln} (if{-}list(B, l1, l2), n) = or(\ or(\ and(not(B), \le_{ln} (n, l2)), and(B, \le_{ln} (n, l1))), and(\le_{ln} (n, l1), \le_{ln} (n, l2)))$

$Q_{1.3.1}$: $\le_{nl} (n, app(l1, cons(m, l2))) = and(\le (n, m), and(\le_{nl} (n, l1), \le_{nl} (n, l2)))$

$Q_{1.3.2}$: $and(\le (n, m), and(\le_{nl} (m, lower(n, l)), \le_{nl} (m, greater(n, l)))) = and(\ \le (n, m), \le_{nl} (m, l))$

$Q_{1.3.2.1}$: $\le_{nl} (n, if{-}list(B, l1, l2)) = or(\ or(\ and(not(B), \le_{nl} (n, l2)), and(B, \le_{nl} (n, l1))), and(\le_{nl} (n, l1), \le_{nl} (n, l2)))$

$Q_{1.4.1}$: $\le_{ln} (if{-}list(B, l1, l2), n) = or(\ or(and(not(B), \le_{ln} (l2, n)), and(B, \le_{ln} (l1, n))), and(\le_{ln} (l1, n), \le_{ln} (l2, n)))$

$Q_{1.5.1}$: $\le_{nl} (n, if{-}list(B, l1, l2)) = or(\ or(and(not(B), \le_{nl} (n, l2)), and(B, \le_{nl} (n, l1))), and(\le_{nl} (n, l1), \le_{nl} (n, l2)))$

3 Conclusions

In this paper we have presented a system architecture for combining planning with automatic theorem proving. It was our intent to develop the architecture such that it can be employed to extend existing automatic theorem provers by planning aspects. Our desire is to be able to exploit the knowledge represented by existing systems without the need of transfering it into a new one. First experiments in connection with the inductive theorem prover UNICOM have already illustrated a considerable improvement of the enhanced system, as detailed. The proto-type that is presently implemented in Common-LISP on Spark-Workstations has so far been straight-forward and has not led to any major conceptual changes.

Currently, we are extending our research to further domains to broaden our experience and to strengthen the positive results. Our ultimate goal lies in the development of a range of methods and sensors such that inductive theorem proofs can be planned without too specific knowledge concerning the domain a theorem belongs to. However, this can only be achieved after thorough testing of our new system. It is our believe that the methods and sensors of the examples presented can then be further generalized, leading to smaller and somewhat less complex planning steps. The flexibility of adapting methods and sensors to the changing conceptions of a proof planner enable systems - following this architecture - to be a powerful tool for assisting proof planners with a multitude of backgrounds.

A Specifications

Starting with boolean algebra, the required base systems of defining equations are described. Each of them is defined completely over a set of constructors. The set of constructors of the boolean algebra contains $true$ and $false$ over which the well-known boolean functions and, or, not and $if-bool$ are specified.

Specification A.0.1 (Boolean functions)

- $true :\rightarrow BOOL$
- $false :\rightarrow BOOL$
- $not : BOOL \rightarrow BOOL$

  $not(true) = false$
  $not(false) = true$
- $and : BOOL\ BOOL \rightarrow BOOL$

  $and(true, b) = b$
  $and(false, b) = false$
- $or : BOOL\ BOOL \rightarrow BOOL$

  $or(true, b) = true$
  $or(false, b) = b$
- $if-bool : BOOL\ BOOL\ BOOL \rightarrow BOOL$

  $if-bool(true, b1, b2) = b1$
  $if-bool(false, b1, b2) = b2$

The constructor set of natural numbers comprises the number 0 and the successor function s. Three functions which operate on natural numbers and on boolean values are needed: eq, $\leq$ and $if-nat$. eq is used to check the equality of two natural numbers while $\leq$ tests whether a number is less or equal to a second one. The function $if-nat$ serves to improve the readability of the specification of the sorting algorithms. Semantically it is the same as $if-bool$ except that it returns natural numbers instead of boolean values.

Specification A.0.2 (Functions on natural numbers)

- $0 :\rightarrow NAT$

- $s : NAT \to NAT$
- $eq : NAT\ NAT \to NAT$

  $eq(0, 0) = true$
  $eq(0, s(n)) = false$
  $eq(s(n), 0) = false$
  $eq(s(n), s(m)) = eq(n, m)$

- $\leq: NAT\ NAT \to NAT$

  $\leq (0, n) = true$
  $\leq (s(n), 0) = false$
  $\leq (s(n), s(m)) = \leq (n, m)$

- $if - nat : BOOL\ NAT\ NAT \to NAT$

  $if - nat(true, n, m) = n$
  $if - nat(false, n, m) = m$

In addition to the constructors nil, representing the empty list, and $cons$, which adds a natural number to the beginning of a list, the functions *app* and $if - list$ require mentioning. *app* appends two lists to form one list, only.

Specification A.0.3 (Functions on lists)

- $nil :\to LIST$
- $cons : NAT\ LIST \to LIST$
- $app : LIST\ LIST \to LIST$

  $app(nil, l) = l$
  $app(cons(n, l1), l2) = cons(n, app(l1, l2))$

- $if - list : BOOL\ LIST\ LIST \to LIST$

  $if - list(true, l1, l2) = l1$
  $if - list(false, l1, l2) = l2$

Two functions that compare a natural number with a list of numbers, namely $\leq_{nl}$ and $\leq_{ln}$ are used for specifying sorting algorithms. On the one hand $\leq_{nl}$ checks whether a given natural number is less or equal to every natural number comprised in a given list. $\leq_{ln}$, on the other hand, tests whether a given natural number n is greater or equal to every natural number of a given list.

Specification A.0.4 (Functions on lists)

- $\leq_{nl}$: $NAT\ LIST \rightarrow BOOL$

  $\leq_{nl}(n, nil) = true$
  $\leq_{nl}(n, cons(m, l)) = and(\leq(n, m), \leq_{nl}(n, l))$

- $\leq_{ln}$: $LIST\ NAT \rightarrow BOOL$

  $\leq_{ln}(nil, n) = true$
  $\leq_{ln}(cons(n, l), m) = and(\leq(n, m), \leq_{ln}(l, m))$

The previous specifications enable the specification of the following versions of the sorting algorithms Insertion Sort, Merge Sort and Quick Sort.

Specification A.0.5 (Insertion Sort)

- $isort : LIST \rightarrow LIST$

  $isort(nil) = nil$
  $isort(cons(n, l)) = ins(n, isort(l))$

- $ins : NAT\ LIST \rightarrow LIST$

  $ins(n, nil) = cons(n, nil)$
  $$ins(n, cons(m, l)) = if-list(\leq(n, m),\ cons(n, cons(m, l)),\ cons(m, ins(n, l)))$$

Specification A.0.6 (Merge Sort)

- $msort : LIST \rightarrow LIST$

  $msort(nil) = nil$
  $msort(cons(n, nil)) = cons(n, nil)$
  $$msort(cons(n, cons(m, l))) = merge(msort(cons(n, split1(l))),\ msort(cons(m, split2(l))))$$

- $merge : LIST\ LIST \rightarrow LIST$

  $merge(nil, l) = l$
  $merge(l, nil) = l$
  $$merge(cons(n, l1), cons(m, l2)) = if-list(\leq(n, m),\ cons(n, merge(l1, cons(m, l2))),\ cons(m, merge(cons(n, l1), l2)))$$

- $split1 : LIST \rightarrow LIST$

  $split1(nil) = nil$
  $split1(cons(n, nil)) = cons(n, nil)$
  $split1(cons(n, cons(m, l))) = cons(n, split1(l))$

- $split2 : LIST \rightarrow LIST$

  $split2(nil) = nil$
  $split2(cons(n, nil)) = nil$
  $split2(cons(n, cons(m, l))) = cons(m, split2(l))$

Specification A.0.7 (Quick Sort)

- $qsort : LIST \rightarrow LIST$

  $qsort(nil) = nil$
  $$\begin{aligned} qsort(cons(n, l)) = app(\ & qsort(lower(n, l)), \\ & cons(n, qsort(greater(n, l)))) \end{aligned}$$

- $lower : NAT\ LIST \rightarrow LIST$

  $lower(n, nil) = nil$
  $$\begin{aligned} lower(n, cons(m, l)) = if - list(& \leq (m, n), \\ & cons(m, lower(n, l)), \\ & lower(n, l)) \end{aligned}$$

- $greater : NAT\ LIST \rightarrow LIST$

  $greater(n, nil) = nil$
  $$\begin{aligned} greater(n, cons(m, l)) = if - list(& \leq (m, n), \\ & greater(n, l), \\ & cons(m, greater(n, l))) \end{aligned}$$

B Sensors

$Sensor_1$

Description	$currentTheorem_{lhs}$: function ord is applied to a term of which the leading function symbol is a defined function symbol $?sort$; $?sort$ possesses only one argument of the sort $LIST$
Program	if $currentTheorem_{lhs} = ord(?sort(t))$ and **fs-sort**$(?sort) = LIST$ then $return(true)$ else $return(false)$

$endSensor_1$

$Sensor_2$

Description	$currentTheorem_{lhs}$ contains a defined function symbol $?symb$; $?symb$ does not appear in $currentTheorem_{rhs}$; one of the recursive cases of the definition of $?symb$ is an $if - then - else$ construction
Program	if **fs-contain**$(?symb, currentTheorem_{lhs}) = true$ and **fs-defined**$(?symb) = true$ and **fs-contain**$(?symb, currentTheorem_{rhs}) = false$ and **fs-top**(**fs-rec-rhs**$(?symb)) = ?if - then - else$ then $return(true)$ else $return(false)$

$endSensor_2$

$Sensor_3$

Description	$currentTheorem_{lhs}$: function ord is applied to a term t of which the defined function symbol has an arity greater than 1
Program	if $currentTheorem_{lhs} = ord(?symb(t_1, ..., t_n))$ and **fs-defined**$(?symb) = true$ and **fs-arity**$(?symb) > 1$ then $return(true)$ else $return(false)$

$endSensor_3$

$Sensor_4$	
Description	The global variable ?$sort$ has been instantiated and its value occurs in $currentTheorem_{lhs}$
Program	if **var-global-setp**(?$sort$) $= true$ then if **fs-contain**(?$sort, currentTheorem_{lhs}$) then $return(true)$ else $return(false)$ else $return(false)$
$endSensor_4$	

C Methods

$Method_1$	
Description	Generation of the left hand side of a new theorem $newTheorem$ from the recursive case of ?sort (generalize each recursive occurrences of ?sort in the right hand side of the recursive case by a new, distinctive variable)
Precondition	$Sensor_1 = true(?sort)$
Conflict information	$planning(currentNode)$

Program	$t =$ **fs-rec-rhs**(?$sort$)
	$m =$ **fs-occur**(?$sort, t$)
	For $i = 1$ to m do
	$l_i =$ **var-gen**
	$t_i =$ **fs-subterm**(t, ?$sort$)
	$t =$ **fs-replace**(t, ?$sort, l_i$)
	endfor
	$newTheorem_{lhs} = ord(t)$
	$newTheorem_{rhs} = and_i^m(ord(l_i))$
	nd-attach($currentNode$,
	nd-new($newTheorem, n$,
	$Method_1, none$)
	)

Postcondition	$verifiable(currentNode)$
$endMethod_1$	

$Method_2$	
Description	Generation of several new theorems. The first, $newTheorem$ is generated from the left hand side of the recursive case of $?sort$ with the help of generalization. Further theorems are thereof created by stepwise taking the generalization back.
Precondition	$Sensor_4 = true(?sort)$
Conflict information	$replanning(currentNode)$
Program	see figure 5 and figure 6
Postcondition	$insufficient(currentNode)$
$endMethod_2$	

$Method_3$	
Description	Generation of a theorem that shows dependencies of a symbol $?symb$, that the function *ord* is applied to, and the function $\leq_{nl}$ used in the recursive case of *ord*.
Precondition	$Sensor_3 = true(?symb(t_1, ..., t_n))$
Conflict information	$refinement(currentNode)$

Program	$n =$ **fs-arity**$(?symb)$
	$newTheorem_{lhs} = \leq_{nl} (m, ?symb(t_1, ..., t_n))$
	For $i = 1$ to n do
	    if **fs-sort**(**fs-top**(t_i)) $= LIST$
	    then $? \leq_i = \leq_{nl}$
	    else $? \leq_i = \leq$
	    $newTheorem_{rhs} = and_i^n(? \leq_i (m, t_i))$
	endfor
	nd-attach($currentNode$,
	        nd-new($newTheorem, n,$
	                $Method_3, none$)
	        )

Postcondition	$insufficient(currentNode)$
$endMethod_3$	

$t =$ **fs-rec-rhs**$(?sort)$
$m =$ **fs-occur**$(?sort, t)$
$varTeList_0 = \emptyset$
$varGen = \emptyset$
For $i = 1$ to m do
  $l_i =$ **var-gen**
  $t_i =$ **fs-subterm**$(t, ?sort)$
  $varTeList_0 = varTeList_0 \cup \{(l_i, t_i)\}$
  $varGen = varGen \cup \{l_i\}$
  $t =$ **fs-replace**$(t, ?sort, l_i)$
endfor
$newTheorem_{lhs} =$ **fs-replace-all**$(currentTheorem_{lhs}, ?sort, t)$
if **fs-top**$(currentTheorem_{lhs}) \in \{\leq, \leq_{nl}, \leq_{ln}\}$
then $newTheorem_{rhs} =$ **var-connect**$(newTheorem_{lhs}, varGen, \{\leq, \leq_{nl}, \leq_{ln}\})$
else $newTheorem_{rhs} = and($ **var-connect**$(newTheorem_{lhs}, varGen, \{\leq, \leq_{nl}, \leq_{ln}\}),$
  $and_i^m($**fs-replace-all**$(currentTheorem_{lhs}, ?sort, l_i)))$
"New theorems are generated from the right hand side of $newTheorem$. If the subterms are more or less independent, the right hand side will be split up ($Terms$)."
if **fs-arity**(**fs-top**$(newTheorem_{rhs})) > 1$
  and **var-con-onep**$(newTheorem_{rhs}) = true$
then $Terms =$ **dir-subterms**$(newTheorem_{rhs})$
else $Terms = newTheorem_{rhs}$
$k = 1$
"Special lists ($TeList_j$) are constructed which help to undo the generalization."
while $varTeList_{k-1} \neq \emptyset$ do
  $varTeList_k = \emptyset$
  $TeList_k = \emptyset$
  For each $(l_i, t_i) \in varTeList_{k-1}$ do
    $t_i^* = t_i$
    For each $st_j \in$ **dir-subterms**(t_i) do
      if **varp**$(st_j) = false$
      then $var_j = var - gen$
        $t_i^* =$ **te-replace**(t_i^*, st_j, var_j)
        $varGen = varGen \cup \{var_j\}$
        $varTeList_k = varTeList_k \cup \{(var_j, st_j)\}$
    endfor
    $TeList_k = TeList_k \cup \{(l_i, t_i^*)\}$
  endfor
  $k = k + 1$
endwhile

Figure 5: Program of $Method_2$ - Part 1

"With the help of the $TeList_j$, new left hand sides are constructed. The new right hand sides are either $true$ or derived from the left hand sides by replacing certain interfering subterms (functions used solely for the definition of the sorting algorithm) by extracted variables."

```
l = 1
EQ = ∅
For j = 1 to k − 1 do
    For each  term ∈ Terms do
        term_rhs = term
        For each  (l_i, t_i) ∈ TeList_j do
            term_lhs = fs-replace-all(term, l_i, t_i)
            if    (te-var(fs-superterm(term, l_i)) \ l_i)
                  ∩ te-var(t_i) = ∅
            then term_rhs = fs-replace(term_rhs, l_i, var-extract(t_i))
            else  term_rhs = true
        endfor
        subterms = subterms(term_lhs)
        bool = true
        For each  st_i ∈ subterms do
            if    te-equal(st_i, currentTheorem_lhs) = true
            then bool = false
        endfor
        if    bool = true
        then EQ = EQ ∪ {(term_lhs, term_rhs)}
    endfor
endfor
"The new theorems are attached to the current planning tree."
nd-attach( currentNode,
           nd-new( newTheorem, n,
                   Method_2, none)
         )
For each  {(term_lhs, term_rhs)} ∈ EQ do
    newTheorem_lhs = term_lhs
    newTheorem_rhs = term_rhs
    nd-attach( currentNode,
               nd-new( newTheorem, n,
                       Method_2, none)
             )

endfor
```

Figure 6: Program of $Method_2$ - Part 2

$Method_4$	
Description	Replace disappearing subterm by appropriate if-then-else statement
Precondition	$Sensor_2 = true(?symb)$
Conflict information	$refinement(currentNode)$

Program	$help = currentTheorem_{lhs}$
	$i = 1$
	$stop = false$
	while $stop = false$ do
	    if
	    then $s_i =$ **fs-superterm**$(help, ?symb_i)$
	        $newTheorem^i_{lhs} =$
	            fs-replace$(s_i, ?symb_i, if(B, t_1, t_2))$
	        $tt_1 =$ **fs-replace**$(s_i, t_1, ?symb_i)$
	        $tt_2 =$ **fs-replace**$(s_i, t_2, ?symb_i)$
	        $newTheorem^i_{rhs} =$
	            $or(\ or(and(not(B), tt_2), and(B, tt_1)),$
	                $and(tt_1, tt_2))$
	        $l_i =$ **var-gen**
	        $help =$ **fs-replace**$(help, ?symb_i, l_i)$
	    else $stop = true$
	    $i = i + 1$
	endwhile
	For $j = 1$ to i - 1 do
	    nd-attach($currentNode$,
	            nd-new($newTheorem^j, n,$
	                $Method_4, verifiable)$
	            )
	endfor

Postcondition	$insufficient(currentNode)$ $verifiable(newNode)$
$endMethod_4$	

D Functions

fs-sort: $\langle fsymb \rangle \to \langle sort \rangle$
Returns the sort of the result of the function $\langle fsymb \rangle$

fs-contain: $\langle fsymb \rangle\ \langle term \rangle \to$ {true, false}
$true$, if the function $\langle fsymb \rangle$ occurs in $\langle term \rangle$
$false$, otherwise

fs-defined: $\langle fsymb \rangle \to$ {true, false}
$true$, if $\langle fsymb \rangle$ is a defining function
$false$, otherwise

fs-top: $\langle term \rangle \to \langle fsymb \rangle$
Returns the leading (outermost) function symbol

fs-rec-rhs: $\langle fsymb \rangle \to \langle term \rangle$
Returns the right hand side of the defining equation of the function $\langle fsymb \rangle$, that contains a recursive call of $\langle fsymb \rangle$

fs-arity: $\langle fsymb \rangle \to \mathbf{N}$
Returns the arity of the function $\langle fsymb \rangle$

fs-occur: $\langle fsymb \rangle\ \langle term \rangle \to \mathbf{N}$
Returns the number of occurrences of $\langle fsymb \rangle$ in $\langle term \rangle$

fs-subterm: $\langle term_1 \rangle\ \langle fsymb \rangle \to \langle term_2 \rangle$
Returns a subterm of $\langle term_1 \rangle$ with the leading function $\langle fsymb \rangle$

fs-superterm: $\langle term_1 \rangle\ \langle fsymb \rangle \to \langle term_2 \rangle$
Returns a subterm of $\langle term_1 \rangle$ of which a direct subterm possesses the leading function symbol $\langle fsymb \rangle$

fs-replace: $\langle term_1 \rangle\ \langle fsymb \rangle\ \langle term_2 \rangle \to \langle term_3 \rangle$
One subterm of $\langle term_1 \rangle$ with the leading function $\langle fsymb \rangle$ is replaced by $\langle term_2 \rangle$

fs-replace-all: $\langle term_1 \rangle$ $\langle fsymb \rangle$ $\langle term_2 \rangle \rightarrow \langle term_3 \rangle$
All subterms of $\langle term_1 \rangle$ with the leading function $\langle fsymb \rangle$ are replaced by $\langle term_2 \rangle$

varp: $\langle term \rangle \rightarrow$ {true, false}
true, if $\langle term \rangle$ is a variable
false, otherwise

var-glob-setp: $\langle fsymb \rangle \rightarrow$ {true, false}
true, if $\langle fsymb \rangle$ is a bound global variable
false, otherwise

var-gen: $\rightarrow \langle term \rangle$
Generates a new variable of appropriate sort

var-extract: $\langle term_1 \rangle \rightarrow \langle term_2 \rangle$
Returns the least nested variable of $\langle term_1 \rangle$ that has the same sort as the term itself

var-con-onep: $\langle term \rangle \rightarrow$ {true, false}
true, if each subterm of $\langle term \rangle$ shares at most one variable with any other subterm of $\langle term \rangle$
false, otherwise

var-connect: $\langle term_1 \rangle$ $\langle term - list \rangle$ $\langle fsymb - list \rangle \rightarrow \langle term_2 \rangle$
Returns a term of the form $and(and_i^k(\leq_i (x_i, n)), and_j^m(\leq_j (y_j, n)))$; n is the least nested variable of $\langle term_1 \rangle$ that does not occur in $\langle term - list \rangle$; the x_i describe the (differing) variables occurring in the flattened form of $\langle term_1 \rangle$ before n; the y_j describe the (differing) variables occurring in flattened form of $\langle term_1 \rangle$ before n; the $\leq_i$ and $\leq_j$ stand for the appropriate (sort considering) functions (all of which have two arguments) from $\langle fsymb - list \rangle$

dir-subterms: $\langle term \rangle \rightarrow \langle term - list \rangle$
Returns a list of all direct subterms of $\langle term \rangle$

subterms: $\langle term \rangle \rightarrow \langle term - list \rangle$
Returns a list of all subterms of $\langle term \rangle$

te-var: $\langle term \rangle \rightarrow \langle term - list \rangle$
Returns a list of all variables of $\langle term \rangle$

te-equalp: $\langle term_1 \rangle \ \langle term_2 \rangle \rightarrow$ {true, false}
$true$, if $\langle term_1 \rangle$ and $\langle term_2 \rangle$ are equal except for variable-names
$false$, otherwise

te-replace: $\langle term_1 \rangle \ \langle term_2 \rangle \ \langle term_3 \rangle \rightarrow \langle term_3 \rangle$
The subterm $\langle term_2 \rangle$ of $\langle term_1 \rangle$ is replaced by $\langle term_3 \rangle$

nd-new: $\langle equation \rangle$ {n, v, u, r} $\langle name \rangle \ \langle info \rangle \rightarrow \langle node \rangle$
Returns a node of a planning tree

nd-attach: $\langle node \rangle$
Attaches $\langle node \rangle$ as a son to $currentNode$ of the current planning tree

E Verified planning trees

This section presents the planning trees for the examples of Insertion Sort, Merge Sort and Quick Sort in a reduced form. Each node is represented by its respective theorem, only. Note that the statements on natural numbers neglected in the description of section 2.3 have been added for completeness sake (marked by dotted lines).

Insertion Sort

$ord(isort(l)) = true$

- $ord(ins(n,l)) = ord(l)$
  - $\leq_{nl}(n, ins(m,l)) = and(\leq(n,m), \leq_{nl}(n,l))$
    - $\leq_{nl}(n, if-list(B,l1,l2)) = \text{or}(\ or(and(not(B), \leq_{nl}(n,l2)), and(B, \leq_{nl}(n,l1))), and(\leq_{nl}(n,l1), \leq_{nl}(n,l2)))$
  - $ord(if-list(B,l1,l2)) = \text{or}(\ or(and(not(B), ord(l2)), and(B, ord(l1))), and(ord(l1), ord(l2)))$
  - $and(\leq(n,m), \leq(m,n)) = eqn(n,m)$
  - $and(not(\leq(n,m)), \leq(m,n)) = not(\leq(n,m))$
  - $and(\leq(n,m), and(\leq_{nl}(n,l), \leq_{nl}(m,l))) = and(\leq(n,m), \leq_{nl}(m,l))$

Merge Sort

$$ord(msort(l)) = true$$

$$ord(merge(l1, l2)) = and(ord(l1), ord(l2))$$

$$\leq_{nl} (n, merge(l1, l2)) = and(\leq_{nl} (n, l1), \leq_{nl} (n, l2))$$

$$\begin{aligned}\leq_{nl} (n, if - list(B, l1, l2)) = \mathrm{or}(\ & or(and(not(B), \leq_{nl} (n, l2)), and(B, \leq_{nl} (n, l1))), \\ & and(\leq_{nl} (n, l1), \leq_{nl} (n, l2)))\end{aligned}$$

$$\begin{aligned}ord(if - list(B, l1, l2)) = \mathrm{or}(\ & or(and(not(B), ord(l2)), and(B, ord(l1))), \\ & and(ord(l1), ord(l2)))\end{aligned}$$

$$and(not(\leq (n, m)), and(\leq_{nl} (n, l), \leq_{nl} (m, l))) = and(not(\leq (n, m)), \leq_{nl} (n, l))$$

$$and(\leq (n, m), and(\leq (q, m), not(\leq (q, n)))) = and(\leq (q, m), not(\leq (q, n)))$$

$$\leq (n, 0) = eq(n, 0)$$

$$eq(n, n) = true$$

$$and(\leq (n, m), \leq (m, n)) = eqn(n, m)$$

$$and(not(\leq (n, m)), \leq (m, n)) = not(\leq (n, m))$$

$$and(\leq (n, m), and(\leq_{nl} (n, l), \leq_{nl} (m, l))) = and(\leq (n, m), \leq_{nl} (m, l))$$

Quick Sort

$$ord(qsort(l)) = true$$

$$ord(app(l1, cons(n, l2))) = and(ord(l1), ord(l2), \leq_{ln} (l1, n), \leq_{nl} (n, l2))$$

$$\leq_{nl} (n, app(l1, cons(m, l2))) = and(\leq (n, m), and(\leq_{nl} (n, l1), \leq_{nl} (n, l2)))$$

$$and(\leq (n, m), and(\leq_{nl} (n, l), \leq_{nl} (y, l))) = and(\leq (n, m), \leq_{nl} (m, l))$$

$$\leq_{ln} (qsort(l), n) = \leq_{ln} (l, n)$$

$$\leq_{ln} (app(l1, cons(n, l2)), m) = and(\leq (n, m), and(\leq_{ln} (l1, m), \leq_{ln} (l2, m)))$$

$$and(\leq (n, m), and(\leq_{ln} (lower(n, l), m), \leq_{ln} (greater(n, l), m))) = and(\leq (n, m), \leq_{ln} (l, m))$$

$$\leq_{ln} (if-list(B, l1, l2), n) = or(or(and(not(B), \leq_{ln} (n, l2)), and(B, \leq_{ln} (n, l1))), and(\leq_{ln} (n, l1), \leq_{ln} (n, l2)))$$

$$\leq (n, 0) = eq(n, 0)$$

$$eq(n, n) = true$$

$$\leq_{nl} (n, qsort(l)) = \leq_{nl} (n, l)$$

$$\leq_{nl} (n, app(l1, cons(m, l2))) = and(\leq (n, m), and(\leq_{nl} (n, l1), \leq_{nl} (n, l2)))$$

$$and(\leq (n, m), and(\leq_{nl} (m, lower(n, l)), \leq_{nl} (m, greater(n, l)))) = and(\leq (n, m), \leq_{nl} (m, l))$$

$$\leq_{nl} (n, if-list(B, l1, l2)) = or(or(and(not(B), \leq_{nl} (n, l2)), and(B, \leq_{nl} (n, l1))), and(\leq_{nl} (n, l1), \leq_{nl} (n, l2)))$$

$$\leq (n, 0) = eq(n, 0)$$

$$eq(n, n) = true$$

$$\leq_{ln} (lower(n, l), n) = true$$

$$\leq_{ln} (if-list(B, l1, l2), n) = or(or(and(not(B), \leq_{ln} (l2, n)), and(B, \leq_{ln} (l1, n))), and(\leq_{ln} (l1, n), \leq_{ln} (l2, n)))$$

$$\leq_{nl} (n, greater(n, l)) = true$$

$$\leq_{nl} (n, if-list(B, l1, l2)) = or(or(and(not(B), \leq_{nl} (n, l2)), and(B, \leq_{nl} (n, l1))), and(\leq_{nl} (n, l1), \leq_{nl} (n, l2)))$$

$$and(not(\leq (n, m)), \leq (m, n)) = not(\leq (n, m))$$

References

References

[Ave90] Jürgen Avenhaus. Reduktionssysteme 2. Vorlesungsskript, Universität Kaiserslautern, Fachbereich Informatik, Postfach 3049, D-6750 Kaiserslautern, 1990.

[Ave91] Jürgen Avenhaus. Reduktionssysteme 1. Vorlesungsskript, Universität Kaiserslautern, Fachbereich Informatik, Postfach 3049, D-6750 Kaiserslautern, 1991.

[Bac87] L. Bachmair. *Proof Methods for Equational Theories*. PhD thesis, Dept. of Computer Science, University of Illinois, Urbana, 1987.

[Bac88] L. Bachmair. Proof by consistency in equational theories. *Proc. of LICS*, pages 228–233, 1988.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.

[BM88] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[Bun88] Alan Bundy. The use of explicit plans to guide inductive proofs. *Proc. of 9th International Conference on Automated Deduction*, 1988.

[DM89] Jörg Denzinger and Jürgen Müller. Eqtheopogles - a completion theorem prover for pl1eq. In D. Metzing, editor, *GWAI-89, 13th German Workshop on Artificial Intelligence*, volume 216 of *Informatik-Fachberichte*, pages 92–101. Springer-Verlag, 1989.

[GL91] Bernhard Gramlich and Wolfgang Lindner. A guide to unicom, an inductive theorem prover based on rewriting and completion techniques. SEKI-Report SR-91-17(SFB), Universität Kaiserlsautern, Fachbereich Informatik, Postfach 3049, D-6750 Kaiserslautern, December 1991.

[Gra90] Bernhard Gramlich. Completion based inductive theorem proving: A case study in verifiying sorting algorithms. SEKI-Report SR-90-04, Universität Kaiserlsautern, 1990.

[HHRS86] R. Hähnle, M. Heisel, W. Reif, and W. Stephan. An interactive verification system based on dynamic logic. In *Proc. 8th CADE*, volume LNCS 230, pages 306–315. Springer, 1986.

[HKK92] Xiarong Huang, Manfred Kerber, and Michael Kohlhase. Methods - the basic units for planning and verifying proofs. *Submitted to IJCAI93*, 1992.

[New81] A. Newell. The heuristic of george polya and its relation to artificial intelligence. Technical Report CMU-CS-81-133, Carnegie-Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania, USA, 1981.