Interactive Theorem Proving with Schematic Theory Specific Rules

Elmar Habermalz

Universität Karlsruhe (TH)
Institut für Logik, Komplexität und Deduktionssysteme
elmar.habermalz@ira.uka.de
WWW home page: http://i11www.ira.uka.de/~habermlz

Abstract. This paper presents a framework to make interactive proving over abstract data types (first order logic plus induction) more comfortable. A language of schematic rules is introduced, yielding the ability to write, to use, and even to verify these rules for any abstract data type and its theory.
The language allows to express the functionality of a rule easily and clearly. Nearly all potential rule applications are coupled with the occurrence of certain terms or formulas. One can prove with these rules simply by mouse clicks on these terms and formulas. The rule language is expressive enough to describe even complex induction rules. Nevertheless, the correctness of a rule can be verified within the same theory without use of explicit higher order logic or of a translation to some kind of meta level. So, in each state of a proof, new rules can be introduced, whenever required, and proven.

1 Introduction

An abstract data type can have a rich signature and a complex theory. If one wants to prove difficult lemmas about such a data type, then usually the proof is not only based on the axioms of the theory, e.g., other lemmas of the theory are used. Normally, these lemmas have to be edited and made available first. This can be done in the form of first order lemmas, higher order lemmas interpreted as proof rules, or other forms. This is a multi-step process, and with every step, even more complex lemmas of the theory has to be formulated, until one arrives at a point, where the current proof problem could be solved. With every step, more knowledge of the previous steps will be needed. This knowledge and its easy use have thus a great importance for the feasibility of proving.

The feasibility of interactive proving over an abstract data type is not only determined by the purely logical knowledge that can be specified in the form of formulas. In addition, one can collect information as to when, where, and how the purely logical knowledge could be applied. Then that knowledge can be offered skillfully to the user and he can select the knowledge easily to execute the next proof step.

The schematic theory specific rules presented here form a system that makes such knowledge over a complex theory available in a simple and easily understandable way as proof rules. In particular a *schematic theory specific rule (STSR)* contains:

- pure logical knowledge
- information, how this knowledge should be used
- information, when and where this knowledge should be presented for interactive use
- information, when and where this knowledge should be used automatically

A distinguishing feature of an STSR is that it is not difficult to write and to read it, because of its clear syntax. In addition, STSRs are relatively expressive, even rules for induction can be formulated. Their correctness can be represented by the correctness of a formula that can be calculated from the STSR syntactically. The formula can be proven in the same framework and with the previously introduced STSRs. Therefore correctness proofs for rules are no more difficult than for normal formulas and lemmas.

The concept of STSRs was influenced by two existing systems, KIV and InterAct.

KIV [HMRS90,Rei95,RSSB98] is a tool for specifying and verifying abstract data types and there implementations. As part of this process, one has to prove lemmas over the specified abstract data types. The specifications use full first order predicate logic and generate conditions with loose semantics. The proofs in KIV can be made interactively using a sequent calculus. Not all steps have to be executed interactively. The *simplifier*, the *problem-specific heuristic* and further special heuristics are available as automation assistance. More knowledge about the abstract data type and its theory is gained and organized primarily through lemmas. The *simplifier* does term rewriting and simple logical transformations. If a lemma has a certain form, it can be used by the simplifier. This form also states how the logical knowledge of the lemma should be used. The *problem-specific heuristic* gives the possibility of executing any proof step automatically. Every entry in the problem-specific heuristic includes formula patterns and a proof rule. The prover compares the formula patterns with the formulas of the sequent. When this comparison yields the appropriate matches, the rule will be executed automatically. The interactive steps, e.g. split conjunctions, instantiate univeral quantifier, use an induction rule, or insert lemmas, can be made with a comfortable graphical user interface. Particularly when working with complex theories, induction and insertion of lemmas are the most important steps.

The idea of proving by clicking on terms and formulas was applied before by InterAct [KGC96,GKC96]. InterAct is a system that allows the specification of abstract data types. The specification language uses conditional equations with initial semantics. The underlying calculus is again a sequent calculus. The prover also has a simplifier that can use conditional equations for automated simplifications. But the most interesting feature of InterAct is its graphical user interface. The offered proof steps are divided into global and local steps. Global steps are steps like induction or cut. The local steps are e.g. instantiation of universal quantifier or paramodulation. These steps are controlled by clicking on the appropriate terms and formulas.

The STSRs, which I introduce here, have the advantage that the four concepts of the simplifier, the problem-specific heuristic, the use of lemmas by clicking, and general proving by clicking are united in one concept. An STSR can

- be an automatically or interactively used rewriting rule
- represent a lemma
- contain necessary context information for automatic use
- contain information on using it by mouse-click on formulas or terms

And with the possibility to combine all these features and, at the same time, be more expressive than first order lemmas it goes beyond those concepts.

The idea has simularities with systems like Isabelle [Pau90] or PVS [ORS92], where higher order lemmas can be interpreted as proof rules. Such proof rules are much more powerful than the STSRs presented here. However, the STSRs avoid to confront a user with proofs in higher order logic. Moreover, the direct combination with the interaction makes the STSRs different.

Other concepts on proving by mouse click have been presented: *proof by pointing* [BKT94] and *proof by drag and drop* [Ber97] are two very interesting techniques. The aim of *proof by pointing* is to bring subformulas to the surface, however, it does not focus on theory specific knowledge. The aim of *proof by drag and drop* is to select term rewriting steps by what Bertot calls a *gesture*. The concept he presents allows for highly differentiated preselection. On the other hand, this system is quite complex and therefore requires a lot of training on part of the user. Thus, in developing the STSRs I decided to allow proving by single click only, to keep the concept simple and manageable.

The following sections will present the STSRs. The second section introduces the STSRs. In the third section a calculus is formed by the STSRs. Section four presents a system that operates with the STSRs and section five gives a summary.

2 Schematic Theory Specific Rules

First, an abstract data type for natural numbers is introduced as an example. It serves as a basis for a constructive introduction to the STSRs. Afterwards the STSRs are introduced more formally and with a larger outline. Finally the example is expanded.

2.1 Example: natural numbers

The STSRs are used for proving conjectures about abstract data types. The abstract data types are given as algebraic specifications, using sorts, full first order axioms and generation conditions, like in KIV. It can be seen as a sublanguage of CASL (Common Algebraic Specification Language, [CoF99]).

The specification for natural numbers is:

signature

sorts nat;

functions 0 : nat;

. +1 (nat) : nat;

. −1 (nat) : nat;

. + . (nat,nat) : nat;

. ∗ . (nat,nat) : nat;

predicates . < . (nat,nat);

variables n, n_0, n_1, n_2, ... : nat;

generate conditions {nat} **generated by** {0, +1};

axioms $\neg 0 = n\ {+1}$, $\quad n\ {+1} = n_0\ {+1} \leftrightarrow n = n_0$,

$(n\ {+1})\ {-1} = n$,

$\neg\ n < 0$, $\quad n < n_0\ {+1} \leftrightarrow (n = n_0 \vee n < n_0)$,

$n + 0 = n$, $\quad n + (n_0\ {+1}) = (n + n_0)\ {+1}$,

$n * 0 = 0$, $\quad n * (n_0\ {+1}) = n + (n * n_0)$

The axioms are implicitly, universaly quantified. The generate conditions restricts the models of the mentioned sorts to appropriate term generated models. Natural numbers are not the main goal of STSR. They are more useful for data types that are made new for a software project or other projects. But since natural numbers are well known and therefore easier to understand, they serve here as an illustrating example.

2.2 Steps to STSRs

An STSR contains logical knowledge and information about how, where and when it should be used. Let us start with the lemma

$$n_0 < n_1 \leftrightarrow n_0 + n < n_1 + n$$

to explain, how those informations are coded.

One possibility of reading operational information into this lemma is to take a formula of the form $t_0 < t_1$, to ask the user for a term t, and to replace the formula with $t_0 + t < t_1 + t$. From this special operational interpretation it results immediately that it is only needed when a formula of the form $t_0 < t_1$ is present. Moreover, it is meaningful to connect the lemma with this special operational interpretation and such a formula, and to offer the application of the lemma to the user in this form, if he selects a formula of the form $t_0 < t_1$. The connection of the operational and the logical information results in something of the form:

`find`$(n_0 < n_1)$ `replacewith`$(n_0 + n < n_1 + n)$

The keyword `find` marks a formula pattern that must be found in the current proof problem so that the lemma is applicable in the intended form. The interaction is defined by `find` too. The idea is that this rule will be offered to the user, when he clicks on a formula of the form $n_0 < n_1$. The rule is bound to the formulas or terms that match the expression given with `find`. `replacewith` and the following expression describes how the new proof goal is calculated from the old one. The STSRs are laid out for a sequent calculus. Therefore, the current goal is a sequent and will be adressed as one. I assume that the reader is familiar with the concepts of sequent calculus as explained e.g. in [Gal86]. If such a STSR is used, the variables in the argument of `find` will be instantiated by matching the selected term or formula of the current goal. All other variables need an instantiation supplied by the user.

Also, it must be possible to adress formulas only in the antecedent or succedent of a sequent. This is possible by permitting the use of sequents in `find` and `replacewith` with '$\Rightarrow$' as sequent arrow. Example:

$$\texttt{find}(\Rightarrow n * n_0 = 0)\ \texttt{replacewith}(\Rightarrow n = 0, n_0 = 0)$$

Sometimes, it is necessary to produce more than one remaining goal by an STSR. Consequently, it is possible to specify more than one description for a new goal in an STSR. The descriptions are seperated by ';':

$$\texttt{find}(\Rightarrow n + n_0 = 0)\ \texttt{replacewith}(\Rightarrow n = 0);\ \texttt{replacewith}(\Rightarrow n_0 = 0)$$

If the argument of `find` is a sequent, it must not contain more than one formula. With more than one formula, the way of interacting with the STSR would not be clear.

If a formula should remain in the sequent that is found by `find`, the keyword `add` is used instead of `replacewith`. Example:

$$\texttt{find}(n < n_0 \Rightarrow)\ \texttt{add}(n < n_0{+}1 \Rightarrow)$$

It is also possible to combine both keywords:

$$\begin{aligned}&\texttt{find}(n{-}1{+}1)\ \texttt{replacewith}(n)\ \texttt{add}(\neg n = 0 \Rightarrow);\\ &\phantom{\texttt{find}(n{-}1{+}1)\ }\texttt{add}(n = 0 \Rightarrow)\end{aligned}$$

This STSR makes a case distinction. First it is assumed that n is not equal to 0 ($\texttt{add}(\neg n = 0 \Rightarrow)$). Then, $n{-}1{+}1$ could be replaced by n. Secondly it is assumed that n is equal to 0. $n{-}1{+}1$ remains unchanged and $n = 0$ is added to the antecedent.

It was mentioned that the STSR can also be used automatically. This mechanism should not be compared with automated theorem provers. But it can free the user from doing trivial proof steps. To mark an STSR as a rule that can be used automatically there exists the keyword `heuristics`, followed by a list of names, called heuristic names. The user should select a list of active heuristics from these. If a certain STSR contains an active heuristic, this STSR should be executed automatically. This enables e.g. simple automated term rewriting.

To implement conditional execution, a condition is necessary. The keyword `if` fulfils this task. An example:

$$\texttt{if}(\Rightarrow n = 0)\ \texttt{find}(n{-}1{+}1)\ \texttt{replacewith}(n)\ \texttt{heuristics}(\text{nat})$$

The rule will be executed automatically whenever there is an instantiation for n such that $n{-}1{+}1$ can be found anywhere in the sequent and $n = 0$ can be found in the succedent of the current sequent. The argument of `if` is a condition that has to be fullfilled when the rule is executed. For automated use, the sequent of `if` must be a part of the current sequent. For interactive use, this is not necessary, but an extra goal will be created to check, whether that sequent could be deduced from the current sequent.

Also, the closure of a proof branch can be expressed as an STSR. If no description for a new goal is given, no new goal will be created, and with that the current proof branch will be closed. E.g.:

`if`$(n + 1 < n_0 \Rightarrow)$ `find`$(\Rightarrow n < n_0)$ `heuristics`(nat)

2.3 STSRs more systematically

For a more systematical formulation of STSRs we need to extend the vocabulary of the abstract data type nat and of formulas in general by additionally symbols:

heuristic names formula, nat;
formula skolem symbols p, p_0, p_1, p_2, ...;
skolem symbols c, c_0, c_1, c_2, ... : nat;
formula variables b, b_0, b_1, b_2, ...;
rule variables $rule$, $rule_0$, $rule_1$, $rule_2$, ...;

The term "formula" is a defined heuristic name for STSRs that handles formulas in general. The skolem symbols enrich the set of terms and formulas. With formula skolem symbol p, skolem symbol c of the sort nat and terms $t_1, ..., t_n$ (of arbitrary sorts), p and $p(t_1, ..., t_n)$ are formulas and c and $c(t_1, ..., t_n)$ are terms of the sort nat. Thus, every symbol could be used as a constant or as a function or predicate symbol. These formulas and terms are allowed in rules and also in sequents. The skolem symbols are needed for skolemisation of free variables in the sequents to seperate free variables from bound variables, and also to express the correctness of a rule as a formula. Formula variables are standing for arbitrary formulas in rules, they can only be used in rules. For the formulas and terms occuring in rules but not in `find` or `if`, there are some special kinds of terms and formulas defined. With variable x, terms t_1, t_2 and formulas φ_1, φ_2, rule variable $rule$, there are:

$\{x\ t_1\}(t_2)$ term (of the sort of t_2, x and t_1 must be of the same sort)
$\{x\ t_1\}(\varphi_2)$ formula (x and t_1 must be of the same sort)
$\{x\ \varphi_1\}(\varphi_2)$ formula (x must be a formula variable)
`proofobl`$(rule)$ formula

The first three terms and formulas denote substitutions, e.g. $\{x\ t_1\}(t_2)$ means that every x in t_2 should be replaced by t_1. The substitution will be executed collision free. `proofobl`$(rule)$ stands for the formula that represents the correctness of $rule$. If $rule$ is instantiated by an STSR, that formula could be calculated and could replace `proofobl`$(rule)$. This formula is called the proof obligation of an STSR.

With that expanded notion of terms and formulas, we can go back to the STSRs. An STSR has three parts:

- the application part
- the goal descriptions
- the heuristic part

Definition: application part
The application part has the form

`if`($ifseq$) `find`($findexp$) `varcond`($varexplist$)

where $ifseq$ is a sequent, $findexp$ a term, formula or a sequent with not more than one formula and $varexplist$ a list of elements of the form

x `new` with x variable
y `not free in` z with y, z variables.

Unused keywords can be omitted (`if` with the empty sequent, `find` with the empty sequent, `varcond` with an empty list). `if` and `find` should be clear. `varcond` is necessary for describing independencies between variables on one side and terms and formulas on the other side. A new variable can always automatically be instantiated.

Definition: goal descriptions
The goal descriptions are presented as a list in which the individual descriptions are separated by ';'. Each goal description is of the form

`replacewith`($rwexp$) `add`($addseq$) `addrules`($stsrlist$)

with $rwexp$ appropriate for $findexp$ (if $findexp$ is a sequent, the sequent in $rwexp$ is not limited to one formula), $addseq$ a sequent and $stsrlist$ a list of STSRs and rule variables. Again, unused keywords can be omitted (`replacewith` with the same expression as `find`, `add` with the empty sequent, `addrules` with an empty list). `replacewith` and `add` have already been explained. `addrules` creates new rules that are available for the sequent created by that goal description and its successors. If `addrules` contains a rule variable, it could be instantiated by every possible rule.

Definition: heuristic part
The heuristic part is simple, it is of the form

`heuristics`($heunamelist$)

with $heunamelist$ a list of heuristic names.

Finally, STSRs could (and should) have names, so that the they are easier to organize and to identify. To sum it up, an STSR has the structure:

$name\{$
$\texttt{if}(ifseq)$ $\texttt{find}(findexp)$
  $\texttt{varcond}(x_1 \texttt{ new}, ..., x_\nu \texttt{ new}, y_1 \texttt{ not free in } z_1, ..., y_\mu \texttt{ not free in } z_\mu)$
$\texttt{replacewith}(rwexp_1)$ $\texttt{add}(addseq_1)$ $\texttt{addrules}(r_{1,1}, ..., r_{1,\varrho_1})$;
$\vdots$
$\texttt{replacewith}(rwexp_\gamma)$ $\texttt{add}(addseq_\gamma)$ $\texttt{addrules}(r_{\gamma,1}, ..., r_{\gamma,\varrho_\gamma})$
$\texttt{heuristics}(heu_1, ..., heu_\xi)$
$\}$

Some examples will make the possibilities of the STSRs more clear.

2.4 Examples

STSRs cover all areas from quite general rules to very specific ones. Very general ones represent base rules of the sequent calculus like:

and-left { $\texttt{find}(b \wedge b_0 \Rightarrow)$ $\texttt{replacewith}(b, b_0 \Rightarrow)$ $\texttt{heuristics}$(formula) }
eq-right { $\texttt{find}(\Rightarrow b \leftrightarrow b_0)$ $\texttt{replacewith}(b \Rightarrow b_0)$;
$\texttt{replacewith}(b_0 \Rightarrow b)$ }
close { $\texttt{if}(b \Rightarrow)$ $\texttt{find}(\Rightarrow b)$ $\texttt{heuristics}$(formula) }

and-left and eq-right are based on the appropriate rules of the sequent calculus, close is based on the closing mechanism of it. The whole sequent calculus can be covered. With introducing a sort, e.g. quantifier instantiation is needed. It could be made in to ways:

all-left-1 { $\texttt{find}(\forall x\ b \Rightarrow)$ $\texttt{add}(\{x\ x_0\}(b) \Rightarrow)$ }
all-left-2 { $\texttt{if}(\forall x\ b \Rightarrow)$ $\texttt{find}(x_0)$ $\texttt{add}(\{x\ x_0\}(b) \Rightarrow)$ }

If the user clicks on a universally quantified formula in the antecedent, he can use all-left-1 and would be asked to insert an instantiation for x_0. If there is already an appropriate term in the current sequent, it would be better to use all-left-2. The user would click on that term and choose all-left-2. A system could easily offer all existing possibilities for x and b.

More specialized rules handle induction. The structural induction rule over the natural numbers is:

nat-induction{
$\texttt{varcond}(n_0 \texttt{ new})$
$\texttt{add}(\Rightarrow \{n\ 0\}(b))$;
$\texttt{add}(\{n\ n_0\}(b) \Rightarrow \{n\ (n_0 + 1)\}(b))$;
$\texttt{add}(\forall n\ b \Rightarrow)$ }

If a user executes this rule with a formula $\varphi(n)$ for b, he will receive three new goals. The first goal includes $\Rightarrow \varphi(0)$, where it has to be shown that $\varphi(0)$ is valid. The next goal includes $\varphi(n_0) \Rightarrow \varphi(n_0 + 1)$, where it has to be shown that $\varphi(n_0)$ implies $\varphi(n_0 + 1)$. The third goal includes $\forall n\ \varphi(n) \Rightarrow$, the validity of $\varphi(n)$ for all n could be used for the ongoing proof. This rule has no `find` meaning that it contains $\texttt{find}(\Rightarrow)$ implicitly. To choose this rule, a user has to click on the sequent arrow $\Rightarrow$.

This could be continued, e.g. to a very special kind of induction:

jump-and-back-induction {
    $\texttt{varcond}(n_0\ \texttt{new}, n_1\ \texttt{new})$
    $\texttt{add}(\{n\ n_0\}(b) \Rightarrow \exists n_1\ (n_0 < n_1 \wedge \{n\ n_1\}(b)))$;
    $\texttt{add}(\{n\ (n_0\ +1)\}(b) \Rightarrow \{n\ n_0\}(b))$;
    $\texttt{add}((\forall n\ b) \Rightarrow)$ }

If a user executes this rule with a formula $\varphi(n)$ for b, he also gets three new goals. The first one includes $\varphi(n_0) \Rightarrow \exists n_1\ (n_0 < n_1 \wedge \varphi(n_1))$. It means, it has to be proven that if for some n_0 the formula $\varphi(n_0)$ is valid, there has to be a greater n_1 such that $\varphi(n_1)$ is also valid. It is a jumping up the natural numbers. The second one includes $\varphi(n_0\ +1) \Rightarrow \varphi(n_0)$ and is a step back. It has to be proven that the gaps of the jumping could be filled. So, it is a proof that $\varphi(n)$ is valid for all n and $\forall n\ \varphi(n)$ is included in the antecedent of the third goal.

A simpler rule also specific for the natural numbers is:

less-cases{ $\texttt{find}(n)$ $\texttt{add}(n < n_0 \Rightarrow)$; $\texttt{add}(n = n_0 \Rightarrow)$; $\texttt{add}(n_0 < n \Rightarrow)$ };

With this rule the user can introduce a case distinction over two natural numbers.

Until now, the keyword **`addrules`** was not used. An example will make up for that. It deals with equality on nat:

make-insert-eq { $\texttt{find}(n = n_0 \Rightarrow)$
    $\texttt{addrules}$(insert-eq { $\texttt{find}(n)$ $\texttt{replacewith}(n_0)$ })
    $\texttt{heuristics}$(formula) }

If an equation appears in the antecedent of a sequent, this rule creates a new rule to replace an occurrence of its left-hand side with its right-hand side (the commutativity of the equality should be build in, so a rule is made for both directions).

2.5 The STSR rule-cut

Finally, there is the STSR rule-cut. With this rule, it is possible to introduce any rule at any place of a proof. But the proof splits at that point and the user has to prove in the second branch that the rule is a correct rule at that point:

rule-cut { $\texttt{addrules}(rule)$; $\texttt{add}(\Rightarrow \texttt{proofobl}(rule))$ };

With this rule it is possible to introduce highly specialized rules in a proof, maybe rules that are only correct rules in the context of the current sequent but rules carrying operational and heuristic information that make the rest of the current proof very simple. This rule illustrates one of the main ideas of STSRs. A formula called proof obligation can be calculated syntactically for every rule and these proof obligations guarantee the correctness of the rule. Two examples will illustrate this calculation, a formal explanation will be given elsewhere.

The first example is:

pred-succ-elim{ $\texttt{if}(\Rightarrow n = 0)$ $\texttt{find}(n{-}1{+}1)$
$\texttt{replacewith}(n)$ $\texttt{heuristics}(\text{nat})$ }

Its proof obligation is very simple:

$$c = 0 \vee c{-}1{+}1 = c$$

The second example is:

jump-and-back-induction {
$\texttt{varcond}(n_0\ \texttt{new}, n_1\ \texttt{new})$
$\texttt{add}(\{n\ n_0\}(b) \Rightarrow \exists n_1\ (n_0 < n_1 \wedge \{n\ n_1\}(b)));$
$\texttt{add}(\{n\ (n_0\ {+}1)\}(b) \Rightarrow \{n\ n_0\}(b));$
$\texttt{add}((\forall n\ b) \Rightarrow)$ }

This proof obligation is a little bit more complicated:

$$\exists n_0\ \exists n_1\ (\neg(\exists n_1\ (n_0 < n_1 \wedge p(n_1))) \vee (p(n_0\ {+}1) \wedge \neg p(n_0)) \vee (\forall n\ p(n)))$$

b is replaced by $p(n)$. The first goal description yields $\neg(\exists n_1\ (n_0 < n_1 \wedge p(n_1)))$, the second yields $(p(n_0\ {+}1) \wedge \neg p(n_0))$, and the third $(\forall n\ p(n))$. The $n_0\ \texttt{new}, n_1\ \texttt{new}$ in the rule yields the $\exists n_0\ \exists n_1$ in the proof obligation. The use of skolem functions and predicates (see the predicate $p(nat) \rightarrow nat$ above) covers the higher order elements of the STSRs and makes the guarantee of correctness possible.

3 A calculus with STSRs

So far, the STSRs have been presented. But how do they work in a calculus? Here I describe a calculus that works only with STSRs. But it is also possible to use them as an add-on. One of the ideas of the STSRs is, to give a user a uniform concept for collecting theory specific knowledge. So it is not the idea to use them with a lot more concepts for theory specific knowledge when using them as an add-on. In particular, it is possible to use the STSRs as the only concept for proving. To use them for that, there are two questions to be answered. What is a proof, and how exactly could an STSR be used?

3.1 Proofs

In a standard sequent calculus a proof is a tree, the nodes are sequents and the edges are justified by rule applications of the calculus. The set of rules is fixed for a proof. If one uses STSRs, the set of rules is not fixed anymore. Consequently, every node of a proof tree is enriched by a set of STSRs.

There is a second change, which is not really necessary, but makes the explanation of correct rule instantiations easier. As mentioned before, the free variables in a sequent should be skolemized to avoid conflicts between free and bound occurrences of a variable. In the root sequent and in all sequents that are the result of a rule application, the free variables should be replaced by new skolem constants. Then conflicts are not possible.

Thus, a partial proof for a lemma φ is a tree in which:

– every node includes a sequent
– the root node includes the skolemized version of the sequent $\Rightarrow \varphi$
– every node includes a set of STSRs
– the root node includes a set of STSRs that are given axiomatically or are assumed or proven correct (monitored by a correctness management)
– a node is *'open'* or there is one STSR of the set of STSRs of this node, a correct instantiation of this rule, and the children of that node are justified by the application of that rule with that instantiation

Regarding the last point, if that rule does not introduce any new goals, it is justified that that node has no children and the branch is closed there.

3.2 Rule application

The first thing that has to be done when a user wants to apply an STSR is to instantiate the variables of the STSR. There are some restrictions to the variable instantiations of STSRs. There are three possible types of restriction. An unrestricted variable can be instantiated to any term of appropriate sort or to a formula. This is even the case if free variables of that term or that formula have an interrelation with quantifiers of the current sequent. If such an interrelation would compromise the correctness of the rule, the variable is always restricted to be instantiated to a constant term or formula. This is not a restriction to ground terms, also skolem functions and skolem predicates are allowed. To control this, an STSR is divided into regions. If a variable occurs in more than one region, the variable is restricted to be instantiated by constant terms. For a variable x the regions are defined as follows:

– if $findexp$ is a sequent or x is declared as new, $findexp$, $ifseq$, all $rwexp$, and all $addseq$ taken together are one region
– if $findexp$ is not a sequent and x is not declared as new, $ifseq$ and all $addseq$ taken together are one region.
– if $findexp$ is not a sequent and x is not declared as new, every $rwexp$ together with $findexp$ is a region.
– every single rule in a *stsrlist* is a region of its own.

Rule variables are an exeption. They always have to be instantiated by a concrete STSR (and not a rule variable), but that STSR can always include free variables.

The other two restrictions are simpler. If a variable is used quantified somewhere in the STSR, it has obviously to be instantiated with a variable. This is also the case for all y in a declaration of the form y `not free in` z and for all z of such declarations in rule lists of `addrules`. Also every x in x `new` has to be a variable, if the x `new` appears inside an `addrules` declaration. If x `new` is declared for the instantiated rule, the variable has to be new.

It is possible that a variable belongs to more than one region but also is limited to be instantiated with a variable. In such a case, the STSR has no correct instantiation and could not be used.

After the correct instantiation, the substitution terms are eliminated. If such a substitution results in a collision, the appropriate inner quantified variables would have to be renamed to avoid the collision.

The new nodes of the proof are constructed straightforward. For every goal description a new node is constructed. If the description includes a `replacewith`, the *findexp* will be replaced by it. Then the *addseq* will be added to the current sequent and the *stsrlist* will be added to the set of existing rules of the current node. If the *ifseq* is not a subsequent of the current sequent, it is first added to every sequent from a goal description, and in a second step an extra goal is created to which the negation of *ifseq* is added.

An example. Let us take the lemma

$$(n_0{-}1{+}1 < n_1 \land n_1 < n_2) \to (n_0 < n_2 \lor n_0 = 0) \tag{*}$$

and a set of STSRs including:

less-trans{ `if`($n < n_0 \Rightarrow$) `find`($n_0 < n_1 \Rightarrow$) `add`($n < n_1 \Rightarrow$) }
pred-succ-elim{ `if`($\Rightarrow n = 0$) `find`($n{-}1{+}1$)
`replacewith`(n) `heuristics`(nat) }

After skolemisation of the free variables and automatic transformation steps, (*) leads to the sequent :

$$c_0{-}1{+}1 < c_1,\;\; c_1 < c_2 \;\Rightarrow\; c_0 < c_2,\;\; c_0 = 0 \tag{**}$$

less-trans is attached to two formulas:

$$c_0\ {-}1\ {+}1 < c_1,\;\; c_1 < c_2 \Rightarrow c_0 < c_2,\;\; c_0 = 0$$

less-trans{`if`($n < n_0 \Rightarrow$) `find`($n_0 < n_1 \Rightarrow$) `add`($n < n_1 \Rightarrow$)}

If the user clicks on $c_1 < c_2$, chooses the rule less-trans and completes the instantiation with a choice for n to $n \mapsto c_0$, $n_0 \mapsto c_1$ and $n_1 \mapsto c_2$, two new goals will be created. The `if` yields

$$c_0{-}1{+}1 < c_1,\;\; c_1 < c_2 \;\Rightarrow\; c_0 < c_1,\;\; c_0 < c_2,\;\; c_0 = 0$$

with $c_0 < c_1$ new in the succedent and the `add` yields

$$c_0 < c_2,\;\; c_0 < c_1,\;\; c_0{-}1{+}1 < c_1,\;\; c_1 < c_2 \;\Rightarrow\; c_0 < c_2,\;\; c_0 = 0$$

with $c_0 < c_2$ from `add` and $c_0 < c_1$ from `if` new in the antecedent. But if the heuristic nat is active, pred-succ-elim will be executed before less-trans automatically, because the rule is attached to the term $c_0{-}1{+}1$ and the condition $\Rightarrow c_0 = 0$ is fulfiled.

$$c_0\ {-}1{+}1 < c_1,\;\; c_1 < c_2 \Rightarrow c_0 < c_2,\;\; \underline{c_0 = 0}$$

pred-succ-elim{`if`($\Rightarrow\ n\ =\ 0$) `find`($n\ {-}1\ {+}1$) `replacewith`(n)}

Therefore, the result of this operation is

$$c_0 < c_1, \quad c_1 < c_2 \;\Rightarrow\; c_0 < c_2, \quad c_0 = 0$$

After that, clicking on $c_1 < c_2$ and choosing less-trans does not generates two goals, because the condition $c_0 < c_1 \Rightarrow$ is fulfiled. There will be only one new goal with the sequent

$$c_0 < c_2, \quad c_0 < c_1, \quad c_1 < c_2 \;\Rightarrow\; c_0 < c_2, \quad c_0 = 0$$

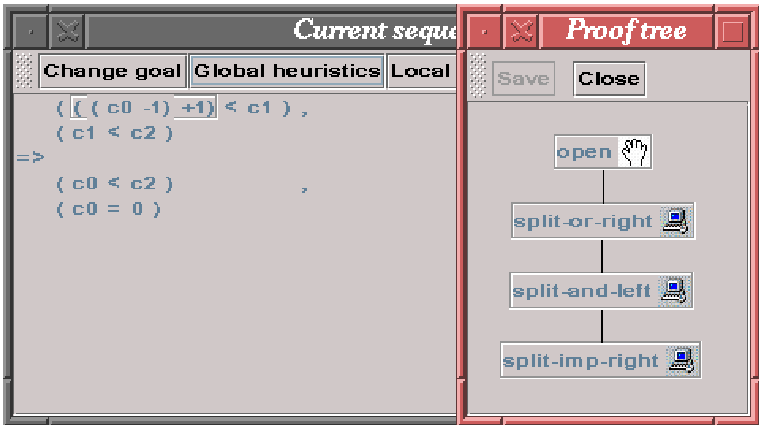
including $c_0 < c_2$ in the antecedent. Then the active heuristic formula and the STSR

close { `if`($b \Rightarrow$) `find`($\Rightarrow b$) `heuristics`(formula) };

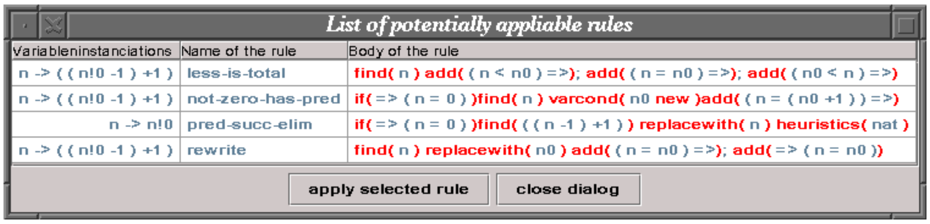
would close the proof immediatly with b instantiated to $c_0 < c_2$.

4 A System working with STSRs

The STSRs are not only a theory. The system IBIJa [Sup98] applies the idea of STSRs. The acronym is derived from the German *Interaktiver Beweiser In Java* which can be translated as *Interactive Prover In Java*.



IBIJa works on two levels, the project level and the prover level. At the project level, it is possible to create and edit projects. A project contains a structured algebraic specification enriched by the necessary STSR signature, lemmas and most importantly by STSRs. Currently, the specification language is not very elaborate. There are no generic features, but an enrichment of specifications is possible. A specification file does not only include the signature enriched for STSRs, full first order axioms and generation condition. Also lemmas, axiomatic STSRs and derivable STSRs can be written in those files and be managed by the system. A user can choose a lemma or a derivable STSR and can start to prove them. The picture illustrates the example (*) of the previous section. It shows the current sequent with the selected term c_0-1+1 and the current proof. The already made proof steps split (*) in its parts. The symbols of the proof nodes indicate, whether the step was made interactively (the hand) or automatically (the computer). If the user clicks on the selected term, the following window appears where the user can select the rule pred-succ-elim for application.



IBIJa is a prototype written in Java2. It is completely functional and can be downloaded from `http://i11www.ira.uka.de/~ibija`.

5 Conclusion

The idea of schematic theory specific rules (STSRs) is a concept to make interactive proving more comfortable. It is designed particularly to work with theory specific knowledge of abstract data types.

The rules are easy to write and easy to integrate with the already existing ones by clicking on terms and formulas in the user interface. They can be run automatically (in a simple way) and their correctness can be proven within the same framework. The application of a rule follows its syntax in a straightforward way. Basically, the STSRs provide a framework that can be adapted easily to build a comfortable interactive prover for any abstract data type.

For the future it is planned to integrate the STSRs into a theorem prover, which will be used in a project called *KeY* developed at the *Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe (TH)*. The aim of this project is to integrate formal software specification and verification into the industrial software engineering process. You can find more information about KeY at `http://i11www.ira.uka.de/~key/`.

References

[Ber97] Y. Bertot. Direct manipulation of algebraic formulae in interactive proof systems. http://www-sop.inria.fr/croap/events/uitp97-papers.html, 1997. Proceedings of the 3rd international Workshop on User Interfaces for Theorem Provers.

[BKT94] Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software (TACS '94)*, number 789 in LNCS. Springer, 1994.

[CoF99] The CoFI Task Group on Language Design. Casl, the common algebraic specification language. http://www.brics.dk/Projects/CoFI, 1999.

[Gal86] J. Gallier. *Logic for Computer Science*. Harper & Row, 1986.

[GKC96] R. Geisler, M. Klar, and F. Cornelius. Interact: An interactive theorem prover for algebraic specifications. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, number 1101 in LNCS. Springer, 1996.

[HMRS90] M. Heisel, W. Menzel, W. Reif, and W. Stephan. Der Karlsruhe Interactive Verifier (KIV). Eine Übersicht. In H. Kersten, editor, *Sichere Software, Formale Spezifikation und Verifikation vertrauenswürdiger Systeme*. Hütig Verlag, 1990. (In German).

[KGC96] M. Klar, R. Geisler, and F. Cornelius. Interact: An interactive theorem and completeness prover for algebraic specifications with conditional equations. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specification*, number 1130 in LNCS. Springer, 1996.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, number 607 in LNCS. Springer, 1992.

[Pau90] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[Rei95] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, number 1009 in LNCS. Springer, 1995.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured Specifications and Interactive Proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume I. Kluwer Academic Publishers, 1998.

[Sup98] M. Supp. Implementierung eines integriert interaktiven und reflexiven Beweisers für Prädikatenlogik. Master's thesis, Universität Karlsruhe (TH), 1998. (In German).