Proposing a Task Interface

Claus-Peter Wirth,
Serge Autexier, Christoph Benzmüller, Andreas Meier, &al.
Dept. of Computer Sci., Universität des Saarlandes, D–66123 Saarbrücken, Germany
`cp@ags.uni-sb.de`

May 20, 2004

Abstract

This draft is for internal communication and not completely self-contained.

1 Motivation

A careful and objective inspector of the history of automated theorem proving in the last fifty years would come to the following hypothesis, which is also the private opinion of the leading developers and scientists in the field of automated theorem provers today:

> Stand-alone automated theorem provers will never develop into practically useful mathematical assistant systems.

To achieve the original design goal of a practically useful mathematical assistant system nevertheless, we aim at *interactive* systems with a high degree of automated support. To combine interaction and automation into a synergetic interplay is an enormous task. It requires sophisticated achievements from logic, tactics programming, proof planning, agent-based approaches from artificial intelligence, graphical user interfaces, and bus connection to other reasoning tools on the one hand, and a deeper experience in informal and formal human proof construction on the other hand. Obviously, the development of a state-of-the-art mathematical assistant system is a huge enterprise, requiring expertise from many different fields.

In this paper we suggest to cut this huge enterprise in twain. The resulting interface is called the *task interface*. Roughly speaking, it is to separate the heuristic part of a theorem prover from its logic engine. The heuristic part may consist of user interaction, tactics, and proof planning. The logic engine is the part that executes proof steps and reports and keeps track of the soundness conditions of these steps. Thus, tacticals, proof planning, and human interaction are to communicate with the proof construction only via this interface.

Such an interface comes with all the typical pros and cons regarding information hiding, reusability, &c., which do not have to be mentioned here in more detail.

The crucial consequence, however, is that the heuristic part cannot be used to compensate for the insufficiencies of the logic engine, which is the *status quo* especially of the more useful theorem provers. Indeed, the concepts of the task interface must be free from the peculiarities that occur only in inference systems but not in the minds of Bourbakian mathematicians.

Instead of what is captured by buzzwords like bottom-up or top-down reasoning, a primarily semantical point of view has to be taken. With a clear understanding of the semantics of the logic as well as the data structures and operations of the task interface, it should be possible to design and implement the heuristic part of a theorem prover without any deeper knowledge of the calculi of the logic engine. This heuristic part operates on the primitives that are provided by the task interface and not on the logic level.

The task interface as proposed in this paper consists of a data structure called *task forests* and a set of *actions* modifying a task forest. In principle, a set of modifying operations and look-up functions would be sufficient, but as the orientation in the task forest would require many look-up functions and as the heuristic programs must be able to hash new attributes to the nodes of the task forest, on the abstract level of design chosen for this paper it is more natural to describe the structure of a task forest explicitly and then explain how the actions modify it.

2 Task Forests and Closedness

2.1 Tasks

A *task* can be seen as a proof obligation, a proof line with logical context, a sequent. For an example, consider a lemma we want to prove from a set of assumptions. A task may contain free variables to be instantiated (or holes to be filled) during the proof construction appropriately. It is only via these free variables that a task is related to or interacts with other parallel tasks. Thus, a task can be seen as a somewhat independent proof goal. Without loss of generality, we can become more concrete here:

Definition 2.1 (Task)
A *task* is a list of $(+/-)$ signed formulas, possibly augmented with some further syntactical structure.

For example, a typical augmentation is a weight term guaranteeing wellfoundedness of the application of tasks as induction hypotheses. The positive sign $(+)$ may be dropped in examples. Note that we leave open what a "formula" is. Notice that a task consists of a *list* and not a *set* of signed formulas. The order of the signed formulas is relevant for the operational behavior of the task. Especially the first signed formula A_0 in a non-empty task $A_0, A_1, \ldots, A_n$ is emphasized. In Hübner &al. (2003) it is called the *goal window* of the task and the task is written as $A_1, \ldots, A_n \rhd A_0$.

The semantics of a task is given by the semantics of the formula of the task:

Definition 2.2 (Formula of a Task)
The *formula of the task* $A_1, \ldots, A_n$ is $(\mathrm{Unsign}(A_1) \vee \ldots \vee \mathrm{Unsign}(A_n))$, where Unsign returns the (unsigned) formula of a signed formula and prefixes it with $\neg$ in case of negative sign.

Obviously, this definition is only appropriate for two-valued logics, to which we restrict ourselves here.

2.2 Actions, Reductions, Closedness, and Proofs

To achieve a task, we have to do something, i.e. we have to perform *actions*. Such an action *reduces* the original task to a new set T of other tasks. If T is empty, then we have achieved the task, and we say that the task is *closed*.

Just as tasks are structured into proof attempts by recursive application of reduction to subtasks, multiple proof attempts result from multiple reduction of the same task. Multiple proof attempts, reductions, and tasks can be considered as a hierarchy:

level	subject	subunits	connection of subunits	reason for the modus of the logical connection
1st	**Multiple Proof**	parallel reductions	disjunctive	area of application
2nd	**Reduction**	subtasks	conjunctive	disjunctive normal form together with level 1
3rd	**Task**	signed formulas	disjunctive	conjunctive normal form together with level 2

Contrary to forms of political or juristic argumentation where total evidence is the sum of the evidences of alternative "proofs", in our area of application it typically suffices to establish a task only once, simply because a second proof does not give more evidence than a single one. Although alternative ways to achieve a task are useful during proof construction and interesting when considering alternative axiomatizations, most of the time we would be lucky if we found a single way to achieve a task. As multiple mathematical proofs (1st level) are thus connected disjunctively, it is advantageous to connect the subtasks resulting from a reduction (2nd level) conjunctively, because the two levels together further the normalization of proof constructions to disjunctive normal form.

This normal form of proof construction is not obligatory, but it furthers a certain form or style. On the one hand, this style helps human beings to understand foreign proofs and maintain own ones, and, on the other hand, makes it easier for automatic proof heuristics to recognize the triggering structures and applicable lemmas.

The conclusion is that a set T of subtasks resulting from a reduction should be considered to be conjunctive by default, in the sense that we have to achieve *all* of its tasks to achieve the original task.

Note that, due to the (default of a) conjunctive connection of the subtasks resulting from a reduction step (2nd level), it is advantageous to connect formulas inside a task (3rd level) disjunctively, because this furthers normalization of proof constructions to conjunctive normal form.

We do not know of the dual (and thus isomorphic) choice of a conjunctive instead of a disjunctive connection of the formulas of a task in the literature. By tradition, both in informal human mathematical practice (starting form Aristotle's syllogisms and ending with lemmas in a modern textbook) and in formal logic calculi (Hilbert, resolution, natural deduction, tableau, sequent, and matrix calculi), tasks have a disjunctive structure.

2.3 Task Forests

Think tasks in a proof to be nodes in a graph. When we want to consider an alternative proof for a task, we must be able to do this without changing or affecting our previous constructions. Thus, the tasks themselves have to reign the reduction steps disjunctively, i.e. the *task nodes* have to be OR nodes w.r.t. the actions reducing them. Due to the conjunctive connection of the subtasks generated by an action application, it is appropriate to take the actions as AND nodes in a bipartite AND-OR graph.

Nevertheless, to be more flexible, besides such *AND actions*, we admit *OR actions*, i.e. actions whose set of subtasks is connected disjunctively. They do not complicate the matter and the resulting structural flexibility may admit a more straightforward representation of translations from other formalisms.

Below, we list several reasons to consider this graph as a *task forest* of trees. For technical reasons (lemma applications global to a tree), the root of such a *task tree* is the AND node of the action that started this tree. This AND action node has exactly one task node among its children. It is called the *root task node* and the task labeling it is called the *root task* of the tree. Vice versa, the tree is called a *tree for this task*.

Distinction of Tasks: Some tasks are more important than others for one of the following reasons: Some belong to the input set describing our problem, others are the result of a creative generalization, others are just in the center of our interest. Human users, planners, &c. should have the possibility to distinguish a task by starting a new task tree for it.

Restriction of the Applicability of Tasks: For the sake of human-oriented lucidity and computational efficiency, only the root tasks should be applicable as lemmas and induction hypotheses. The sometimes occurring practical necessity of applying a task of an inner task node of a tree as a lemma can still be satisfied as follows: We first split the tree in twain above this task node—thereby turning it into a root node—and then connect the two trees with a reference for lemma application (*LEM*).

Safeness: On the one hand, all reduction steps must be *sound* in the sense that we expect the original task to be solvable when we are able to close the tasks it is reduced to. On the other hand, it is not always the case that reduction steps are *safe* in the sense that the original task is unsolvable when one of the tasks it is reduced to is unsolvable.

For example, if we reduce the task of $x_0^{\delta^-}{<}y^\gamma{<}x_1^{\delta^-},\ x_1^{\delta^-}{\le}x_0^{\delta^-},\ \mathsf{s}(x_0^{\delta^-}){=}x_1^{\delta^-}$ to the task $x_0^{\delta^-}{<}y^\gamma{<}x_1^{\delta^-},\ x_1^{\delta^-}{\le}x_0^{\delta^-}$, which asks us to find a natural number y^γ in between the natural numbers $x_0^{\delta^-}$ and $x_1^{\delta^-}$ unless $x_1^{\delta^-}{\le}x_0^{\delta^-}$, then the reduction is sound but unsafe, because the solution $\{y^{\delta^-}{\mapsto}\mathsf{s}(x_0^{\delta^-})\}$ gets lost in case of $\mathsf{s}(x_0^{\delta^-}){=}x_1^{\delta^-}$. Note that the free δ^--variables are parameters and the free γ-variables are meta or query variables.

The property of *safeness* is useful for backtracking after failure detection. A good way to distinguish unsafe reductions from safe reduction steps is to require all reductions within a tree to be safe. Thus, if we want to strengthen a task, we start a new tree and connect it to the tree of the previous task with a reference for lemma application.

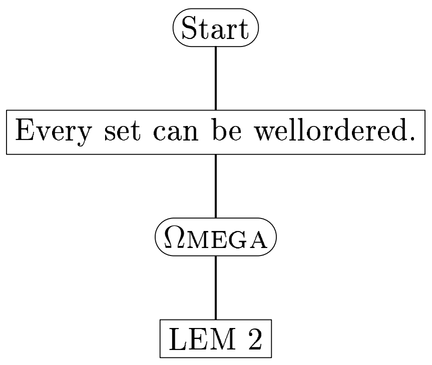
Breaking Cycles: If we reduce the task of showing the Wellordering Theorem to Zorn's Lemma and the task of showing Zorn's Lemma to the Wellordering Theorem, we know that the two are logically equivalent, but we have not closed any of our tasks. It seems that these cycles in our graph do not really have to bother us. Indeed, when we define closedness *inductively*, it never enters the cycles of our proof graph and soundness is maintained. Besides deduction, we also want to model mathematical induction in the form of Fermat's *descente infinie*, and this, however, is a form of *cyclic* reasoning, cf. Wirth (2004a). For example, if we reduce the task of showing a property P on the natural numbers to properties P and Q on smaller numbers and do the same for Q, then we have closed the tasks of showing P and Q on the natural numbers. To compute closedness in the presence of applications of induction hypotheses (*HYP*) in addition to lemmas (*LEM*), we have to find a small set of task nodes such that any cycle in the graph contains a node of this set. Picking the root task node from each tree we get such a set of cycle breaking nodes.

Definition 2.3 (Task Tree)
A *task tree* is a labeled rooted directed tree.
Its nodes are partitioned into *task*, *action*, and *reference* nodes. The action nodes are again partitioned into *AND* and *OR* nodes. The reference nodes into *LEM* and *HYP* nodes.
Each node has exactly one label. A task node is labeled with a task, an action node with an action, and a reference node with a positive natural number (referring to the tree with that number, cf. Definition 2.4).
All child nodes of task nodes are action nodes. All child nodes of action nodes are task or reference nodes. All reference nodes are leaf nodes.
The root of a task tree t is an AND action node, which has among its children exactly one task node. This task node is called the *root task node of* t and its label Φ is called the *task of* t. Vice versa, the tree t is then called a *tree for* Φ.

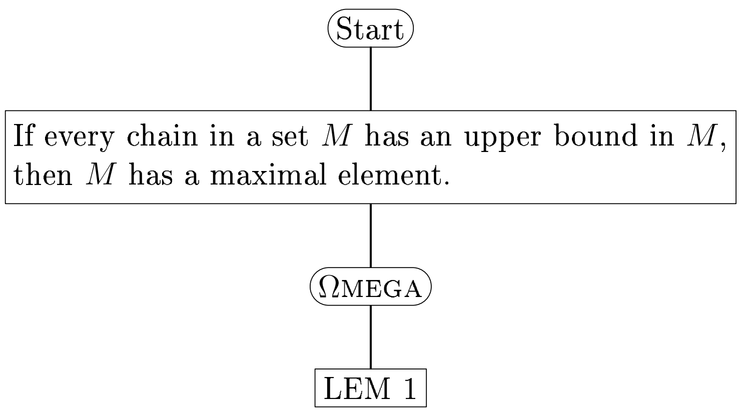
Definition 2.4 (Task Forest)
A *task forest* is a partial function F from the set of positive natural numbers $\mathbf{N}_+$ into the set of those task trees whose LEM and HYP nodes are labeled with natural numbers that are in the domain of F.
For n in the domain of F, we call $F(n)$ the *task tree* n of F, or simply *tree* n.

Example 2.5 (Lemma Application: Wellordering-Theorem and Zorn's Lemma)
Consider the following task forest consisting of tree 1 and 2:

1:



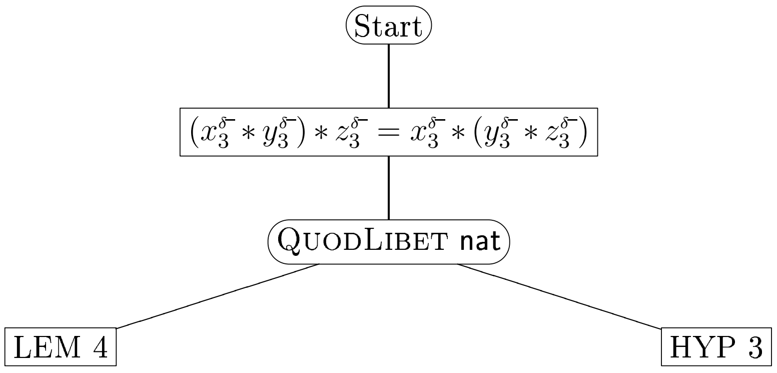
2:


Start
If every chain in a set M has an upper bound in M, then M has a maximal element.
ΩMEGA
LEM 1

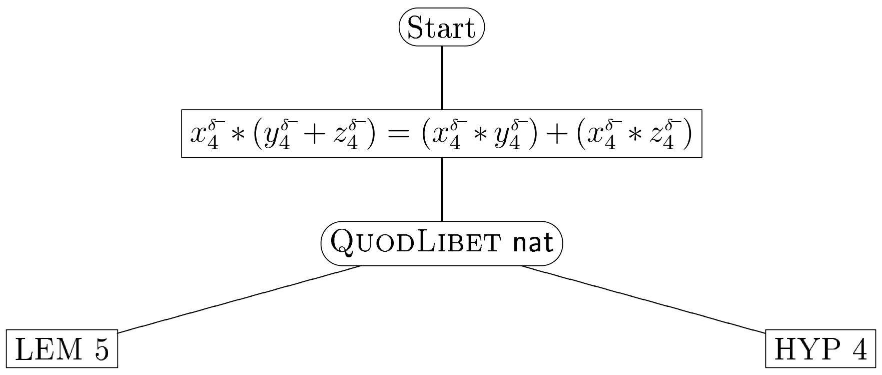
Besides the technical AND action "Start" (which has the function to accumulate lemma applications that are global to the task tree), for simplicity we have only one single AND action "ΩMEGA" which means to call the ΩMEGA theorem prover, cf. Siekmann &al. (2002). Note that the lemma application relation is cyclic and none of the tasks is closed.

Example 2.6 (Associativity of Multiplication on the Natural Numbers)
Consider the following task forest consisting of tree 3, 4, and 5:

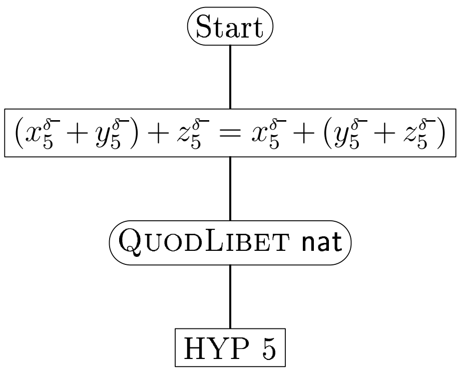
3:


Start
$(x_3^{\delta^-} * y_3^{\delta^-}) * z_3^{\delta^-} = x_3^{\delta^-} * (y_3^{\delta^-} * z_3^{\delta^-})$
QUODLIBET nat
LEM 4
HYP 3

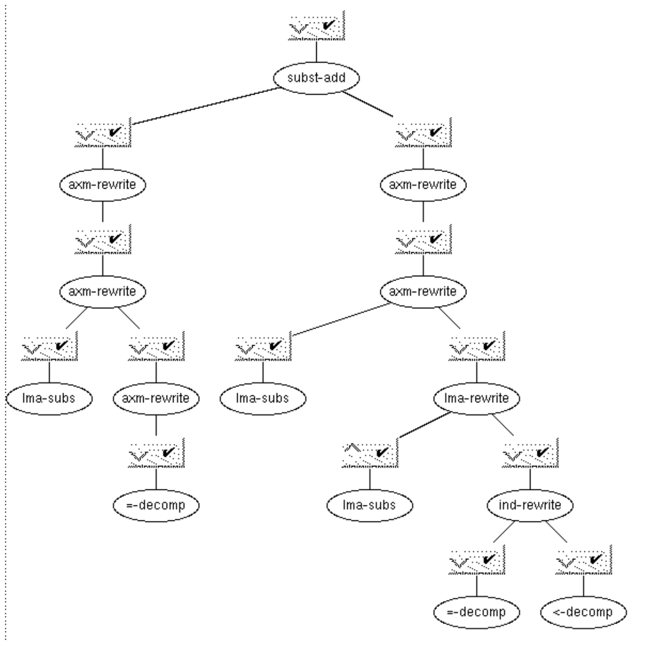
4:

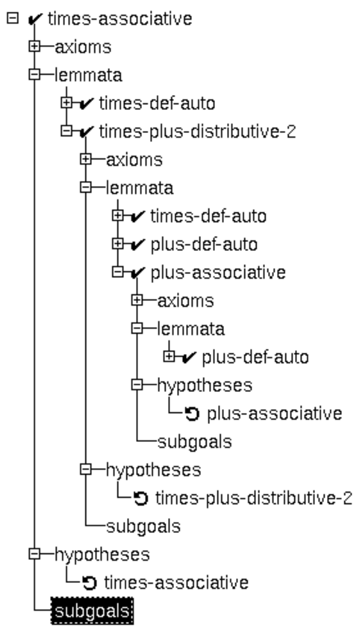

Start
$x_4^{\delta^-} * (y_4^{\delta^-} + z_4^{\delta^-}) = (x_4^{\delta^-} * y_4^{\delta^-}) + (x_4^{\delta^-} * z_4^{\delta^-})$
QUODLIBET nat
LEM 5
HYP 4

5:


Start
$(x_5^{\delta^-} + y_5^{\delta^-}) + z_5^{\delta^-} = x_5^{\delta^-} + (y_5^{\delta^-} + z_5^{\delta^-})$
QuodLibet nat
HYP 5

The AND action “QuodLibet nat” means to call the QuodLibet theorem prover in the theory of the natural numbers with the lemmas and induction hypotheses given below, cf. Avenhaus &al. (2003). Thus, tree 3 says that we can prove the associativity of multiplication inductively (with itself (HYP 3) as induction hypothesis) using the distributivity of multiplication over addition as a lemma (LEM 4). Again we can prove this distributivity inductively (HYP 4) with the associativity of addition as a lemma (LEM 5), which can be proved by a simple induction (HYP 5). This is of course a very abstract view, showing no subtasks at all. The task tree presented by QuodLibet for the associativity of multiplication instead of our task tree 3 has more details, but does not display the “Start” action:





```
⊟ ✔ times-associative
  ⊞─axioms
  ⊟─lemmata
    ⊞✔ times-def-auto
    ⊟✔ times-plus-distributive-2
      ⊞─axioms
      ⊟─lemmata
        ⊞✔ times-def-auto
        ⊞✔ plus-def-auto
        ⊟✔ plus-associative
          ⊞─axioms
          ⊟─lemmata
            ⊞✔ plus-def-auto
          ⊟─hypotheses
            └↻ plus-associative
          └subgoals
      ⊟─hypotheses
        └↻ times-plus-distributive-2
      └subgoals
  ⊟─hypotheses
    └↻ times-associative
  └subgoals
```

Note that the tree to the left only shows the task and action nodes. The reference nodes are displayed to the right where we see the whole dependency graph starting with this tree in a different representation. More precisely, the “lma-rewrite” and “ind-rewrite” actions in the rightmost branch of left tree correspond to the application of LEM 4 and HYP 3 in the more abstract tree 3; the three “lma-subs” actions use subsumption with the lemmas “times-def-auto” and “plus-def-auto”, which are automatically generated and proved and which state the total definedness of multiplication and addition on the natural numbers. This is necessary because QuodLibet admits the specification of partial recursive functions.

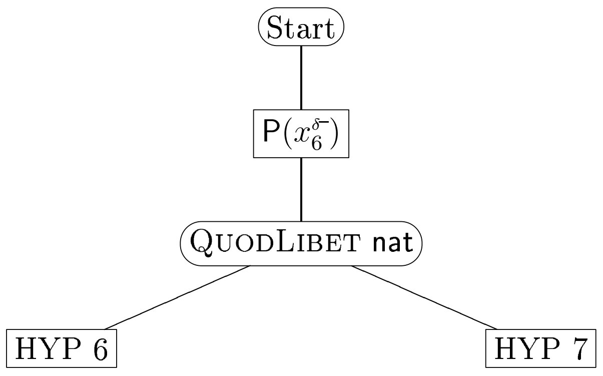
For capturing the maximal set of trees that may influence tree n in a task forest in a single induction loop (i.e. a sequence of possibly mutual applications of induction hypotheses that is not interrupted by a lemma application) we define:

Definition 2.7 (HYP Grove)
The n^{th} *HYP grove* of a given task forest is inductively defined as a set of task trees as follows: It includes the tree n. If a tree in it has a HYP node labeled with m, it includes the tree m, too.

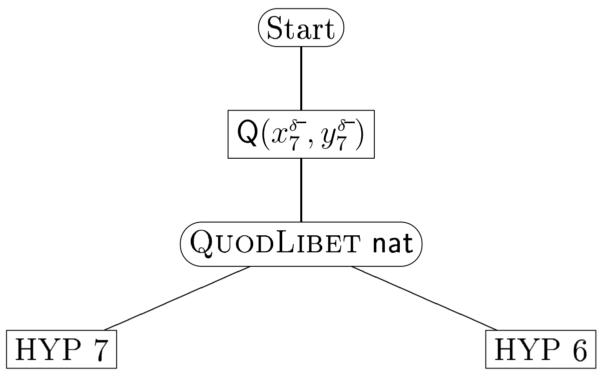
In Example 2.6, for $i \in \{3,4,5\}$, the i^{th} HYP groove consists only the tree i. This is not generally the case:

Example 2.8 (P **and** Q)
The abstract proof trees for the universal validity of the predicates P and Q (for details, cf. Wirth (2004a), Section 3.2.2) look as follows:

6:



7:



This is obviously an example of a mutual induction and the 6th HYP grove is equal to the 7th HYP grove and consists of trees 6 and 7.

2.4 Closedness

The *closedness of task tree $n \in \mathbf{N}_+$ of a task forest F* satisfies the following axioms:

> Task tree n is closed if and only if the root node of $F(n)$ is closed.
> A task node is closed if some of its child nodes are closed. An AND action node is closed if all of its child nodes are closed. An OR action node is closed if some of its child nodes are closed. A reference node with label n is closed if task tree n is closed.

If no HYP nodes occur in the forest, closedness is the property inductively defined by these axioms.

(Beware: We get the property of *deductive* closedness when we read the axioms inductively, taking the smallest fixed point. For the property of inductive closedness we have to look for another fixed point of the axioms, but not for the greatest. This should become clear from Example 2.5, where Zorn's Lemma and the Wellfoundedness-Theorem must not become closed as valid theorems just because they are equivalent to each other.)

For the general case that admits HYP nodes, there are different and quite complex ways to define closedness. The following algorithm will do.

Definition 2.9 (Closedness)
A node of a task forest is *closed* if it is labeled with true by the following algorithm, assuming that the nodes of the task forest have no labels true or false initially:

Deductive Closedness
This is a simple bottom-up inductive AND-OR graph true/false-labeling: Label the trees with true and false such that all leaf nodes that are task or OR action nodes are labeled with false, all leaf nodes that are AND action nodes are labeled with true, all task and OR action nodes that have a child node labeled with true are labeled with true, and all AND action nodes that have a child node labeled with false are labeled with false. This can be performed in a bottom-up way, where we have to stop when

Too Late: a parent has already a label true or false (in this case we came up an irrelevant branch) or when

Too Early: a *relevant* sibling has no label.

Anytime a root node of tree say n is labeled with b, label all HYP and LEM nodes referring to n with b.

If all HYP reference nodes are labeled with true, stop.

Inductive Closedness

- Label all unlabeled LEM nodes with false′. Taking false′ temporarily for false, label the trees with false′ bottom-up, stopping when

  Too Late: a parent has already a label true, false or false′ (in this case we came up an irrelevant branch) or when

  Too Early: a *relevant* sibling has no label.

  Anytime a root node of tree say n is labeled with b, label all HYP nodes referring to n with b. Note that this label b can only be false′.

- Label all root nodes that have no label with true. Do the same with the LEM and HYP nodes of the same number. If there are no such root nodes, stop; otherwise go to *Deductive Closedness*.

The root nodes labeled in the last step form a set of induction hypotheses for each other, simply because any of their trees contains an unlabeled HYP node that is relevant for labeling the tree's root to true. Thus if we take all of them, the induction will definitely go through.

The Deductive plus Inductive Closedness can be computed in linear time. The bad news is that we have to repeat this closedness procedure until no new root nodes are labeled with true in the last step. This is because these nodes may have LEM nodes of the same number, which close other trees deductively, which close other trees inductively, and so on. So we end up with quadratic time complexity. Finally, since the false′-labeling of a proof tree by the Inductive Closedness procedure can only be triggered by the original false′-labeling of the LEM nodes in the HYP grove of that tree, there are several ways to improve the average time. More importantly, roughly speaking, we can start this algorithm locally when adding new child nodes with costs being low unless the proof construction step is crucial. Precisely speaking, this locality refers to the single new child node and its bottom-up consequences, unless this child node happens to be a HYP node that is relevant to its root node. In this case we have to start the Inductive Closedness procedure on the HYP groove of that tree.

Example 2.10 *(continuing Example 2.6)*
When we compute the closedness of the nodes of the task forest of Example 2.6, Deductive Closedness first labels no nodes. Then Inductive Closedness labels all nodes of the trees 3 and 4 with false′, with the exception of the HYP nodes. Consequently the root of tree 5 is labeled with true, and the same is the case for the HYP node of tree 5 and the LEM node of tree 4.
In the next round Deductive Closedness labels the remaining nodes of tree 5 with true. Then Inductive Closedness labels the LEM node of tree 3 with false′, and consequently does the same with all nodes of that tree except the HYP node. Then the root of tree 4 is labeled with true, just like the LEM node of tree 3 and the HYP node of tree 4.
In the next round Deductive Closedness labels the remaining nodes of tree 4 with true.

Then Inductive Closedness labels no nodes with false'. Then the root of tree 3 is labeled with true, just like the HYP node of tree 3.
In the next round Deductive Closedness labels the remaining nodes of tree 3 with true and then the algorithm stops because all HYP nodes are labeled with true.

Example 2.11 *(continuing Example 2.8)*
When we compute the closedness of the nodes of the task forest of Example 2.8, Deductive Closedness and Inductive Closedness are idle in the first round. Thus, all root and HYP nodes are labeled with true. Finally Deductive Closedness labels the remaining nodes with true.

In Kühler (2000) we can find an alternative definition of closedness. It proceeds local to HYP groves by fixing a single alternative for each OR node, resulting in "partial proof attempts", cf. Kühler (2000), Definition 6.1.4. When we always take the alternative that results in the labeling of an OR node with true in the above algorithm, we easily see that both definitions of closedness are equivalent.

3 Free Variable Administration

3.1 Introduction

As already indicated in Section 2.1, tasks are not completely independent because they share common free variables. Since the actions that will be described in Section 4 depend on these free variables, we have to introduce them here briefly; cf. Wirth (2004a), Wirth (2002) for a more detailed treatment of free variables.

To avoid the problem of binders capturing free variables and in the tradition of Gentzen (1935), Hilbert & Bernays (1968/70), and Snyder & Gallier (1989), we assume the following four sets of symbols to be mutually disjoint:

V_γ	*free γ-variables*, i.e. the free variables of Fitting (1996)
V_δ	*free δ-variables*, i.e. nullary parameters, instead of Skolem functions
$\mathrm{V}_{\mathrm{bound}}$	*bound variables*, i.e. variables occurring only bound
Σ	*constants*, i.e. the function (and predicate) symbols from the signature

Note that the 'γ' and 'δ' are the symbols of the classification of inference rules into α-, β-, γ-, and δ-rules of Smullyan (1968). We partition the free δ-variables V_δ into *free δ^--variables* V_{δ^-} that are introduced by the (non-liberalized) δ-rules; and *free δ^+-variables* V_{δ^+} that are introduced by the liberalized δ-rules (δ^+-rules):

$$\mathrm{V}_\delta = \mathrm{V}_{\delta^-} \uplus \mathrm{V}_{\delta^+}$$

We define the *free variables* by

$$\mathrm{V}_{\mathrm{free}} := \mathrm{V}_\gamma \uplus \mathrm{V}_\delta$$

and the *variables* by

$$\mathrm{V} := \mathrm{V}_{\mathrm{bound}} \uplus \mathrm{V}_{\mathrm{free}}$$

Finally:

$$\mathrm{V}_{\gamma\delta^+} := \mathrm{V}_\gamma \uplus \mathrm{V}_{\delta^+}$$

We use '$\mathcal{V}_k(\Gamma)$' to denote the set of variables from V_k occurring in Γ.

The functionality of variables is that the can be replaced with terms. On the one hand, free γ- and δ^+-variables must be replaced globally. This *rigidity* introduces goal conflicts between the tasks and thereby complicates matters considerably, but there is no practical way to proceed without them because this would make a *natural flow of information* impossible, cf. Wirth (2004a), Section 1.2.1, II.1. On the other hand, free δ^--variables are replaced locally when a lemma or an induction hypothesis is applied. While—in these applicative steps—free δ^--variables function just as the free variables in a Hilbert calculus, they behave as constant unknowns in reductive proof steps: When a task is reduced to a set of subtasks, the free δ^--variables cannot be replaced with anything because they have to work as parameters, eigenvariables, or Skolem symbols in a sequent, tableau, or matrix calculus.

3.2 Variable-Conditions and Choice-Conditions

When we look at the possible terms that can be substituted for a free variable in more detail, we find that they are not only restricted by the signature and the type of the variable, but also by certain conditions capturing the dependency of the free variables among each other, just as the order in a sequence of existential and universal quantifiers introduces a dependency between bound γ- and δ-variables. We capture this dependency in a directed graph which is global to the task forest and whose nodes are the free variables themselves. The overall idea is that an edge (x, y) occurring in this graph means something like "x is necessarily older than y" or "the value of y depends on or is described in terms of x". The set of edges of such a graph is called a "variable-condition":

Definition 3.1 (Variable-Condition)
A *variable-condition* is a subset of $\mathrm{V}_{\mathrm{free}} \times \mathrm{V}_{\mathrm{free}}$.

The restriction on the substitution of free δ^--variables will be treated in the applicative rules in Section 4.

The restriction on the replacements of free γ- and δ^+-variables is that they have to be consistent with the global variable-condition R of the task forest. And the replacements for the δ^+-variables additionally have to be compatible with a *choice-condition* C, which is also global to the task forest and whose semantics is given by Q_C as defined below.

Definition 3.2 (Choice-Condition)
C is an *R-choice-condition* if R is a wellfounded variable-condition, C is a partial function from V_{δ^+} into the set of formulas, and $z \; R^* \; y^{\delta^+}$ for all $y^{\delta^+} \in \mathrm{dom}(C)$ and $z \in \mathcal{V}_{\mathrm{free}}(C(y^{\delta^+}))$. More generally, the values of C can be formula-valued λ-terms (instead of formulas) where, for $y^{\delta^+} \in \mathrm{dom}(C)$ and $C(y^{\delta^+}) = \lambda v_0. \; \ldots \lambda v_{l-1}. \; B$,

$$\begin{array}{c} B \text{ is a formula whose free occurring variables from } \mathrm{V}_{\mathrm{bound}} \\ \text{are among } \{v_0, \ldots, v_{l-1}\} \subseteq \mathrm{V}_{\mathrm{bound}} \end{array}$$

and where, for $v_0 : \alpha_0, \; \ldots, \; v_{l-1} : \alpha_{l-1}$, we have

$$y^{\delta^+} : \alpha_0 \to \ldots \to \alpha_{l-1} \to \alpha_l \text{ for some type } \alpha_l,$$

and any occurrence of y^{δ^+} in B must be of the form $y^{\delta^+}(v_0) \cdots (v_{l-1})$.

Definition 3.3 (Q_C)
For an R-choice-condition C, we let Q_C be a total function from $\mathrm{dom}(C)$ into the set of single-formula sequents such that for each $y^{\delta^+} \in \mathrm{dom}(C)$ with $C(y^{\delta^+}) = \lambda v_0. \; \ldots \lambda v_{l-1}. \; B$ for a formula B, we have $Q_C(y^{\delta^+}) \; =$

$$\forall v_0. \; \ldots \forall v_{l-1}. \; \big(\;\; \exists y. \; (B\{y^{\delta^+}(v_0) \cdots (v_{l-1}) \mapsto y\}) \;\; \Rightarrow \;\; B \;\; \big)$$

for an arbitrary $y \in \mathrm{V}_{\mathrm{bound}} \backslash \mathcal{V}(C(y^{\delta^+}))$.

The semantics of a choice-condition C is to restrict the value of y^{δ^+} to satisfy the formula $Q_C(y^{\delta^+})$, which is nothing but the axiom (ε_0) for Hilbert's ε-term

$$y^{\delta^+}(v_0)\cdots(v_{l-1}) \quad = \quad \varepsilon y.\ \big(\ B\{y^{\delta^+}(v_0)\cdots(v_{l-1}) \mapsto y\}\ \big),$$

cf. Wirth (2002). Thus, if there is any y such that $B\{y^{\delta^+}(v_0)\cdots(v_{l-1}) \mapsto y\}$ holds, then $y^{\delta^+}(v_0)\cdots(v_{l-1})$ is one of them. Note that we need the bound variables $v_0,\ldots,v_{l-1}$ to model "subordinate" ε-terms without nesting.

What makes our special existential (i.e. γ-like) treatment of Hilbert's ε practically important is that it admits the global replacement of a free δ^+-variable y^{δ^+} with *any* term t satisfying $Q_C(y^{\delta^+})\{y^{\delta^+}{\mapsto}t\}$. Thus, we can model any set of constraints on a variable y^{δ^+} by simply setting $C(y^{\delta^+})$ to a formula expressing it. Above that, in combination with the variable-condition R the concrete representation of this set of constraints may provide some information on how to solve it operationally:

Example 3.4
Suppose that we have the set of ordering constraints $x^{\delta^+}{<}y^{\delta^+}$ and $y^{\delta^+}{<}z^{\delta^+}$.
We can model this in several different ways.

If the problem is to find an intermediate value y^{δ^+}, then we take

$$C(y^{\delta^+}) \ := \ \big(\ x^{\delta^+}{<}y^{\delta^+} \wedge y^{\delta^+}{<}z^{\delta^+}\ \big),$$

with the variable-condition $R := \{(x^{\delta^+},y^{\delta^+}),\ (z^{\delta^+},y^{\delta^+})\}$, saying:

Pick y^{δ^+} depending on x^{δ^+} and z^{δ^+} such that $C(y^{\delta^+})$ holds, if possible.

If the problem is to find an upper and lower bound for y^{δ^+}, then we take

$$\begin{array}{lcl} C(x^{\delta^+}) & := & (x^{\delta^+}{<}y^{\delta^+}), \\ C(z^{\delta^+}) & := & (y^{\delta^+}{<}z^{\delta^+}), \end{array}$$

with $R := \{(y^{\delta^+},x^{\delta^+}),\ (y^{\delta^+},z^{\delta^+})\}$, instead, saying:

Pick x^{δ^+} depending on y^{δ^+} such that $C(x^{\delta^+})$ holds, if possible.
Pick z^{δ^+} depending on y^{δ^+} such that $C(z^{\delta^+})$ holds, if possible.

If we do not exactly know what the problem is, then we take

$$\begin{array}{lcl} C(x^{\delta^+}) & := & (x^{\delta^+} = 1^{\mathrm{st}}(w^{\delta^+})), \\ C(y^{\delta^+}) & := & (y^{\delta^+} = 2^{\mathrm{nd}}(w^{\delta^+})), \\ C(z^{\delta^+}) & := & (z^{\delta^+} = 3^{\mathrm{rd}}(w^{\delta^+})), \\ C(w^{\delta^+}) & := & \big(\ 1^{\mathrm{st}}(w^{\delta^+}) < 2^{\mathrm{nd}}(w^{\delta^+}) \ \wedge\ 2^{\mathrm{nd}}(w^{\delta^+}) < 3^{\mathrm{rd}}(w^{\delta^+})\ \big), \end{array}$$

with $R := \{w^{\delta^+}\} \times \{x^{\delta^+},y^{\delta^+},z^{\delta^+}\}$, instead.

3.3 The global variables R and C

For each task forest we need two global variables R and C for the variable-condition and the R-choice-condition, respectively. Initially both global variables R and C are set to the empty set. They are extended by actions performing inference step, especially δ-steps. Moreover, when a substitution σ on $\mathrm{V}_{\gamma\delta^+}$ is globally applied, R and C are updated as follows:

Definition 3.5 (σ-Update)
Let R be a variable-condition and σ be a substitution.
The *σ-update of R* is $R \;\cup\; \{\ (z,x) \mid x \in \mathrm{dom}(\sigma) \wedge z \in \mathcal{V}_{\mathrm{free}}(\sigma(x))\ \}$.

Definition 3.6 (Extended σ-Update)
Let C be an R-choice-condition and let σ be a substitution.
The *extended σ-update* (C', R') *of* (C, R) is given by:

$$\begin{aligned} C' \;&:=\; \{\ (x, B\sigma) \mid (x,B) \in C \wedge x \notin \mathrm{dom}(\sigma)\ \},\\ R' \;&\text{ is the } \sigma\text{-update of } R, \text{ cf. Definition 3.5.}\end{aligned}$$

The variable-condition R checks whether it is admitted to apply a substitution globally to the proof forest. If so, it is called an *R-substitution*:

Definition 3.7 (R-Substitution)
Let R be a variable-condition.
σ is an *R-substitution* if σ is a substitution and the σ-update of R is wellfounded.

Note that wellfoundedness in a *finite* graph is the same as acyclicity and can be checked in linear time.

An R-substitution σ replacing a free δ^+-variable y^{δ^+} has to satisfy $Q_C(y^{\delta^+})\sigma$ in the sense that $Q_C(y^{\delta^+})\sigma$ becomes a global lemma of the task trees depending on y^{δ^+}. This means that an R-substitution may instantiate a free δ^+-variable y^{δ^+} only if there is a task tree whose root task is $Q_C(y^{\delta^+})\sigma$.

3.4 Semantics?

Finally, we should remark that our three different kinds of free γ-, δ^+-, and δ^--variables together with a variable-condition R and an R-choice-condition C enjoy a modular semantics which can be added to practically any two-valued semantics for formulas and with respect to which the described syntactical operations are sound, cf. Wirth (2004a), Wirth (2002).

4 Actions

The actions defined in this section all refer to an implicitly assumed task forest.

We need some global variable "*current task node*" which points to a task node in this forest unless the forest is empty, which we assume to be the initial state. There are to be some operations for setting this variable to any task node in the task forest. Moreover, as described in Section 3, we need two global variables R and C for the variable-condition and the R-choice-condition, respectively.

As in the abstract calculi of Wirth (2004a), Section 2.4, we partition our actions into *Hypothesizing*, *Instantiation*, and *Expansion* actions.

4.1 Hypothesizing

Parameters: A task t.
Effect: A new task tree with a new number is added to the task forest. This new tree consists only of a root, which is an AND action node labeled with "Start", and its single child, which is a task node labeled with t and becomes the new current task node.

4.2 Instantiation

4.2.1 Instantiation of Free γ-Variables

Parameters: An R-substitution σ on V_{γ}.

Effect: σ is applied *globally* to the whole task forest. The global variable-condition R and the choice-condition C are updated such that (C, R) is set to the extended σ-update of (C, R), cf. Definition 3.6.

4.2.2 Instantiation of Free γ- and δ^{+}-Variables

Parameters: An R-substitution σ on $\mathrm{V}_{\gamma} \cup \mathrm{V}_{\delta^{+}}$ and a function j.

Condition: The domain of j is the intersection of the domains of C and σ and its range is among the numbers of proof trees. For each free δ^{+}-variable $y^{\delta^{+}}$ in the domain of j, the task of proof tree number $j(y^{\delta^{+}})$ must be $Q_C(y^{\delta^{+}})\sigma$, cf. Definition 3.3.

Effect: σ is applied *globally* to the whole task forest. Moreover, for each $y^{\delta^{+}}$ in the domain of j, we add (unless already present) a LEM node labeled with $j(y^{\delta^{+}})$ as child of the root of each task tree with number i, provided that the following property holds:

> $y^{\delta^{+}}$ is related in the reflexive transitive closure of R to some free δ^{+}-variable $x^{\delta^{+}}$ (i.e. $y^{\delta^{+}} R^{*} x^{\delta^{+}}$) and $x^{\delta^{+}}$ occurs in a task labeling a task node in the i^{th} HYP grove or in a root task of a tree whose number labels a LEM node in the i^{th} HYP grove.

Finally, the global variable-condition R and the choice-condition C are updated such that (C, R) is set to the extended σ-update of (C, R).

4.3 Expansion

Expansion actions are local to a task node: They pick a task node from a tree and add a new action node as its child, which again may have a subtree below it, typically only a set of children again. In case of an action that recognizes a tautology, this set is empty.

Note that all the following actions are sound and safe.

4.3.1 Tautologies

Condition: The task labeling the current task node is among the set of tautologies of the logic, i.e. the formula of this task (cf. Definition 2.2) is tautological. These tautologies typically include those tasks that have two signed formulas differing only in the sign or that have a formula that is a positively signed reflexivity $(t = t)^+$.

Effect: The current task node gets a new child labeled with the AND action "Taut", which has no children.

Remark: A meaningful update of the current task node would be nice here.

4.3.2 Rewriting

Parameters: Two different natural numbers m and n, and a position p.

Condition: In the task of the current task node there are signed formulas A and B at positions m and n, respectively, where $\mathrm{Unsign}(A)$ is of the form $\neg(s = t)$ or $\neg(t = s)$ and B/p is the term s.

Effects: The current task node gets a new child labeled with the AND action "Rewrite(m, n, p)", which again gets a single child, which is a new task node (which finally is to become the current task node) whose task results from the one of the current task node by replacing the n^{th} signed formula with $B[\,p \leftarrow t\,]$.

Remark: Although our rewrite actions are sufficient for first-order completeness, we would actually like to have stronger rewrite actions, such as the ones in CoRe, cf. Autexier (2004).

4.3.3 Decomposition

Parameters: A natural number n and a position p.

Condition: The task labeling the current task node lists at least n formulas, the n^{th} formula has a position p, and the calculus admits the decomposition of the n^{th} formula up to position p.

Effect: The current task node gets a new child labeled with the AND action "Decomp(n,p)", which again gets as children some new task nodes whose labels form a non-empty set T. T results from a decomposition that turns the formula at position p in the n^{th} formula of the original task into a top formula of one of the tasks in T, and a task node labeled with this task becomes the new current task node. T must be logically equivalent to the original task.

Remark: The exact way in which the decomposition takes place is to be programmed by the logic programmer at another interface of the logic engine that is not available to the user. This programming of the decomposition may add optional parameters to the Decomposition action, such as a tag for β-downfolding taking values from $\{\mathsf{fold\text{-}left}, \mathsf{fold\text{-}right}, \mathsf{do\text{-}not\text{-}fold}\}$, cf. Wirth (2004b) for details.

4.3.4 Lemma and Hypothesis Application

Parameters: A non-empty list l of triples (n,r,σ), where n is a number of an existing task tree, $r \in \{\text{LEM}, \text{HYP}\}$, and σ is a substitution on $\left\{\ y^{\delta^-} \in \mathcal{V}_{\delta^-}(\varPhi) \ \middle|\ \mathcal{V}_{\gamma\delta^+}(\varPhi) \times \{y^{\delta^-}\} \ \subseteq\ R\ \right\}$, where $\varPhi$ is the task of task tree n, $\mathcal{V}_{\gamma\delta^+}(\varPhi)$ is the set of free γ- and free δ^+-variables in $\varPhi$, and R is the current variable-condition.

Effect: The current task node gets a new child labeled with the AND action "Apply(l)", which again gets the following children: A new task node (which finally is to become the current task node) whose task results from the one of the current task node by prepending, for each (n,r,σ) in l, the negatively signed formula of the task of tree n instantiated with σ. Moreover, for each (n,LEM,σ) in l, a LEM node labeled with n. Finally, for each (n,HYP,σ) in l, a HYP node labeled with n and a task node whose task results from the one of the current task node by prepending a signed formula taking care of wellfoundedness of induction, cf. Wirth (2004a) for details.

Remark: For practical usefulness, this has to be improved a little: We should not add the formula of the whole instantiated task but only the formula of the subtask that results from deleting all signed formulas of the instantiated task that occur already in the current task.

4.3.5 Cut

Parameters: A formula A.

Effect: The current task node gets as a new child an AND action node labeled with "Cut(A)", which again gets two task nodes as children whose tasks results from the one of the current task node by prepending A^- (i.e. A with a negative sign) (whose task node finally is to become the current task node) and A^+, respectively.

4.3.6 Safe Method Application

Parameters: A method call M and a set of tasks T.

Condition: T is the result of a safe call to the method M for the current task node.

Effect: The current task node gets as a new child an AND action node labeled with "Method(M)", whose children are a set of task nodes labeled with T plus a new task for guaranteeing soundness. The new current task node is set to one of those new task nodes that has a label from T.

4.3.7 General Method Application

Parameters: A method call M and a set of tasks T.

Condition: T is the result of a call to the method M for the current task node.

Effect: The tasks in T are partitioned by a semi-decision procedure for safeness relative to the current task node with timeout into a set of safe tasks T_1 and a set of possibly unsafe tasks T_2. The Hypothesizing action is executed for all tasks in T_2, resulting in new task trees whose numbers form the set N. The current task node gets as a new child an AND action node labeled with "Method(M)". This action node gets a set of LEM nodes as children whose set of labels is N. Moreover, we add to this action node a set of children labeled with T_1 plus a new task for guaranteeing soundness. The new current task node is set to one of those new task nodes that has a label from T.

5 Backtracking

5.1 Introduction

Mathematicians often make different proof attempts, switching from one attempt to another if they get stuck, until they succeed. They introduce lemmas which they only prove if they turn out to be useful. If they find out that a lemma was wrong, they remove all its applications. The task interface is to support this (tentative) style of *proof engineering*.

As explained in Section 2, it is already possible to start several proof attempts in parallel, which is already implemented in the systems TECTON and QUODLIBET, cf. Kapur &al. (1994) and Avenhaus &al. (2003). To do so, a task node may have several action nodes as children. Any of the subtrees rooted in these children represents a proof attempt. Hence in general one can construct an AND-OR tree to represent the proof construction. One can work on the different proof attempts independently, just as it seems most promising to achieve a complete proof. So neither replay nor backtracking is needed so far. All the proof attempts are at the disposal of the user.

Although backtracking to choice-points that freeze states at a certain point of time in the past of our proof search process is not needed, sometimes there is a need to clear up the task forest from useless branches. This is especially the case after a failure has been detected, i.e. a task has been disproved, i.e. shown to be invalid for all possible instantiations of the free variables. Since this removal of useless steps has to take care of the depending steps, the clearance process can be seen as a dependency-directed backtracking.

Suppose we have disproved a task t' of a tree n. In this case we should backtrack to a possibly unsafe step that may have caused this invalidity. If, however, all steps in tree n are safe, then the root task t is invalid. This may have two reasons: Either a Hypothesizing step introduced an invalid root task, or the root task was modified later by an invalidating Instantiation step:

- If there have been no Instantiation steps affecting the task t, then we should either patch the task t or otherwise remove tree n from the task forest and undo all the applications of t as a lemma (LEM) or as an induction hypothesis (HYP).
- Otherwise, we should undo an Instantiation step affecting the task t and then see whether we can still detect a failure by disproving the disinstantiated task t'.

In practical theorem proving, many hypothesized root tasks turn out to be invalid. Thus, failure detection is of major practical importance. As a very rudimentary but already quite useful example on how failure detection may work, the simplification tactics in QUODLIBET remove redundant signed formulas in the tasks. This often ends up with the empty task, which is the only task that is recognized to be invalid by QUODLIBET in the current version.

5.2 Requirements for the Task Interface

Apart from why we want to have a dependency-directed backtracking, what we have to specify here are the undoing facilities that the task interface has to provide. Again categorizing into Hypothesizing, Instantiation, and Expansion, these facilities are the following.

5.2.1 Undoing Hypothesizing

There must be means for undoing a Hypothesizing step. When we delete tree n once started for an in the meanwhile refuted root task, we have to correct all Expansion actions that introduced reference nodes labeled with n. There are two ways to do this; either we delete the Expansion action (cf. Section 5.2.3) and possibly lose the whole subtree; or else—if we have reason to believe that we only missed some premise of the hypothesized root task—we can add a new negatively signed free γ-variable of Boolean type to the task and generate the new branches resulting from this new signed formula to all Expansion actions that introduced reference nodes labeled with n. In the latter case we should give the tree n a new number and replay its construction with the extended root task. A similar replay mechanism after deletion of a signed formula of a root task will be useful for the repair mechanism when undoing Expansion actions.

5.2.2 Undoing Instantiation

It must be possible to undo each variable instantiation independently. Even if several variables are instantiated by a single substitution, there must be a way to disinstantiate each variable separately. As our treatment of variable-conditions does not provide means to reuse variables anyway, it is reasonable to require that each free variable symbol is used only for one purpose and not reintroduced after it has been removed by global substitution. When instantiation is realized lazily in the sense that we connect a variable symbol to a term but do not replace the occurrences of the variable symbol in the tasks, this can be easily achieved. The price of such a disinstantiation can nevertheless be high because we have to check for all the Expansion actions connected to tasks containing the disinstantiated variable, whether the actions are still executable. If not, we have to undo these Expansion actions, too.

5.2.3 Undoing Expansion

Undoing an Expansion action in QuodLibet has the consequence that the whole subtree rooted in this step is lost. This is a simple but not always convenient solution, because the construction of this subtree may have taken some efforts. Preferable would be a repair mechanism. Deleting an action node cuts a tree into several proof trees when we add a "Start" action as a new root to each former child of the deleted action node. Note that we have to copy all reference nodes that are children of the "Start" action root node of the original tree as children to the new "Start" action root nodes. Finally, the user may be asked for further help; either to delete the new proof trees, to keep them, or to modify the root task and trying to replay as much of the tree construction as possible.

6 Example

7 Conclusion

References

Serge Autexier (2004). *Theory and Architecture of an Hierarchical Contextual Reasoning Framework.* Ph.D. thesis. FR Informatik, Saarland Univ..

Jürgen Avenhaus, Ulrich Kühler, Tobias Schmidt-Samoa, Claus-Peter Wirth (2003). *How to Prove Inductive Theorems?* QUODLIBET*!.* 19th CADE 2003, LNAI 2741, pp. 328–333, Springer.

Melvin C. Fitting (1996). *First-Order Logic and Automated Theorem Proving.* 2nd extd. ed., Springer.

Gerhard Gentzen (1935). *Untersuchungen über das logische Schließen.* Mathematische Zeitschrift **39**, pp. 176-210, 405–431.

David Hilbert, Paul Bernays (1968/70). *Grundlagen der Mathematik.* 2nd ed., Springer.

Malte Hübner, Christoph Benzmüller, Serge Autexier, Andreas Meier (2003). *Interactive Proof Construction at the Task Level.* Proceedings of the Workshop User Interfaces for Theorem Provers (UITP 2003), pp. 81–100.

Deepak Kapur, David R. Musser, and X. Nie (1994). *An Overview of the Tecton Proof System.* Theoretical Computer Sci. **133**, pp. 307–339, Elsevier.

Ulrich Kühler (2000). *A Tactic-Based Inductive Theorem Prover for Data Types with Partial Operations.* Ph.D. thesis, Infix, Sankt Augustin.

Jörg H. Siekmann, Christoph Benzmüller, Vladimir Brezhnev, Lassaad Cheikhrouhou, Armin Fiedler, Andreas Franke, Helmut Horacek, Michaël Kohlhase, Andreas Meier, Erica Melis, Markus Moschner, Immanuël Normann, Martin Pollet, Volker Sorge, Carsten Ullrich, Claus-Peter Wirth, Jürgen Zimmer (2002). *Proof Development with* ΩMEGA. 18th CADE 2002, LNAI 2392, pp. 144–149, Springer.

Raymond M. Smullyan (1968). *First-Order Logic.* Springer.

Wayne Snyder, Jean Gallier (1989). *Higher-Order Unification Revisited: Complete Sets of Transformations.* J. Symbolic Computation **8**, pp. 101–140, Academic Press.

Claus-Peter Wirth (2002). *A New Indefinite Semantics for Hilbert's epsilon.* 11th TABLEAU 2002, LNAI 2381, pp. 298–314, Springer. `http://www.ags.uni-sb.de/~cp/p/epsi/welcome.html (Feb. 04, 2002).`

Claus-Peter Wirth (2004a). *Descente Infinie + Deduction.* Logic J. of the IGPL **12**, pp. 1–96, Oxford Univ. Press. `http://www.ags.uni-sb.de/~cp/p/d/welcome.html (Sept. 12, 2003).`

Claus-Peter Wirth (2004b). *Proof Trees as Formulas.* Draft. `http://www.ags.uni-sb.de/~cp/p/formulas/all.ps.gz (Oct. 09, 2003).`