# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>Proving the Associativity of <code>plus</code></td>
<td>4</td>
</tr>
<tr>
<td>A hand crafted proof</td>
<td>5</td>
</tr>
<tr>
<td>A system generated proof</td>
<td>7</td>
</tr>
<tr>
<td>Proving the Commutativity of <code>plus</code></td>
<td>17</td>
</tr>
<tr>
<td>Developing Prooftrees</td>
<td>21</td>
</tr>
<tr>
<td>Using Proof Rules</td>
<td>22</td>
</tr>
<tr>
<td>Computed Proof Rules</td>
<td>23</td>
</tr>
<tr>
<td>Induction</td>
<td>24</td>
</tr>
<tr>
<td>Case Analysis</td>
<td>29</td>
</tr>
<tr>
<td>Use Lemma</td>
<td>31</td>
</tr>
<tr>
<td>Unfold Procedure</td>
<td>33</td>
</tr>
<tr>
<td>Apply Equation</td>
<td>34</td>
</tr>
<tr>
<td>Delete and Move Hypotheses</td>
<td>36</td>
</tr>
<tr>
<td>Insert Hypotheses</td>
<td>40</td>
</tr>
<tr>
<td>Testing a Program</td>
<td>41</td>
</tr>
<tr>
<td>Disproving a Lemma</td>
<td>41</td>
</tr>
<tr>
<td>Running a Procedure</td>
<td>46</td>
</tr>
<tr>
<td>Proof Engineering</td>
<td>47</td>
</tr>
<tr>
<td>Keeping Control</td>
<td>48</td>
</tr>
<tr>
<td>Analyzing a Goalterm</td>
<td>49</td>
</tr>
<tr>
<td>Decomposing a Goalterm</td>
<td>50</td>
</tr>
<tr>
<td>Creating a Lemma</td>
<td>51</td>
</tr>
<tr>
<td>Exploring Alternatives</td>
<td>52</td>
</tr>
<tr>
<td>Using Proof Libraries</td>
<td>53</td>
</tr>
<tr>
<td>Tips and Tricks</td>
<td>55</td>
</tr>
</tbody>
</table>
Introduction

Computer Science is concerned with the development of algorithms to solve certain problems. Usually one starts from a specification of the problem and then tests the first draft of the algorithm with a collection of certain test data. If the algorithm fails on some input, it must be debugged and then testing is resumed with the whole collection of the test data. This development-and-test cycle is continued until eventually the most recent version of the algorithm survives all tests given by the collection.

Having successfully passed all tests one knows that the developed algorithm solves the problem for the test data. However, it may fail for some input not provided with the test collection. We may extend this collection by additional data to become more convinced that our algorithm solves the problem in fact. But this will never guarantee the correctness of our solution, as (except for trivial problems) we would need to test the algorithm for an infinite set of data, which obviously is impossible.

The answer to this problem is Verification. While testing may prove the presence of bugs, verification may prove their absence. This is not to say that testing is not needed when algorithms are developed. Usually, the first versions of an algorithm are faulty and one would waste a lot of time when trying to verify such an algorithm. Verification should only start after a thorough test of the algorithm to avoid bothering with an unsolvable verification problem. But then if verification succeeds, it is guaranteed that the algorithm solves the given problem for all inputs it can be applied to.

To verify that an algorithm satisfies a specification means to perform mathematical proofs. In other words: Verification is (also) Theorem Proving. Often students of Computer Science become acquainted with verification in beginner courses when they learn to prove a statement about a while-
loop computing the factorial function or things there like. Usually these are time consuming (and error-prone) pencil-and-paper exercises involving a lot of tedious proof steps to learn some principles of verification. However, by the complexity of non-trivial algorithms one can hardly go beyond the simple while-loops investigated for verification in the beginner courses. If more complex algorithms should be verified, a verification tool is needed which relieves the user from many of the time consuming and error-prone calculations thus giving time to reason about the program and its specification.

This document gives an introduction to the verification tool \texttt{veriFun} (abbreviating Verification of Functional Programs). It is assumed that a reader has some basic knowledge about programming and formal logic, as e.g. taught in undergraduate courses of computer science. For instance, one should be capable to write a recursively defined procedure and also to write a logic formula involving connectives and quantifiers in order to follow the presentation.

This tutorial demonstrates the operation and the basic features of \texttt{veriFun}. As verification cannot be completely mechanized, user interaction is required from time to time to help the system to find a proof. Therefore some hints and “recipes”, what to do if the system fails in finding a proof, are also provided. However, commands and proof rules are illustrated only to the extend necessary to start working with \texttt{veriFun}. All details about the system and how to operate it can be found in the \texttt{veriFun User Guide}. This document also contains further explanations, illustrations and examples, e.g. about predefined program elements, user defined data structures, user guided termination proofs, etc.

This tutorial uses simple examples to guide a novice to the first steps in working with \texttt{veriFun}. It is strongly recommended to follow the explanations with your own installation of the system having also the \texttt{veriFun User Guide} at hand. The system, installation information and the \texttt{veriFun User Guide} can be obtained from

\url{http://www.informatik.tu-darmstadt.de/pm/verifun/}. 

3
Since this document underlies a permanent revision, this web-page should be also visited to see whether a more recent version exists. Questions, comments and critique may be directed to

verifun@informatik.tu-darmstadt.de.

This tutorial is based on VeriFun version 2.5.6. The system may behave different for the examples presented here, if other system versions are used.

### Proving the Associativity of plus

We start with a very simple verification problem. First we consider how to solve the problem using “pencil and paper”, and then how to use VeriFun for computing a solution.

Let us consider the following procedure:

```plaintext
function plus(x,y:nat):nat <=
   if x=0
      then y
      else succ(plus(pred(x),y))
   fi .
```

This procedure obviously computes the addition of a pair of natural numbers. We want to prove that `plus` computes an associative operation, i.e. we want to prove

\[ \forall x,y,z:nat \; plus(plus(x,y),z) = plus(x,plus(y,z)) \]

which means that `plus(x,plus(y,z))=plus(plus(x,y),z)` should hold for all natural numbers `x, y` and `z`.  

4
A hand crafted proof

Since statement (1) is a proposition about natural numbers (where the successor function is written as \texttt{succ}) and a recursively defined procedure, we may prove this statement by induction.

Recall that by the induction principle, a statement of form

\[
(2) \ \forall x: \text{nat} \ \phi[x]
\]

holds, if

\[
(3) \ \phi[0]
\]

and

\[
(4) \ \forall x: \text{nat} \ \phi[x] \rightarrow \phi[\text{succ}(x)]
\]

can be proved. Formula (3) is called the \textit{base case} or the \textit{base formula}, and formula (4) is called the \textit{step case} or \textit{step formula} of the induction, where \(\phi[x]\) denotes the \textit{induction hypothesis} and \(\phi[\text{succ}(x)]\) is the \textit{induction conclusion} of the step formula. This form of induction, known as \textit{Peano Induction}, is also called \textit{constructor induction}, because we use the successor operation \texttt{succ} to define the natural numbers, i.e. each natural number different from 0 can be “constructed” from some other natural numbers using \texttt{succ} only.

However, it is advantageous for our purposes to use another variant of Peano Induction, viz. the so-called \textit{destructor induction}. Here the predecessor function \texttt{pred} is used instead of \texttt{succ} to formulate the induction principle. This form of Peano Induction uses the fact that each natural number can be obtained from some other natural numbers using the “destructor” \texttt{pred} only. Using destructor induction, the base case reads as

\[
(5) \ \forall x: \text{nat} \ x=0 \rightarrow \phi[x]
\]

and the step case is written as

\[
(6) \ \forall x: \text{nat} \ x\neq 0 \land \phi[\text{pred}(x)] \rightarrow \phi[x] .
\]
Now if we define

\[(7) \ \phi[x] := \forall y,z: \text{nat} \ \\
\text{plus(plus(x,y),z)} = \text{plus(x,plus(y,z))} \]

in order to verify (1), by destructor induction it is enough to prove the base case (5) now reading as

\[(8) \ \forall x,y,z: \text{nat} \ x=0 \rightarrow \ \\
\text{plus(plus(x,y),z)} = \text{plus(x,plus(y,z))} \]

as well as the step case (6) given as

\[(9) \ \forall x,y,z: \text{nat} \ x\neq 0 \land \ \\
(\forall y^*,z^*: \text{nat} \ \text{plus(plus(pred(x),y^*),z^*)} = \ \\
\text{plus(pred(x),plus(y^*,z^*))} \ \\
\rightarrow \text{plus(plus(x,y),z)} = \text{plus(x,plus(y,z))}). \]

Both proofs are straightforward: Using the definition of plus, we may replace in the base case (8) the right hand side \(\text{plus(x,plus(y,z))}\) of the equation by \(\text{plus(y,z)}\) and \(\text{plus(x,y)}\) in the left hand side \(\text{plus(plus(x,y),z)}\) by \(y\). Both replacements yield the modified base formula

\[(10) \ \forall x,y,z: \text{nat} \ x=0 \rightarrow \text{plus(y,z)} = \text{plus(y,z)} \]

which obviously is true by the reflexivity of equality \(=\). The proof of the step case proceeds quite similar to the base case. Also here the definition of plus is used to replace some terms in the induction conclusion: \(\text{plus(x,plus(y,z))}\) is replaced by \(\text{succ(plus(pred(x),plus(y,z))}\) and \(\text{plus(plus(x,y),z)}\) is replaced by \(\text{plus(succ(plus(pred(x),y))),z}\) in a first step and then by \(\text{succ(plus(plus(pred(x),y)),z)}\) in a second step, where we use the facts that \(\text{succ(...)} \neq 0\) and \(\text{pred(succ(...))} = \ldots\) in the latter step. This yields the modified induction conclusion

\[(11) \ \forall x,y,z: \text{nat} \ x \neq 0 \land \ldots \rightarrow \ \\
\text{succ(plus(plus(pred(x),y))),z}) = \ \\
\text{succ(plus(pred(x),plus(y,z))}). \]
Now we may use the induction hypothesis

\[(12) \quad \forall y^*, z^*: \mathbb{nat} \quad \text{plus}(\text{plus}(\text{pred}(x), y^*), z^*) = \text{plus}(\text{pred}(x), \text{plus}(y^*, z^*))\]

to replace \(\text{plus}(\text{plus}(\text{pred}(x), y), z)\) with \(\text{plus}(\text{pred}(x), \text{plus}(y, z))\), and then the modified induction conclusion looks like

\[(13) \quad \forall x, y, z: \mathbb{nat} \quad x \neq 0 \land \ldots \rightarrow \text{succ}(\text{plus}(\text{pred}(x), \text{plus}(y, z))) = \text{succ}(\text{plus}(\text{pred}(x), \text{plus}(y, z)))\]

which also obviously is true by the reflexivity of equality.

So, summing up, how did we prove statement (1)? First we decided to use induction, which is not such an ingenious idea, because \text{plus} is recursively defined and proofs about recursively defined procedures usually require induction. Then we used the definition of \text{plus} to replace some terms in the base and in the step formulas with other terms. Formally, these replacements are proof steps justified by some obvious facts about 0, \text{pred}, and \text{succ} and by the equalities given with the definition of \text{plus}. This proved the base case and made the induction hypothesis applicable in the step case, yielding also here a proof after an additional proof step.

**A system generated proof**

We now turn to the use of \texttt{VeriFun} for proving statement (1). Before proceeding, it is recommended to start the system in order to follow the subsequent instructions and explanations using a running system. Also the \texttt{VeriFun User Guide} should be at hand as we do not explain each detail of operating the system here. In particular, now continue with reading \texttt{File\About File} and about the commands \textbf{Open, Save, Save as} and \textbf{Exit} of the \texttt{File Menu} in the \texttt{VeriFun User Guide} before proceeding, to learn how to save and reload intermediate work.

After starting, \texttt{VeriFun} comes up with the \texttt{Main System Window}. 
The Main System Window is separated into 3 subwindows, called
- the Program Window, which displays the actual program, i.e. the system and the user defined data structures, the procedures operating on them and the statements about the data structures and procedures,
- the Proof Window, which displays proofs of the statements, and
- the Evaluation Window, which displays so-called symbolic evaluations, i.e. a special kind of proofs.

Our first action is to provide `veriFun` with the definition of `plus`:
- select the root folder `Program` in the Program Window,
- choose Insert from the Program Menu or alternatively from the context menu,
- in the Insert dialog opened by the command type in the definition of `plus`, and
- quit by pushing OK.
In the VeriFun User Guide see About VeriFun\Functional Programs and About VeriFun\Syntax to become familiar with the syntax of the system’s input language, and see Program\Insert to learn about the Insert command.

If something was misspelled, the system reacts with an error message, and otherwise accepts the definition yielding an updated actual program:

To view a pretty print of an element of the actual program, say plus,
⇒ select plus in the Program Window,
⇒ choose Open Program Viewer from the Window Menu,
⇒ or alternatively choose Properties from the Program Menu or from the context menu.

See Window\About Window and the commands of the Window Menu in the VeriFun User Guide to learn about the Program Viewer and the Program Property Windows.
The procedure `plus` is displayed in green color in the Program Window. This means that the system has proved the termination of `plus`. Generally, each element of the actual program is assigned a program state indicating the progress of the verification. Program states are displayed by colors in the Program Window, were e.g. all program elements having state verified are displayed green. You may look up the coloring assignments of the program states by opening the Legend Viewer from the Window Menu, and you may learn about the program states by visiting Program\About Program\Status of Program Elements in the VeriFun User Guide.

Since each procedure has state verified if its termination has been proved, the system displays `plus` in green. Consequently, a non-terminating procedure must be displayed with another color. We demonstrate this with another input which also helps to learn about some other commands of the Program Menu:
⇒ select plus in the Program Window,
⇒ choose Duplicate from the Program Menu or alternatively from the context menu,
⇒ in the Insert dialog opened by the command replace both occurrences of plus by a new name, say wrong.plus,
⇒ replace pred(x) in the recursive call of wrong.plus by x, and
⇒ quit by pushing OK.

The system also accepts the definition of wrong.plus. However, this program element is displayed in blue (denoting the program state ready) because the system (fortunately) failed to prove the termination of wrong.plus. In order to debug wrong.plus

⇒ select wrong.plus in the Program Window,
⇒ choose Modify from the Program Menu or alternatively from the context menu,
⇒ in the Insert dialog opened by the command replace x in the recursive call of wrong.plus by pred(x), and
⇒ quit by pushing OK.

Since we do not need wrong.plus in the sequel, it should be deleted:

⇒ select wrong.plus in the Program Window,
⇒ choose Delete from the Program Menu or alternatively from the context menu, and
⇒ confirm the system’s deletion warning.

We now resume our work on the associativity of plus by providing the system with our challenge:

⇒ select plus or the root-folder Program in the Program Window,
⇒ choose Insert from the Program Menu or alternatively from the context menu,
⇒ in the Insert dialog opened by the command type in lemma

plus_associative <= all x,y,z:nat plus(plus(x,y),z) = plus(x,plus(y,z)), and
⇒ quit by pushing OK.

This input yields the updated Main System Window
displaying `plus_associative` in program state `ready` (blue). This means that the system does not hold a proof for `plus_associative`, but it is ready to work on this lemma. Therefore, the system has generated an initial prooftree for `plus_associative`. Generally, the prooftree of a lemma is displayed in the system’s `Proof Window`.

To see the initial prooftree of `plus_associative`

> select `plus_associative` in the `Program Window`, and
> choose `Proof` from the `Program Menu` or alternatively from the context menu.

The initial prooftree consists of a root node only, which is displayed in `blue` color in the `Proof Window`. This means that the system does not hold a proof for the root node. Generally, each node of a prooftree, also called a `proofnode`, is assigned a `proof state` indicating the progress of proving the proofnode. Like program states, also proof states are displayed by colors, were e.g. all proofnodes having state `proved` are displayed in green color,
and a blue color denotes the proof state *unproved*. You may look up the coloring of the proof states by opening the *Legend Viewer* from the *Window Menu* and you may learn about the proof states by visiting *Proof\About Proof\Status of Proofnodes* in the *VeriFun User Guide*.

Since all required definitions are provided, we finally may instruct *VeriFun* to prove *plus_associative*:

⇒ select *plus_associative* in the *Program Window*, and

⇒ choose *Verify* from the *Program Menu* or from the context menu or alternatively double-click *plus_associative* in the *Program Window*.

Now the system computes a proof for *plus_associative* by extending the proof tree of the lemma step by step finally yielding:
The computed prooftree of `plus_associative` corresponds to the hand crafted proof of the former section in the following way: The root node corresponds to statement (1) to which the proof rule `Induction` has been applied yielding two successor nodes. The first successor node corresponds to the base formula (8) of the induction, and the other successor node corresponds to the step formula (9). To both formulas the system has applied the proof rule `Simplification` yielding the successor nodes `true` in either case, where "Simplification* using all z*,y*:nat p..." indicates that the induction hypothesis (12) has been used when simplifying the step formula. Since both formulas are simplified to `true`, each formula and in turn the main statement (1) is proved.

To explore the generated prooftree in detail, select a proofnode and open the `Proof Viewer` using the appropriate command from the `Window Menu` (or open the proofnode’s `Proof Property Window` using the appropriate command from the `Proof` or from the context menu). If the proofnode corresponding to the step formula is selected, the induction conclusion is
displayed by choosing the **Goal** tab, the hypothesis $x \neq 0$ is displayed when choosing the **Hypotheses** tab, and the induction hypothesis is displayed by choosing the **Induction Hypothesis** tab.

The proofs of the base and the step formula obtained with the application of the *Simplification* proof rule are computed by **VeriFun's Symbolic Evaluator**, i.e. an automated theorem prover. See **Evaluation\About Evaluation** in the **VeriFun User Guide** to learn something about the **Symbolic Evaluator**.

To explore a **symbolic evaluation**, i.e. a proof computed by the **Symbolic Evaluator**,  
⇒ select the proofnode to which *Simplification* has been applied, e.g. the proofnode corresponding to the step case in our example, and  
⇒ choose **Evaluation** from the **Proof Menu** or alternatively from the context menu.  

This command displays the symbolic evaluation in the **Evaluation Window** and also opens the **Evaluation Viewer** to browse through the symbolic evaluation step by step.
See *Proof\ Evaluation* in the *VeriFun User Guide* to learn how to browse through a symbolic evaluation in the *Evaluation Viewer*, and compare the symbolic evaluations of the step and the base cases with the manually developed proofs from the former section.
Proving the Commutativity of plus

The verification of plus_associative was completely automated, i.e. we only had to tell the system which problem to solve and then pushed the button for the Verify command. However, verifications are not always that easy. Let us try to prove the commutativity of plus and see what happens:

⇒ select the root folder Program in the Program Window,
⇒ choose Insert from the Program Menu or from the context menu,
⇒ in the Insert dialog opened by the command type in lemma plus_commutative <= all x,y:nat plus(x,y) = plus(y,x),
⇒ quit insertion by pushing OK, and
⇒ choose Verify from the Program Menu or from the context menu or double-click plus_commutative in the Program Window.

The system now tries to compute a proof for plus_commutative by extending the proof tree of the lemma step by step finally yielding:
However, different to the associativity case, the verification gets stuck: The lemma remains in state \textit{ready} and all proofnodes have state \textit{unproved}. Let us explore the prooftree to get some idea why the verification of \texttt{plus_commutative} failed.

We see in the \textit{Proof Window} that the system applied the \textit{Induction} proof rule to the root node and then applied \textit{Simplification} to the base and to the step formula. Since for some reasons the induction hypothesis was not used when simplifying the step formula, the system brought the induction hypothesis into play with a further proof rule, viz. \textit{Apply Equation}.

Using the \textit{Proof Viewer}, we now inspect the leaf of the prooftree belonging to the step case, and – using the \textit{Induction Hypotheses} tab – see that the system did an induction upon \( y \). However, by the symmetry in \texttt{plus_commutative}, an induction upon \( y \) must be as good as an induction upon \( x \), so the choice of the induction variable \( y \) could not cause the failure here.

Choosing the \textit{Goal} tab in the \textit{Proof Viewer}, we see the simplified induction conclusion as 
\[
\forall x,y : \text{nat} \ y \neq 0 \land \\
(\forall x^* : \text{nat} \ \text{plus}(x^*, \text{pred}(y)) = \text{plus}(\text{pred}(y), x^*)) \\
\rightarrow \text{plus}(x, y) = \text{succ}(\text{plus}(x, \text{pred}(y))).
\]

Since the arguments of \texttt{plus} has been swapped in the modified induction conclusion, the induction hypothesis must have been already used when forming (14). Hence we may focus on the stronger statement

\[
\forall x,y : \text{nat} \ y \neq 0 \\
\rightarrow \text{plus}(x, y) = \text{succ}(\text{plus}(x, \text{pred}(y)))
\]

obtained by ignoring the induction hypothesis. This statement obviously is \textit{true}, but \texttt{VeriFun} fails to recognize this because it cannot be proved.
without induction. A remedy to this problem is to formulate statement (15) as an auxiliary lemma, verify it and then resume the verification of \texttt{plus\_commutative}, now having available the auxiliary lemma as an additional truth about \texttt{plus}.

So here is how to proceed:

\begin{itemize}
  \item select the root folder \texttt{Program} in the \texttt{Program Window},
  \item choose \texttt{Insert} from the \texttt{Program Menu} or from the context menu,
  \item in the \texttt{Insert dialog} type in \texttt{lemma plus\_right\_succ <= all x,y:nat \ if(y=0, true, plus(x, y) = succ(plus(x, pred(y))))},
  \item quit insertion by pushing \texttt{OK},
  \item choose \texttt{Verify} from the \texttt{Program Menu} or from the context menu or alternatively double-click \texttt{plus\_right\_succ} in the \texttt{Program Window},
  \item select \texttt{plus\_commutative} in the \texttt{Program Window} and choose \texttt{Proof} from the \texttt{Program Menu} or from the context menu or alternatively double-click \texttt{plus\_commutative} in the \texttt{Program Window},
  \item in the \texttt{Proof Window} select the leaf of the proof-tree’s subtree belonging to the step case,
  \item choose \texttt{Proof Rules} in the \texttt{Proof Menu} or from the context menu, and
  \item click \texttt{Simplification} in the \texttt{Proof Rules} submenu opened by \texttt{Proof Rules}.
\end{itemize}

\texttt{VeriFun} now completes the proof of the step case, as - using the verified auxiliary lemma \texttt{plus\_right\_succ} - the system can simplify the step formula to \texttt{true}.

So let us continue with the base case. Using the \texttt{Proof Viewer}, we do a similar analysis as for the step case and find that also here an auxiliary lemma, viz.

\begin{verbatim}
lemma plus_right_zero <= all x,y:nat
  if(y=0,plus(x,y)=x,true),
\end{verbatim}

is needed. We therefore continue as in the step case and finally succeed in proving the commutativity of \texttt{plus}.
The proof of plus_commutative required user interaction, and this is generally needed in verification. There are two types of interaction, viz.

(i) the formulation of useful auxiliary lemmas, as for instance plus_right_succ and plus_right_zero, and

(ii) telling the system to use a certain proof rule, as e.g. calling for Simplification to complete the proof of plus_commutative.

Finding the “right” lemma and applying a “good” rule are two of the main challenges for the user in verification, and both can be considerably hard. While interactions of type (i) may require a deep insight into the program and its specification, interactions of type (ii) need some competence in logic and in proof engineering. In both cases a thorough analysis of the situation is crucial to find out how to proceed. However, insight and competence quite often grow with experience.
Developing Prooftrees

As we have already seen, VeriFun creates a prooftree when called to do so by the Verify command of the Program Menu. See Proof\About Proof\Prooftrees and Proof\About Proof\Developing Prooftrees in the VeriFun User Guide for some formal background and details about prooftrees.

Starting with the initial prooftree, a prooftree is developed by applying a proof rule to some of its leaves. This process is continued until the prooftree is closed, i.e. each leaf of the prooftree is labeled with true. VeriFun provides a set of 13 proof rules which can be applied to the leaves of a prooftree in order to become a closed one. The situation is like in a game, e.g. Chess, where one applies a certain rule of the game in order to improve the situation towards a win. However, the art is to decide when to apply which rule in which place in order to win the game, and a similar problem must be solved when building a closed prooftree.

VeriFun supports the user by trying to compute a closed prooftree, as it did e.g. for plus_associative. However, the system cannot be perfect and may fail to succeed, as it did e.g. for plus_commutative. In such a situation, the user must step in to guide the system for the continuation of the proof.

If the failure is caused by a missing lemma, an appropriate lemma - based on the analysis of the current proof goal and the available hypotheses and induction hypotheses - must be inserted into the actual program. Otherwise it is the user’s responsibility to modify the prooftree created by VeriFun. To this effect, the user selects an open leave in the prooftree, i.e. a leave not labeled with true, and then applies a proof rule to that leave, possibly after some unwanted branch of the prooftree has been removed. After a user selected proof rule has been applied, VeriFun tries to take over control in order to develop the prooftree further.
A branch of a prooftree is removed using the **Prune** command from the *Proof Menu*. A proof rule is selected from the *Proof Rules* submenu of the *Proof Menu*.

### Using Proof Rules

The *Proof Rules* submenu lists 12 of the 13 proof rules, as the proof rule **Inconsistency** can be called by the system only. A proof rule is applied to the proofnode which is selected in the *Proof Window*. Each proofnode is labeled with a so-called *sequent* of the form

\[(16) \ h_1, \ldots, h_n, \ \text{all} \ldots i h_1, \ldots, \text{all} \ldots i h_1 \vdash \text{goal}\]

where the expressions \(h_1\) are the hypotheses, the expressions \(\text{all} \ldots i h_k\) are the induction hypotheses, and the expression \(\text{goal}\) is the goalterm of the sequent. The components of a sequent can be seen by selecting the proofnode in the *Proof Window* and then choosing the **Hypotheses**, the **Induction Hypotheses** or the **Goal** tab in the *Proof Viewer* or in the proofnode’s *Proof Property Window*.

The application of a proof rule to a certain proofnode creates one or more successor nodes, each of which inherits a modification of the proofnode’s sequent in which the hypotheses, the induction hypotheses or the goalterm has been modified.

A sequent is *true*, if the truth of the hypotheses and the induction hypotheses entails the truth of the goalterm. **VERIFUN**’s proof rules are *sound* in the sense, that the truth of all sequents of the successor nodes entails the truth of the sequent belonging to the proofnode to which the proof rule has been applied.

Subsequently, we illustrate the use of the proof rules by examples. See *Proof\Proof Rules* in the **VERIFUN User Guide** to learn the details about the application and the effect of each proof rule.
Computed Proof Rules

For proving a sequent, \texttt{\textipa{veriFun} uses an automated theorem prover, called the Symbolic Evaluator. The Symbolic Evaluator is based on the so-called Evaluation Calculus, i.e. a calculus consisting of a set of inference rules, called the evaluation rules. Starting with the goalterm of a sequent, the Symbolic Evaluator applies the first applicable evaluation rule to this goalterm, then applies the first applicable evaluation rule to the result of the recent evaluation step and so on, until eventually a goalterm is obtained to which no further evaluation rule can be applied. The list of goalterms obtained thereby defines a deduction in the evaluation calculus, called the symbolic evaluation of the (initial) goalterm.

When evaluating the goalterm of a sequent symbolically, the Symbolic Evaluator may use the hypotheses and the induction hypotheses of the sequent and may also use the verified lemmas and the definitions of the data structures and verified procedures of the actual program.

The Symbolic Evaluator is a completely automated tool on which the \texttt{\textipa{veriFun} user has no direct influence, except to stop or to cancel a symbolic evaluation. However, a user may indirectly influence the computation of a symbolic evaluation by providing or not providing verified lemmas.

The Symbolic Evaluator is called via the computed proof rules of the Proof Rules submenu, viz.

- Simplification,
- Weak Simplification,
- Normalization, and
- Weak Normalization

(and the command Refute from the Proof Menu) each of which invoke an incarnation of the Symbolic Evaluator parameterized for the user’s need.

A proofnode to which a computed proof rule has been applied is called a computed proofnode, and is assigned a symbolic evaluation of the proof-
node’s goalterm. Such a symbolic evaluation can be explored using the Evaluation command from the Proof Menu.

Induction

By the Induction rule, the system is instructed to prove a statement by induction. This rule can only be applied to a pure proofnode, i.e. a proofnode (like the root node of a lemma’s initial prooftree) possessing neither hypotheses nor induction hypotheses. Usually several induction axioms are available in the system and there may be also several variables to induct upon in the statement. Therefore the user has to decide which induction axiom to use and which variable(s) to induct upon when applying Induction to a proofnode.

VeriFun computes induction axioms from the definition of the data structures and the verified procedures. The system represents induction axioms in a compact form by so-called relation descriptions. A relation description lists the step cases of an induction axiom by stipulating the hypotheses of each step case and how the induction hypotheses of this step case are obtained. For instance, the relation description computed from the procedure plus is given as

\[(17) \quad \langle\langle \text{if}(x=0,\text{false},\text{true})\rangle, \{x/\text{pred}(x)\} \rangle\]

and represents the destructive Peano Induction used when we proved plus_associative. Each relation description has a set of recursion variables, e.g. \(\{x\}\) in the example, denoting the variables “to induct upon”. The relation descriptions of a procedure or a data structure can be viewed in the Program Viewer or in the Program Property Window of the program element when choosing the Termination tab and selecting a set of recursion variables in the Recursion Variables field.

However, destructive Peano Induction is not the only (destructive) form of induction. Let us insert, for instance, the procedure
function half(x:nat):nat <=
  if x=0
    then 0
  else if pred(x)=0
    then 0
    else succ(half(pred(pred(x))))
  fi
fi

and the lemma

lemma half_plus <= all n:nat half(plus(n,n))=n

into the actual program. From the procedure half, VeriFun computes the relation description

(18) \{\{if(x=0,false,true)\},\{x/pred(pred(x))\}\}\}

which represents an induction axiom with the base case

(19) \forall x:nat x=0 \rightarrow \phi[x]

and the step case

(20) \forall x:nat x\neq0 \land \phi[pred(pred(x))] \rightarrow \phi[x]

for proving a statement \forall x:nat \phi[x]. Now if we aim to verify half_plus, it has to be decided whether to use the induction axiom stemming from plus or the induction axiom computed from half. This decision is made when using the Induction rule:

\Rightarrow select half_plus in the Program Window and choose Proof from the Program Menu or from the context menu,
\Rightarrow choose Proof Rules in the Proof Menu or from the context menu,
\Rightarrow click Induction in the Proof Rules submenu opened by Proof Rules,
\Rightarrow open the folder labeled with half in the input dialog opened by Induction,
\Rightarrow open the folder labeled with \{x\},
\Rightarrow select the leaf labeled with \{x/n\}, and
\Rightarrow push OK to apply the induction axiom represented by the relation description displayed in the lower part of the Induction dialog to the root node.
The system now extends the initial prooftree of `half_plus` by the base and the step case of the induction but fails to complete the proof in the step case.

We illustrate what we did in the Induction dialog before we discuss how to continue with the proof of `half_plus`: By opening a folder labeled with the name of a procedure (or a data structure), a list of recursion variable sets is displayed, each of which corresponds to a relation description of the procedure (or the data structure). Since only one relation description is associated with `half`, a singleton list is displayed in our example. By opening a folder labeled with a recursion variable set, all possible assignments of the variables of the statement to the recursion variables of the relation description are displayed. As `half_plus` has only one variable, viz. `n`, and also the selected relation description has only one recursion variable, viz. `x`, only one assignment, viz. `{x/n}`, is
displayed. This variable assignment now is used by the system to adapt the relation description of half to the variables of the statement, yielding the relation description (displayed in the lower part of the Induction dialog) which is actually used.

Since the half–induction was not successful, let us see what happens if we try the induction stemming from plus:

⇒ select the root node of the prooftree of half_plus in the Proof Window and choose Prune from the Proof Menu or from the context menu,
⇒ choose Proof Rules in the Proof Menu or from the context menu,
⇒ click Induction in the Proof Rules submenu opened by Proof Rules,
⇒ open the folder labeled with plus in the input dialog opened by Induction,
⇒ open the folder labeled with \{x\},
⇒ select the leaf labeled with \{x/n\}, and
⇒ push OK to apply the induction axiom represented by the relation description displayed in the lower part of the Induction dialog to the root node.

Now the system extends the initial prooftree of half_plus by the base and the step case corresponding to the plus-induction and, fortunately, succeeds in verifying half_plus. So the plus-induction is the right choice here, and this induction is also used by the system when calling the Verify command from the Program Menu instead.

We continue with another example. Let us insert the procedure

```plaintext
function foo(x,y:nat):nat <=
if x=0
  then half(y)
else if y=0
  then x
else if pred(y)=0
  then x
else succ(succ(foo(pred(x),
  pred(pred(y))))))
fi
fi
```
and the lemma

\[
\begin{align*}
\text{lemma } \text{foo\_plus\_half} & \\
& \iff \\
& \text{all } n, m : \text{nat} \ \text{foo}(n, m) = \text{plus}(n, \text{half}(m))
\end{align*}
\]

into the actual program. From the procedure \text{foo}, \texttt{\text{veriFun}} computes the relation descriptions

\[(21) \ \{\langle x=0, false, true \rangle \}, \{x/\text{pred}(x)\}\} \text{ and } \{\langle y=0, false, true \rangle \}, \{y/\text{pred(pred}(y))\}\}
\]

the first one coincidentally representing the same induction as the plus-induction and the second one coincidentally representing the same induction as the half-induction. Now if \texttt{Induction} is applied to the root node of the initial proof tree of \texttt{foo\_plus\_half} and the folder labeled with \texttt{foo} is opened, two recursion variable sets - one labeled with \{x\} and the other labeled with \{y\} - are displayed, each corresponding to a relation description of \texttt{foo}. If we open one of the recursion variable sets, say the one labeled with \{y\}, two variable assignments, viz. \{y/m\} and \{y/n\} are displayed. The selection of \{y/m\} will yield an induction with the step case

\[(23) \ \forall n, m : \text{nat} \ m \neq 0 \land \forall n^* : \text{nat} \ [\phi[n^*,\text{pred(pred(m))}] \rightarrow \phi[n,m] \] 

while the selection of \{y/n\} would yield an induction with the step case

\[(24) \ \forall n, m : \text{nat} \ n \neq 0 \land \forall m^* : \text{nat} \ [\phi[\text{pred(pred(n))},m^*] \rightarrow \phi[n,m] \].

If we open the recursion variable set labeled with \{x\} instead, we must choose between an induction with the step case

\[(25) \ \forall n, m : \text{nat} \ m \neq 0 \land \forall n^* : \text{nat} \ [\phi[n^*,\text{pred(m)}] \rightarrow \phi[n,m] \] 

and an induction with the step case

\[(26) \ \forall n, m : \text{nat} \ n \neq 0 \land \forall m^* : \text{nat} \ [\phi[\text{pred(n)},m^*] \rightarrow \phi[n,m] \].

The same alternatives for the variable assignments exist if we inspect the folder labeled with \texttt{half} or the folder labeled with \texttt{plus} respectively, as it must be determined in both cases which variable(s) to induct upon.
If the **Verify** command is applied to a lemma, `veriFun` decides which proof rule to apply to the root node of the lemma’s initial prooftree. In case of **Induction**, the system determines a relation description and an appropriate variable assignment for the recursion variables, creates the successor nodes for the base and the step cases, and then applies further proof rules to the successor nodes. In case of `foo_plus_half`, the system chooses the `foo`-induction with the step case (23) but unfortunately fails to complete the proof.

**Case Analysis**

By the **Case Analysis** rule, the system is instructed to prove a statement by cases. Depending on the `caseterm` provided, `veriFun` performs a propositional or a structural case analysis. We may use the **Case Analysis** rule to complete the proof of `foo_plus_half` in the following way:

- open the prooftree of `foo_plus_half` in the **Proof Window**,
- select the successor node of the unproved proofnode labeled with **Simplification** in the **Proof Window** and choose **Prune** from the **Proof Menu** or from the context menu,
- choose **Proof Rules** in the **Proof Menu** or from the context menu,
- click **Case Analysis** in the **Proof Rules** submenu opened by **Proof Rules**,
⇒ insert the caseterm \( plus(n,\text{half}(\text{pred}(\text{pred}(m)))) = \text{succ}(\text{plus}(\text{pred}(n),\text{half}(\text{pred}(\text{pred}(m)))) \) in the input dialog of Case Analysis, and
⇒ push OK to apply the case analysis given by the caseterm to the selected proofnode.

Now VeriFun extends the prooftree by two successor nodes, one for the case if the caseterm is true and the other for the case if the caseterm is false. Then (using the induction hypothesis) the system completes the proof by applying the Simplification rule to each of the proofnodes just created.
Use Lemma

By the Use Lemma rule, the system is instructed to continue a proof by using an instance of a lemma or an induction hypothesis. When calling this proof rule, the user must

1. select one of the available lemmas or induction hypotheses,
2. provide a variable assignment for the universally quantified variables to form an instance of the selected lemma or induction hypothesis, and
3. determine the position where the selected instance should be applied in the goalterm.

For instance, we may try to complete the proof of foo_plus_half by telling the system to use a certain instance of the induction hypothesis:

⇒ open the prooftree of foo_plus_half in the Proof Window,
⇒ select the successor node of the unproved proofnode labeled with Simplification* in the Proof Window and choose Prune from the Proof Menu or from the context menu,
⇒ choose Proof Rules in the Proof Menu or from the context menu,
⇒ click Use Lemma in the Proof Rules submenu opened by Proof Rules,
⇒ select IH from the list of Available Lemmas in the input dialog opened by Use Lemma,
⇒ in the substitution field of the input dialog assign an appropriate term, viz. pred(n), to the universally quantified variable n* of the induction hypothesis,
⇒ in the Available Positions field of the input dialog select the position in the goalterm where the instance of the induction hypothesis should be applied, e.g. (3,3), and
⇒ push OK to apply the instance of the induction hypothesis given by the variable assignment to the selected proofnode.

Now veriFun extends the prooftree by a successor node and completes the proof by applying the Simplification rule to the proofnode just created.
While the Symbolic Evaluator only uses verified lemmas, a user may also select non-verified lemmas with the Use Lemma proof rule. However, a closed prooftree build with a non-verified lemma does not represent a proof of the lemma to which the prooftree belongs. In such a case, this lemma is assigned the program state developed, which is changed to verified only if each lemma used in the prooftree gets the program state verified.
Unfold Procedure

By the *Unfold Procedure* rule, the system is instructed to replace a procedure call with the body of the procedure, such that the formal parameters in the procedure body are replaced by the actual parameters of the procedure call. When calling *Unfold Procedure*, the user must determine which procedure call in the goalterm to unfold.

For instance, we may try to guide *VeriFun* to a proof of `foo_plus_half` in the following way:
⇒ open the prooftree of foo_plus_half in the Proof Window,
⇒ select the successor node of the unproved proofnode labeled with Simplification* in the Proof Window and choose Prune from the Proof Menu or from the context menu,
⇒ choose Proof Rules in the Proof Menu or from the context menu,
⇒ click Unfold Procedure in the Proof Rules submenu opened by Proof Rules,
⇒ in the input dialog opened by Unfold Procedure open the folder labeled with plus to tell the system that a plus-call shall be unfolded,
⇒ select (3,3,2) to tell the system which plus-call shall be unfolded, and
⇒ push OK to unfold the procedure call just selected in the goalterm.

Now VeriFun extends the prooftree by a successor node, and (using the induction hypothesis) completes the proof by applying the Simplification rule to the proofnode just created.

Apply Equation

By the Apply Equation rule, the system is instructed to continue a proof by applying an equation given by a hypothesis or an (instance of) a lemma or an induction hypothesis. When calling this proof rule, the user

1. selects one equational hypothesis or one of the available lemmas or induction hypotheses containing some equation,
2. selects one of the subformulas of the selected lemma or induction hypothesis containing the desired equation,
3. determines the direction in which the equation should be applied,
4. decides which subterm in the goalterm should be replaced by applying the selected equation.

For instance, we may instruct VeriFun to complete the proof of foo_plus_half by using the equation given with the induction hypothesis:

⇒ open the prooftree of foo_plus_half in the Proof Window,
⇒ select the successor node of the unproved proofnode labeled with Simplification* in the Proof Window and choose Prune from the Proof Menu or from the context menu,
⇒ choose **Proof Rules** in the **Proof Menu** or from the context menu,
⇒ click **Apply Equation** in the **Proof Rules submenu** opened by **Proof Rules**,
⇒ open the folder labeled with **Induction Hypotheses** in the input dialog opened by **Apply Equation**,
⇒ in the input dialog open the folder labeled with the (only) induction hypothesis of the sequent,
⇒ in the input dialog open the folder labeled with the (only) equation of the induction hypothesis,
⇒ open the folder labeled with `foo(...)`=`plus(...)` in the input dialog to tell the system that we want to replace a `foo`–term by a `plus`–term (and not vice versa as the system did before),
⇒ determine the subterm of the goalterm to be replaced by selecting the (only) position `(3,3,1,1)`, and
⇒ push **OK** to replace the selected subterm of the goalterm using the selected equation.
Now \texttt{veriFun} extends the proof-tree by a successor node, and then completes the proof by applying the \textit{Simplification} rule to the proofnode just created.

While the \textit{Symbolic Evaluator} only uses \textit{verified} lemmas, a user may also select non-verified lemmas with the \textit{Apply Equation} proof rule. However, a closed proof-tree build with a non-verified lemma does not represent a proof of the lemma to which the proof-tree belongs. In such a case, this lemma is assigned the program state \textit{developed}, which is changed to \textit{verified} only if each lemma used in the proof-tree becomes the program state \textit{verified}.

\textbf{Delete and Move Hypotheses}

Sometimes a statement can be proved by a \textit{nested induction}, i.e. the proof of a base or a step case is proved by a further induction. However, as the \textit{Induction} rule can only be applied to a \textit{pure} proofnode, i.e. a proofnode possessing neither hypotheses nor induction hypotheses, all hypotheses and induction hypotheses of the proofnode sequent must be eliminated before \textit{Induction} can be used. This elimination is performed by the \textit{Delete Hypotheses} and the \textit{Move Hypothesis} proof rule.

Eliminating a hypothesis or an induction hypothesis from a proofnode sequent \texttt{seq} yields a stronger sequent \texttt{seq’}, called a \textit{generalization} of \texttt{seq}, i.e. the truth of \texttt{seq’} entails the truth of \texttt{seq} but (usually) not vice versa. So if \texttt{seq’} can be proved, \texttt{seq} is also proved. But if may happen that \texttt{seq’} is an \textit{over-generalization} of \texttt{seq}, i.e. \texttt{seq’} is false only by the elimination of a hy-
hypotheses or an induction hypothesis, and then nothing can be said about the truth of \( \text{seq} \).

Over-generalizations can be avoided by “saving in the goalterm” the hypotheses and induction hypotheses needed to maintain the truth of the generalization. Hypotheses are “saved” using the Move Hypothesis proof rule. This rule moves a hypothesis from the sequent’s set of hypotheses to the sequent’s goalterm, thus yielding a sequent equivalent to the original one.

Induction hypotheses are “saved” by using the Use Lemma proof rule to insert one or more instances of the induction hypotheses into the goalterm. Different to the saving of hypotheses, the absence of over-generalizations usually cannot be guaranteed by saving induction hypotheses before deletion, because this would require applying Use Lemma to infinitely many instances of each induction hypothesis.

We illustrate the use of both proof rules with a proof of the commutativity of \( \text{plus} \) using nested inductions:

1. select the root folder Program in the Program Window,
2. choose Insert from the Program Menu or from the context menu,
3. in the Insert dialog opened by the command type in \( \text{lemma} \)
   \[ \text{plus_commutative} <= \text{all x,y:nat plus(x,y) = plus(y,x)}, \]
4. quit insertion by pushing OK, and
5. choose Verify from the Program Menu or from the context menu or double-click plus_commutative in the Program Window.

The system now starts an induction proof for plus_commutative, but gets stuck in proving the base and the step case. Exploring the prooftree of plus_commutative, we see in the Proof Window that the system applied the Induction proof rule to the root node and then applied Simplification to the base and to the step formula. Since for some reasons the induction hypothesis was not used by the Symbolic Evaluator when simplifying the step formula, the system brought the induction hypothesis into play with a call of Apply Equation.
Using the *Proof Viewer*, we now inspect the leaf of the prooftree’s subtree belonging to the step case, and - choosing the **Goal** tab - see the simplified goalterm as \( \text{plus}(x,y) = \text{succ}(\text{plus}(x,\text{pred}(y))) \) which should hold under the hypothesis \( \text{if}(y=0, \text{false}, \text{true}) \), displayed when pushing the **Hypotheses** tab. Since the arguments of \( \text{plus} \) has been swapped in the simplified goalterm, the induction hypothesis must have been already used. Hence we may skip it from the proofnode’s sequent but “save” the hypothesis \( \text{if}(y=0, \text{false}, \text{true}) \) in order to continue with a further induction:

1. In the **Proof Window** select the leaf of the prooftree’s subtree belonging to the step case,
2. Choose **Proof Rules** in the **Proof Menu** or from the context menu,
3. Click **Move Hypothesis** in the **Proof Rules** submenu opened by **Proof Rules**,
4. In the input dialog opened by **Move Hypothesis** selected the hypotheses to be moved, which is only one in our example, and
⇒ push OK to move the hypothesis \( \text{if}(y=0, \text{false}, \text{true}) \) from the sequent’s set of hypotheses to the sequent’s goalterm.

Now \texttt{VeriFun} extends the prooftree by a successor node, then applies \textit{Delete Hypotheses} to this successor node to get rid of the induction hypothesis, and, as the resulting proofnode is \textit{pure}, continues with a further application of the \textit{Induction} rule to complete the proof of the step case.

We continue with the base case: Using the \textit{Proof Viewer}, we do a similar analysis as in the step case and find that also here a further induction would do the job. We therefore apply \textbf{Move Hypothesis} to the leaf of the prooftree’s subtree belonging to the base case, thus moving \( y=0 \) from the sequent’s set of hypotheses to the sequent’s goalterm. Also here \texttt{VeriFun} extends the prooftree by a successor node, and then continues with a further application of the \textit{Induction} rule to complete the proof of the base case and consequently of the whole lemma.
Insert Hypotheses

The Insert Hypotheses proof rule implements another form of a case analysis, where, however, the caseterms are inserted into a sequent’s set of hypotheses instead of applying them to the goalterm directly (as the Case Analysis rule does). With respect to the truth of the sequents, it does not matter which rule is applied. However, the system behaves in another way when using Insert Hypotheses instead of Case Analysis.

When evaluating goalterms, the Symbolic Evaluator must decide for each procedure call in the goalterm whether to execute it or not. This decision must be made with great care because the execution of a procedure call may result in a so-called over-evaluation, i.e. a goalterm which is too complex for finding a proof. The decision whether to execute a procedure call is made heuristically and based upon the elements in the sequent’s set of hypotheses. So generally, the more hypotheses are present, the more procedure calls may be executed.

✓eriFun’s heuristic for the execution of procedure calls is conservative in the sense that procedure calls are only executed if the avoidance of over-evaluations is guaranteed. As a consequence, it may happen that a proof needs the execution of a procedure call, but the Symbolic Evaluator fails to do so. In this situation, the user may step in by using the Unfold Procedure proof rule. However, it may also happen that many procedure calls must be unfolded. In such a case, Insert Hypotheses may be used in order to achieve the desired effect with the application of one proof rule only instead of calling Unfold Procedure many times.

Note that the Insert Hypotheses rule bears the risk of over-evaluations. The right use of this proof rule requires some experience with ✓eriFun and is not recommended for beginners.
Testing a Program

Usually a program does not what it is supposed to do, and then one says that the program does not satisfy its specification. A specification is given by a collection of lemmas about some procedures of the program, and at least one of these lemmas does not hold if the program does not satisfy its specification. In such a case, it is a waste of time working on the verification of this lemma, and – as programs can be large and specifications can be complex – it may take several hours before giving up having no result at all.

The answer to this problem is testing. Since testing can prove the presence of bugs, it helps to find errors in the program (or the specification) thus saving the time which would otherwise wasted with unsuccessful attempts of verification.

There are two ways to test a program in **VeriFun**:

- the truth or falsity of a lemma is *computed* after the variables of the lemma have been replaced by certain values in order to see whether the lemma holds at least for the values provided, or
- a procedure is called with certain inputs in order to see whether the results are as expected.

Disproving a Lemma

Testing is performed in **VeriFun** using the *Refute* command from the **Proof Menu**. This command opens an input dialog to assign terms to some variables of a proofnode sequent, and then tries to compute truth or falsity. E.g., we may claim that **plus** is also *idempotent*, but as skeptical users we make some tests before starting verification:

⇒ choose **Insert** from the **Program Menu** or from the context menu,
⇒ in the **Insert dialog** opened by the command type in *lemma*
   `plus_idempotent <= all x:nat plus(x,x) = x`,
⇒ quit insertion by pushing **OK**,
⇒ choose **Proof** from the **Program Menu** or from the context menu,
⇒ choose **Refute** in the **Proof Menu** or from the context menu,
⇒ in the *substitution* field of the input dialog opened by **Refute** assign the term 0 to the only variable \( x \) of the proofnode sequent,
⇒ push **OK** to test the instance of the proofnode sequent obtained by the variable assignment provided in the previous step.

Now **veriFun** extends the prooftree by a successor node with the goal-term **true**, which means that the lemma has successfully passed the test. So let us try another test:

⇒ select the root node of the prooftree of **plus_idempotent** in the **Proof Window** and choose **Prune** from the **Proof Menu** or from the context menu,
⇒ choose **Refute** in the **Proof Menu** or from the context menu,
⇒ in the *substitution* field of the input dialog opened by **Refute** assign the term 1 to the only variable \( x \) of the proofnode sequent,
⇒ push **OK** to test the instance of the proofnode sequent obtained by the variable assignment provided in the previous step.
Again \texttt{VeriFun} extends the prooftree by a successor node, but this time this proofnode’s goalterm equals \texttt{false}. Since the hypotheses set of the proofnode’s sequent is empty, \texttt{plus_idempotent} cannot hold (and consequently this lemma gets the program status \textit{falsified}), because at least one counter example, viz. 1, exists.

Sometimes counter examples are not so easy to spot as in the \texttt{plus_idempotent}-example. Then it often helps to intertwine verification and testing, because with the development of a prooftree, goalterms may be created which support the speculation of a counter example. For instance, suppose that the associativity of \texttt{foo} is claimed, i.e.

\begin{verbatim}
lemma foo_associative <= all u,v,w:nat
foo(foo(u,v),w)=foo(u,foo(v,w))
\end{verbatim}

is inserted into the actual program. But which counter example (if any) to use? Let us start verification and see what happens.

When calling the \texttt{Verify} command from the \textit{Program Menu}, the system starts an induction proof, but fails to complete the proof of the step case. Using the \textit{Proof Viewer} (or a \textit{Proof Property Window}) we inspect the goalterm of the unproved leaf of the prooftree trying to find terms for the variables \texttt{u,v,w} which disprove the leaf.

For instance, we may proceed in the following way: We focus on the goalterm’s subterm

\begin{verbatim}
if(half(pred(pred(w)))=half(pred(half(pred(pred(w))))),
  if(pred(pred(w))=0,
    false,
    if(pred(pred(pred(w)))=0,false,true)),
false))
\end{verbatim}

and assume that we know a variable assignment which satisfies the condition

\begin{verbatim}
half(pred(pred(w)))=half(pred(half(pred(pred(w))))).
\end{verbatim}
Then the condition’s then-part

\[ \text{if}(\text{pred}(\text{pred}(w))=0, \text{false}, \text{if}(\ldots)) \]

must be falsified by the variable assignment, which is granted if the assignment for \( w \) is 0, 1 or 2. If our assignment does not satisfy the subterm’s condition, no further constraint is imposed on the variable assignment, as then the else-part \( \text{false} \) is returned. Next we inspect the conditions leading to the subterm. We see that \( \text{pred}(w)=0 \) must be \( \text{false} \), and therefore 2 is the only remaining candidate for \( w \). Inspecting the conditions leading to \( \text{if}(\text{pred}(w)=0, \text{true}, \text{if}(\ldots)) \) immediately reveals that \( u \) and \( v \) must be 0. So \( u/0, v/0, w/2 \) seems to be the right candidate for a counter example of \( \text{foo_ass ociative} \):
⇒ select the unproved leaf in the prooftree of foo_associative in the Proof Window,
⇒ choose Refute in the Proof Menu or from the context menu,
⇒ in the substitution field of the input dialog opened by Refute assign 0 to the variables u and v and assign 2 to the variable w of the proof-node sequent,
⇒ push OK to test the instance of the proofnode sequent obtained by the variable assignment provided in the previous step.

Now VeriFun extends the prooftree by a successor node with goalterm false (and an empty set of hypotheses) which means that we have found a counter example in fact, and consequently foo_associative is falsified.
Running a Procedure

Using the **Refute** command, we can also “run” a procedure for arbitrary input arguments. To do so, we must build a testbed in form of a lemma first so that then **Refute** can be applied to this lemma. The testbed is build using a so-called *undefined function* which returns a bool-value and has (only) one argument of the sort the procedure to be tested returns. E.g., if we like to test procedures returning nat-terms, like `plus`, `half` or `foo`, we have to

⇒ choose **Insert** from the **Program Menu** or alternatively from the context menu,
⇒ in the *Insert dialog* opened by the command type in `function eval-nat(x:nat):bool <= ?`, and
⇒ quit by pushing **OK**.

To build a testbed for a procedure like e.g. `plus`, we

⇒ choose **Insert** from the **Program Menu** or alternatively from the context menu,
⇒ in the *Insert dialog* opened by the command type in `lemma test_plus <= all x,y:nat eval-nat(plus(x,y))`, and
⇒ quit by pushing **OK**.

To “run” `plus` for certain input arguments, say to compute `plus(28,13)`,

⇒ select **test_plus** in the **Program Window** and choose **Proof** from the **Program Menu** or from the context menu,
⇒ choose **Refute** in the **Proof Menu** or from the context menu,
⇒ in the *substitution* field of the input dialog opened by **Refute** assign 28 to the variable `x` and 13 to the variable `y` of the proofnode sequent,
⇒ push **OK** to “run” `plus` with the arguments given by the variable assignment in the previous step.

Now `verifun` extends the prooftree by a successor node with goalterm `eval-nat(41)`, which means that 41 is the result of running `plus(28,13)`.
Proof Engineering

Although veriFun provides a high degree of automatization, the user must step in from time to time to help the system to find a proof. This needs the creativity and skill of the user, who may save a lot of time and frustration, if he or she keeps the actual program well organized and tackles the problems in a systematic and goal directed way.

For instance, program folders should be created to structure the actual program in a useful way, and the comment feature should be used for notes worth to remember. Also the program items should be assigned meaningful names which help to remember what they denote. If the program grows large and work is resumed after some days, it is hard to directly spot what was meant with a procedure named hack3 and a lemma called hope_this_helps.
However, obeying these rules is not enough to find a proof, and some ideas how to draw the right conclusions from the systems outcome seem worthwhile.

**Keeping Control**

Programs and specifications are formal artefacts used to represent some domain of discourse. E.g. if we try to verify some piece of hardware, say a *byte adder* using carry bits and implemented by shift operations to perform addition of bitstrings, we have to define a data structure representing bitstrings and then write a procedure which represents the operation of the *byte adder*. Next we specify the correctness requirement for the *byte adder* in form of a lemma and then start verification.

Now assume that the proof gets stuck and we wonder how to continue. In this situation, people often start to push some buttons, clicking any proof rule etc. hoping that the system somehow recovers from the failure. Proceeding in this way, the prooftree grows larger and larger so that the user eventually loses control of the whole business.

Only pushing the buttons without having any idea what is really needed is definitely the wrong way to cope with the problem. A formula, like a goalterm in *VeriFun*, denotes some fact about the domain, and the first step is to find out what the goalterm *expresses*. While we map the domain to a formal representation in order to make it available for verification, we have to map the formal representation *back to the domain* in order to think about why a proof gets stuck. Maybe we discover that a goalterm expresses some fact which is impossible in our domain. This would mean that something is wrong either with our formal representation or even with the real *byte adder*. Or we see that some fact about the *byte adder* needs an explicit representation in form of a lemma, which then helps the system to continue with the proof. In simpler cases, it may be enough to apply an equation or a lemma which the *Symbolic Evaluator* failed to use, but its need is obvious after our analysis, etc.
So a key rule in using the system is never to push the buttons without a specific intention, thus always keeping control of the developments.

Analyzing a Goalterm

If the development of a prooftree gets stuck, the user has to decide how to continue. Usually one starts with an analysis of the goalterm of some unproved leaf. Quite often it is useful then to consider an innermost conditional in the goalterm and think about how to prove it. The if-structure of the goalterms helps to see which conditions lead the conditional under investigation and therefore can be assumed true or false. E.g. for a goalterm

\[ \text{if}(A, \text{if}(B, \text{true}, \text{if}(C, D, \text{true})), \text{true}) \]

the innermost conditional \( \text{if}(C, D, \text{true}) \) – representing an implication \( C \rightarrow D \) - is true if either \( C \) is disproved or \( D \) is proved (by using a case analysis, applying an equation or the use of some other proof rule). When disproving \( C \) we may assume that \( A \) is true and \( B \) is false. When proving \( D \) instead, we may assume the truth of \( C \) in addition. If we fail to succeed in both cases, we step back to the second-innermost conditional, viz. \( \text{if}(B, \text{true}, \text{if}(C, D, \text{true})) \) in the example, and try to prove \( B \) assuming the truth of \( A \). If we fail also here, we must step back another time and try to disprove \( A \), now having no additional assumptions left. Working on an innermost conditional quite often helps, because the more nested a conditional is, the more additional assumptions are available.

The truth values in an if-term determine the kind of connective denoted by this term. E.g., \( \text{if}(A, \text{true}, B) \) denotes a disjunction, i.e. \( A \lor B \), while \( \text{if}(A, B, \text{false}) \) denotes a conjunction, i.e. \( A \land B \). See About VeriFun\ Syntax\ Quantifications in the VeriFun User Guide for the representation of connectives by if-terms. For instance, if the goalterm under investigation is given as

\[ \text{if}(A, \text{if}(B, \text{true}, \text{if}(C, D, \text{true})), \text{false}) \]

we have to prove \( \text{if}(B, \text{true}, \text{if}(C, D, \text{true})) \) as we must before, but we must also prove \( A \) in addition.
Decomposing a Goalterm

Sometimes large goalterms are created and it is hard to see where to continue with the proof. In such a case it may help to decompose the goalterm using the Case Analysis proof rule, and then working on the smaller goalterms of the successor proofnodes independently. However, a “good” caseterm must be used, i.e. a caseterm which yields smaller goalterms in fact. Finding a “good” caseterm is sometimes a matter of trial-and-error, in particular if the goalterm under consideration is too complex. If the used caseterm yields not a good result, we simply prune the prooftree and try another one.

For instance, one may use Case Analysis with the caseterm A to decompose the goalterm

\[
\text{if}(A, \\
\quad \text{if}(B, \text{true}, \text{if}(C, \text{D}, \text{false})), \\
\quad \text{if}(D, \text{if}(C, \text{if}(B, \text{false}, \text{true}), \text{false}), \text{false})).
\]

This yields (after symbolic evaluation) two new goalterms, viz.

\[
\text{if}(A, \text{if}(B, \text{true}, \text{if}(C, \text{D}, \text{false})), \text{true})
\]

and

\[
\text{if}(A, \text{true}, \text{if}(D, \text{if}(C, \text{if}(B, \text{false}, \text{true}), \text{false}), \text{false}))
\]

which do not decompose the original goalterm significantly. We withdraw the use of Case Analysis by pruning the prooftree and then try Case Analysis again, but this time using the caseterm B. This yields (after symbolic evaluation) the pair of new goalterms

\[
\text{if}(B, \text{A}, \text{true}) \quad \text{and} \quad \text{if}(B, \text{true}, \text{if}(C, \text{D}, \text{false}))
\]

which represents a good decomposition to continue with the proof.
Creating a Lemma

We have already seen two alternatives for verifying the commutativity of \texttt{plus}: One verification used the auxiliary lemmas \texttt{plus_right_succ} and \texttt{plus_right_zero} while the other attempt did not require auxiliary lemmas, but needs two nested inductions, one for proving the base case and the other for proving the step case of the “outer” induction.

So one may ask which approach is the better one. The answer depends on the usefulness of an auxiliary lemma for subsequent verifications. If such a lemma can be used subsequently, a nested induction would “hide” the lemma thus necessitating additional nested inductions in cases where the auxiliary lemma could have been applied. On the other hand, if the auxiliary lemma is not useful in other places, a nested induction would be the better choice because system performance may decrease with an increase of verified lemmas, and a huge amount of verified lemmas may also irritate the user in guiding the system to a proof.

Since it may happen that a user decided to use a nested induction but later on discovers that an auxiliary lemma would be the better choice, \texttt{veriFun} supports the creation of a lemma and its prooftree from the prooftree of another lemma.

Consider, for instance, the proof of \texttt{plus_commutative} by nested inductions and let us create the auxiliary lemmas from the prooftree of \texttt{plus_commutative}:

\begin{itemize}
  \item select \texttt{plus_commutative} in the \texttt{Program Window},
  \item choose \texttt{Proof} from the \texttt{Program Menu} or from the context menu or alternatively double-click \texttt{plus_commutative} in the \texttt{Program Window},
  \item in the \texttt{Proof Window} select the proofnode labeled with “\textit{Induction*: plus}” in the prooftree’s subtree belonging to the base case,
  \item choose \texttt{Create Lemma} from the \texttt{Proof Menu} or from the context menu,
\end{itemize}
⇒ in the input dialog opened by **Create Lemma** specify the name of the lemma to be created, say `plus_right_zero#2`, and
⇒ push **OK** to create the new lemma `plus_right_zero#2` with a prooftree copied from the sub-prooftree of `plus_commutative`.

In the same way the other auxiliary lemma is created, if **Create Lemma** is applied to the proofnode labeled with "*Induction* : plus" in the prooftree’s subtree belonging to the step case.

**Exploring Alternatives**

The **Create Lemma** command is also useful to duplicate a lemma together with its prooftree. While working on a prooftree, it may happen that a user becomes skeptical about the decisions (in form of proof rule applications) already made, and he or she may wonder whether it would be better to follow another proof idea instead.
In this situation, **Create Lemma** applied to the root node of a prooftree can be used to obtain a copy of the prooftree. Then the copied prooftree is opened in the *Proof Window*, the proofnode to which an alternative proof rule shall be applied is pruned, and work is resumed by applying the alternative proof rule to the leaf obtained by the recent **Prune** command.

**Using Proof Libraries**

When working on larger case studies, some formal groundwork usually is required. E.g., the verification of a cryptographic protocol needs many lemmas from number theory, but it is a waste of time to start always over from scratch by proving properties of addition, multiplication, divisors, prime numbers, etc. already proved elsewhere. The situation is same when doing a pencil-and-paper proof, where one consults a textbook or a collection of formulas for well known and proven facts, instead of proving them again thus reinventing the wheel.

**VeriFun** supports this kind of work with the **Import** command applied to so-called *proof-libraries*. A proof-library needs no special format but is set up like any other *vf*-file. Although any *vf*-file can be used, it is good practice to collect only those definitions and lemmas in a proof-library which are related to a specific domain, e.g. *Arithmetic*, *Linear Lists*, *Binary Trees* etc., and which are of general interest within this domain.

Assume for instance, all definitions, lemmas and proofs of this tutorial have been stored in the *vf*-file *Tutorial.vf*. Then we may start in setting up a proof-library for *Arithmetic* in the following way:
choose **New** from the *File Menu* to reset the system to initial state,
choose **Import** from the *File Menu* to open the *Import Window,*
in the *Import Window* push **Open,** select Tutorial.vf in the input
dialog opened by **Open,** and push **Open** in this dialog to load Tutorial.vf as the export-program,
in the *Import Program* part of the *Import Window* select some program
element, say the root folder *Program,* to determine the place where
the program elements to be imported shall be located in the actual
program, also called the import-program,
in the *Export Program* part of the *Import Window* select the program
elements, say plus, plus_associative, plus_commutative, half and half_plus, which shall be imported into the import-program,
choose **Import** from the menu bar of the *Import Window* or from the
context menu in order to import the program elements selected in the
export-program into the import-program, and
choose **Save as** from the *File Menu* to save the actual program created
by the imported program elements to some file, say Arithmetic.vf.

In this way we have set up a (simple) proof-library for Arithmetic,
which we may extend by definitions and lemmas from other *vf-*
files whenever we like to do so. For instance, if we want to extend our
proof-library by some lemmas about multiplication already stored
in another *vf*-file, say Exmpl3.vf, we open Arithmetic.vf as the
import-program, open Exmpl3.vf as the export-program and then
import the definitions and lemmas we are concerned with into our proof-library for Arithmetic.
In the same way proof-libraries are used when working on some verification problem. The proof-library is opened as the export-program to import all, what seems helpful to solve the problem, from the proof-library into the actual program.

See File\Import in the $\text{VeriFun User Guide}$ for further details about the Import command.

**Tips and Tricks**

(1) The system reacts with an *out-of-memory* exception when writing a *vf*-file, if not enough memory is assigned on system installation. In such a case, *do not repeat the Save command* as this would corrupt the backup file which holds all data of the most recent *successful* Save operation. See http://www.informatik.tu-darmstadt.de/pm/verifun/ for details of system installation.

(2) In an induction proof, being it a user guided or a system suggested induction, at least some of the induction hypotheses should have been used by the Symbolic Evaluator in the subsequent evaluation step. Failing to apply induction hypotheses often indicates either the use of “wrong” induction variables or the use of a “wrong” relation description, i.e. a “wrong” induction axiom. Analyze the prooftree computed by the system, and, if required, use the Induction proof rule with the “right” settings. Otherwise, consider Use Lemma or Apply Equation to bring a useful instance of an induction hypothesis into play.

(3) If an instance of an induction hypothesis or a lemma is applied with the Use Lemma or the Apply Equation proof rule, it may happen that the Symbolic Evaluator uses the induction hypothesis or the lemma in the subsequent proof step to eliminate the applied instance from the goal-term. This usually means that the instance does not contribute to a proof. E.g., a conditional equation has been applied in a position of the goal-term, which does not satisfy the conditions of the equation. Analyze the
goalterm to check out which of the available lemmas or induction hypotheses could contribute to a proof.

(4) If the Symbolic Evaluator fails to use an induction hypothesis, the system may guess an instance of the induction hypothesis to apply it to the goalterm with the Use Lemma or the Apply Equation proof rule. If the proof gets stuck, remember that the system’s heuristics are not foolproof. E.g., a wrong instance may have been guessed or an equation has been applied in the wrong direction. Analyze the goalterm and withdraw the system’s decision if required.

(5) Do not linearize terms in order to enable a certain induction. E.g., a term like \( \ldots \text{plus}(\text{half}(n), m) \ldots \) “blocks” a plus–induction because the place of the recursion variable is occupied by a non-variable term, viz. \( \text{half}(n) \). Of course, a linearization \( \text{if}(x=\text{half}(n), \ldots \text{plus}(x, m) \ldots , \text{true}) \) represents an equivalent statement and also enables a plus–induction, but this usually does not help because unusable induction hypotheses would result.

(6) It may happen (in very rare cases) that neither of the available relation descriptions represents the induction axiom to be used. In such a situation, write a procedure with recursive calls similar to the induction hypotheses of the desired induction axiom. Then prove the procedure’s termination (if the system fails to do so) in order to create the relation description representing the desired induction.

(7) Only use the predefined procedure \( > \) to reason about inequalities of natural numbers. E.g., use \( y>x \) to express \( x<y, \text{if}(y>x, \text{false}, \text{true}) \) instead of \( \text{ge}(x,y) \) (for a user defined procedure \( \text{ge} \) computing \( x \geq y \)), etc. Using only \( > \) pays, because \( \text{VeriFun} \) provides several predefined lemmas about \( > \), e.g. transitivity, irreflexivity, etc., which must be reformulated and proved if user defined procedures (like \( \text{ge} \)) are used.

(8) In the Insert dialog, the comment field may be used as a clipboard.
(9) Use **User Settings** from the *File Menu* to disable the tests for the applicability of the computed proof rules in order to speed up navigation in proof trees of *large* programs.

(10) Most dialogs allow the usual keys `ctrl-A`, `ctrl-C`, `ctrl-V` and `ctrl-X` for *select all*, *copy*, *paste* and *cut*.

(11) Parsing errors may result from special characters which do not appear in print. In such a case, copy the input and use an ASCII-Editor, which makes such characters visible, for deletion.

(12) Create an *fp*-file and open this file in some ASCII-Editor to look at the actual program as a whole. Note that the order of program elements in an *fp*-file depends on the order of insertion of the program elements into the actual program.

(13) Use the **Evaluation** command from the *Proof Menu* to explore a symbolic evaluation in order to check out why the application of a certain proof rule does not yield the expected result.