Bachelor Thesis

Developing a Web-Based Hoare Logic Proof Assistant

Flavio Goldener

Supervisors:

Malte Schwerhoff
Prof. Dr. Peter Müller

ETH Zürich, Switzerland

23.03.2016

Contents

1 Motivation

Hoare logic [1] (also known as axiomatic semantics) is a family of formal systems for reasoning about properties of programs, for example, correctness and termination.

Hoare logic introduces the concept of Hoare triples, which use assertions (boolean expressions) to describe, how the execution of statements affects the program. A Hoare triple is typically of the form

$$\{\mathbf{P}\}\ s\ \{\mathbf{Q}\}$$

where **P** and **Q** are assertions, **P** is called the precondition of a triple, **Q** is called the postcondition of a triple and s is a statement. Informally, the meaning of the triple is, that if **P** holds before the statement s, then **Q** will hold after the statement s is executed.

When formalizing Hoare logics, derivation systems are used to describe valid Hoare triples and therefore proofs in Hoare logics are usually done by building derivation trees. When it comes to proving properties manually using pen and paper, proofs are rather done by building proof outlines than by building derivation trees. The reason why proof outlines are chosen is the fact that they are more convenient to work with, because derivation trees tend to get large and a lot of assertions and statements have to be written multiple times while proof outlines reduce the writing amount that has to be done when proving properties on paper (please refer to appendix A for a comparison between a proof outline and a derivation tree).

Although Hoare logic is relatively easy to apply, it is not intuitive right from the start, especially for people which have never heard of Hoare logics before. Building correct proof outlines can be challenging, especially during the learning phase, when the concept of Hoare logic is not fully understood yet. One potential problem that many beginners face is that they are not sure which rule should be applied in which step. If the task were to build a proof outline for

$$\{true\}\ \texttt{skip; x:=0}\ \{x = 0\},$$

many beginners would be tempted to apply the skip rule from Figure 1 as a first step, because the first part of the statement is a skip statement. But this would not be the correct rule to apply, despite the fact that the first statement is a skip statement. Neither can the assignment rule be used, the right rule to apply is the rule for sequential composition. Only after applying the rule for sequential composition the other rules can be applied.

$$\frac{}{\{\underline{P}\}\ \texttt{skip}\ \{\underline{P}\}}\ (\text{Skip})$$

$$\frac{}{\{\underline{P}[\underline{x} \mapsto \underline{e}]\}\ \underline{x} := \underline{e}\ \{\underline{P}\}}\ (\text{Ass})$$

$$\frac{\{\underline{P}\}\ \underline{s}\ \{\underline{Q}\} \qquad \{\underline{Q}\}\ \underline{s}'\ \{\underline{R}\}}{\{\underline{P}\}\ \underline{s};\underline{s}'\ \{\underline{R}\}}\ (\text{Seq})$$

Figure 1: Derivation rules, a complete list can be found in chapter 3.

As one can see, already recognizing which rule to apply can cause some troubles. But even if the correct rule were chosen, there would still be a chance that an inexperienced person wouldn't apply the rule correctly. If such a mistake is made, it is possible that such an error propagates in the succeeding steps done by the person building the proof outline without being detected. If the error is then spotted later, all the work done in-between has to be undone, which is quite cumbersome, especially when the proof is done on paper.

At ETH, Hoare logics is taught in the lecture "Formal Methods and Functional Programming", which is a course that is mandatory for all undergraduate computer science students. Experience has shown that many students face exactly the problems described above. One of the reasons why the learning phase can be quite challenging is the one, that immediate feedback indicating whether or not a rule has been applied correctly is often not available and therefore there is the danger that the insights gained by building proof outlines as an exercise are not optimal.

This thesis tackles this problem by providing the students with a Hoare logic proof assistant, which supports them during their learning phase by supporting them when they build proof outlines.

2 Results

The result of this bachelor's thesis is a web-based Hoare logic proof assistant for teaching purposes, which supports students in becoming familiar with Hoare logic proof outlines. The reason, why the proof assistant is web-based is the fact, that it is easily accessible, easy to use and no installation is required which leads to a higher chance that the assistant is used by future students.

The focus of the development was on usability and feedback, which helps the students to get familiar with Hoare logics. The results achieved by the thesis are the following ones:

- Development of a web application which is intuitively usable and makes sure that the proofs are correct by construction, unless the user introduces entailments which don't hold. The application also provides immediate feedback, such as errors if the user tried to apply the wrong rule.
- Development of the necessary functionality for parsing the proof outline, storing it as an abstract syntax tree and pretty-printing it in the browser.
- Implementation of a suitable data structure for representing partial proof outlines in a way that supports both forward (typically used when if statements are involved) and backward reasoning (typically used in assignments).
- Support of partial and total correctness, which includes providing the functionality to reason about termination of loops.
- Development of the functionality for downloading the proof outline as a PDF, such that the proof can be handed in to an assistant to get additional feedback.
- Development of a history view in which the user can review previous steps of the proof outline which helps keeping track of what has been done so far. The whole history can be saved and loaded to be reviewed at a later time.
- Providing the functionality for undoing and redoing previous steps.
- Providing the functionality for saving and loading (unfinished) proofs.
- Record usage statistics, e.g. the number of users, usage time, error messages and jumps back resulting from undoing steps which helps the assistants to recognize where students faced difficulties.

3 Prerequisites

3.1 IMP

In this thesis, we use a programming language called IMP (inspired by While [2]), which is also used in the course "Formal Methods and Functional Programming" at ETH. IMP has just the basic features, in concrete:

- Boolean and arithmetic expressions with no side effects and no division
- Variables which are initialized and range over integers
- Assignments, If-then/else statements, loops

In the lecture "Formal Methods and Functional Programming", there were also extensions of the IMP language proposed. But as this thesis focused on the essential basics, adding these extensions to the assistant is left for future work (please refer to section 9).

The concrete syntax of IMP is the following:

```
Letter   = | 'A' | ... | 'Z' | 'a' | ... | 'z' |
Digit    = | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7 ' | '8' | '9'

Ident    = Letter{Letter | Digit}*
Numeral  = Digit| Numeral Digit
Var      = Ident

Aexp     = '(' Aexp OP Aexp ')'
         | Var
         | Numeral

Op       = '+' | '−' | '∗'

Bexp     = '(' Bexp 'or' Bexp ')'
         | '(' Bexp 'and' Bexp ')'
         | 'not' Bexp
         | Aexp Rop Aexp

Rop      = '=' | '#' | '<' | '<='| '>'| '>='

Stm      = 'skip'
         | Var := Aexp
         | '(' Stm ';' Stm ')'
         | 'if' Bexp 'then' Stm 'else' Stm 'end'
         | 'while' Bexp 'do' Stm 'end'
```

3.2 IMP syntax supported by the assistant

When using the proof assistant, the supported syntax was a little bit relaxed, for example, brackets and semicolons are not mandatory and some syntactic sugar elements were added to make sure that users, which are more familiar with languages like C++ and Java, face no difficulties entering a valid IMP program. The syntactic abbreviations are the following ones:

'true'	$\widehat{=}$	1=1
'false'	$\widehat{=}$	0=1
'||'	$\widehat{=}$	'or'
'&&'	$\widehat{=}$	'and'
'!'	$\widehat{=}$	'not'
'!='	$\widehat{=}$	'#'
if b then s end	$\widehat{=}$	if b then s else skip end

Additionally, negative numbers are allowed as well.

3.3 Hoare logic for IMP

The axiomatic semantics of IMP is formalized by describing the valid Hoare triples, which can be done by a derivation system. The rules that are used in this thesis are the following ones:

$$\frac{}{\{\underline{P}\}\ \texttt{skip}\ \{\underline{P}\}}\ \text{(Skip)}$$

$$\frac{}{\{\underline{P}[\underline{x} \mapsto \underline{e}]\}\ \underline{x} := \underline{e}\ \{\underline{P}\}}\ \text{(Ass)}$$

$$\frac{\{\underline{P}\}\ \underline{s}\ \{\underline{Q}\} \qquad \{\underline{Q}\}\ \underline{s}'\ \{\underline{R}\}}{\{\underline{P}\}\ \underline{s};\ \underline{s}'\ \{\underline{R}\}}\ \text{(Seq)}$$

$$\frac{\{\underline{b} \wedge \underline{P}\}\ \underline{s}\ \{\underline{Q}\} \qquad \{\neg\underline{b} \wedge \underline{P}\}\ \underline{s}'\ \{\underline{Q}\}}{\{\underline{P}\}\ \texttt{if}\ \underline{b}\ \texttt{then}\ \underline{s}\ \texttt{else}\ \underline{s}'\ \texttt{end}\ \{\underline{Q}\}}\ \text{(If)}$$

$$\frac{\{\underline{b} \wedge \underline{P}\}\ \underline{s}\ \{\underline{P}\}}{\{\underline{P}\}\ \texttt{while}\ \underline{b}\ \texttt{do}\ \underline{s}\ \texttt{end}\ \{\neg\underline{b} \wedge \underline{P}\}}\ \text{(Wh)}$$

$$\frac{\{\underline{b} \wedge \underline{P} \wedge \underline{e} = \underline{Z}\}\ \underline{s}\ \{\Downarrow \underline{P} \wedge \underline{e} < \underline{Z}\}}{\{\underline{P}\}\ \texttt{while}\ \underline{b}\ \texttt{do}\ \underline{s}\ \texttt{end}\ \{\Downarrow \neg\underline{b} \wedge \underline{P}\}}\ \text{(WhTot) if } \underline{b} \wedge \underline{P} \vDash 0 \leq \underline{e}$$

$$\frac{\{\underline{P}'\}\ \underline{s}\ \{\underline{Q}'\}}{\{\underline{P}\}\ \underline{s}\ \{\underline{Q}\}}\ \text{(Cons) if } \underline{P} \vDash \underline{P}' \text{ and } \underline{Q}' \vDash \underline{Q}$$

An important remark that has to be made about the rules above is the one that they are only rule schemes and not concrete instances of the rules. Therefore they use meta-variables over assertions (printed in capital, underlined letters) and over expressions and statements (printed in lowercase, underlined letter).

3.4 Proof outlines

As already mentioned in the motivation, derivation trees are inconvenient to work with, and proof outlines are therefore usually used to prove properties about a program on paper. The idea of proof outlines is to group the assertions around the program text. Assertions are written before and after each statement to indicate which properties hold in the states before and after the execution of this statement.

The rules in proof outline notation are the following ones:

Skip:

$\{\underline{P}\}$
`skip`
$\{\underline{P}\}$

Ass:

$\{\underline{P}[\underline{x} \mapsto \underline{e}]\}$
$\underline{x} := \underline{e}$
$\{\underline{P}\}$

Seq:

$\{\underline{P}\}$
$\underline{s}$
$\{\underline{Q}\}$
$\underline{s}'$
$\{\underline{R}\}$

If:

$\{\underline{P}\}$
`if` $\underline{b}$ `then`
$\{\underline{b} \wedge \underline{P}\}$
$\underline{s}$
$\{\underline{Q}\}$
`else`
$\{\neg\underline{b} \wedge \underline{P}\}$
$\underline{s}'$
$\{\underline{Q}\}$
`end`
$\{\underline{Q}\}$

While:

$\{\underline{P}\}$
`while` $\underline{b}$ `do`
$\{\underline{b} \wedge \underline{P}\}$
$\underline{s}$
$\{\underline{P}\}$
`end`
$\{\neg\underline{b} \wedge \underline{P}\}$

WhileTot:

$\{\underline{P}\}$
`while` $\underline{b}$ `do`
$\{\underline{b} \wedge \underline{P} \wedge \underline{e} = \underline{Z}\}$
$\underline{s}$
$\{\Downarrow \underline{P} \wedge \underline{e} < \underline{Z}\}$
`end`
$\{\Downarrow \neg\underline{b} \wedge \underline{P}\}$

Cons:

$\{\underline{P}\}$
$\models$
$\{\underline{P}'\}$
$\underline{s}$
$\{\underline{Q}'\}$
$\models$
$\{\underline{Q}\}$

As a convention, if the same assertions appear next to each other without having program text in between, we only write one of them in the proof outline. This allows us to save space, and reduce the writing amount which is illustrated by Figure 2.

$$\dfrac{\dfrac{\dfrac{}{\{0=0\}\ \texttt{skip}\ \{0=0\}}\ (\text{Skip})}{\{true\}\ \texttt{skip}\ \{0=0\}}\ (\text{Cons}) \qquad \dfrac{}{\{0=0\}\ \texttt{skip}\ \{0=0\}}\ (\text{Skip})}{\{true\}\ \texttt{skip; skip}\ \{0=0\}}\ (\text{Seq})$$

$\{true\}$
$\models$
$\{0=0\}$
`skip;`
$\{0=0\}$
`skip;`
$\{0=0\}$

Figure 2: Derivation tree vs proof outline

3.5 Supported syntax for Assertions

The supported syntax for assertions by the proof assistant includes the most common boolean and arithmetic operators as well as sums, products, and functions like floor and ceil. For the complete syntax please refer to the appendix B.

4 Building proof outlines on paper vs building proof outlines using the assistant

Although building proof outlines works very similar when done on paper and when using the assistant, there are some differences, which are described in this chapter. Firstly, we want to give an intuition of how proof outlines are done on paper, what the common mistakes are that can be made if the outline is done on paper and how the assistant prevents these mistakes.

4.1 How to build proof outlines on paper

When building proof outlines on paper, the two techniques that can be used are working forwards and working backwards. To illustrate how these techniques can be used, we build a proof outline for the triple

$$\{true\}$$
```
skip;
x:=0;
```
$$\{x = 0\}$$

To make it clear how the proof outline would be built on paper, its construction is discussed step-by-step and also illustrated by several figures.

Figure 3 shows how the proof outline looks in the very first step, no rules have been applied yet.

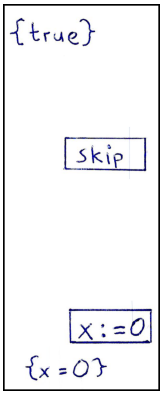


Figure 3: Proof outline written on paper

4.1.1 Working backwards

When working backwards, the postcondition of a statement and the statement itself is used to derive the precondition.

In our example, we use the postcondition $\{x = 0\}$ and the statement `x:=0` and by applying the assignment rule, which is done by replacing all instances of x with 0, we get that the precondition is $\{0 = 0\}$.

An illustration how the proof outline looks written down on paper after this first step can be seen in Figure 4.

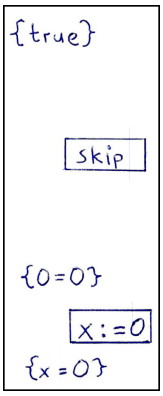


Figure 4: Outline after the first step

4.1.2 Working forwards

When working forwards, the precondition of a statement and the statement itself are used to derive the postcondition.

In our example, we use the assertion $\{true\}$ and the statement skip and by applying the skip rule, we get that the postcondition is also $\{true\}$. After this rule application, our proof outline is the one illustrated in Figure 5.

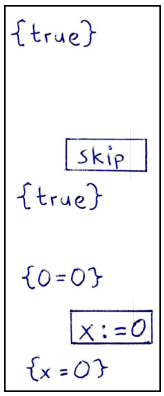


Figure 5: Outline after the second step

4.1.3 Using both techniques

In practice, a proof outline is built up by working both forwards and backwards, depending on the context. As there is no clear answer, which technique should be applied when, and because the rule of consequence can be introduced at any place, proof outlines can be constructed with a high degree of freedom.

As an illustration, consider the proof outline done so far. After working back- and then forwards, which led to Figure 5, we can introduce the rule of consequence, shown in Figure 6, which concludes the proof.

Alternatively, after the first step (Figure 4), we could have worked backwards once again and then introduced the rule of consequence, which would have led to the proof outline illustrated in Figure 7. Both outlines are correct proofs.

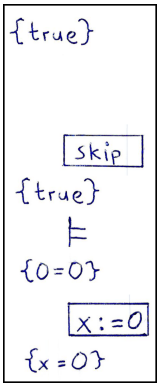


Figure 6: Outline resulting from working backwards and forwards

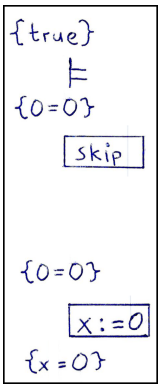


Figure 7: Outline resulting from working backwards twice

4.2 Common mistakes made when doing proofs on paper

As we have seen, building proof outlines on paper comes with a high degree of freedom. But the downside is, that this degree of freedom makes it more likely that beginners make mistakes when building proof outlines. Two mistakes that are made quite often when building proof outlines for the first time are the following ones:

Wrongly applying rules

When the rules aren't fully understood yet, there is a danger that users apply them wrongly. Especially applying the assignment rule seems not straightforward right from the start. Consider Figure 8 to see two typical cases, where the assignment rule is wrongly applied.

Wrong application:

$\{x = 0\}$ `x:=0;` $\{0 = 0\}$

$\{true\}$ `x:=0;` $\{x = 0\}$

Rule:

$\{\underline{P}[\underline{x} \mapsto \underline{e}]\}$
$\underline{x}{:=}\underline{e}$
$\{\underline{P}\}$

Figure 8: Wrong application of the assignment rule

Thinking that the proof is finished although it isn't

If the proof were done by working backwards twice, which means that we applied the assignment rule and then the skip rule, we would get the proof outline which can be seen in Figure 9.

An inexperienced user might think that we are done now, as $\{true\}$ and $\{0 = 0\}$ are semantically the same statements. But the proof outline is not really finished, the rule of consequence needs to be introduced and if this were forgotten the proof outline would not be valid.

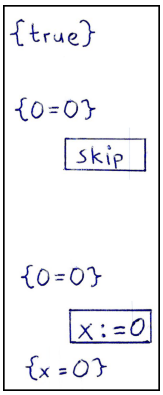


Figure 9: Incomplete proof outline

4.3 How the assistant prevents mistakes

As seen in the previous subsection, a proof outline can be built in different ways. Naturally, the user should have the same possibilities when using the assistant as if they did the proof on paper. But on the other hand, we want to prevent the user from making mistakes as the ones discussed before.

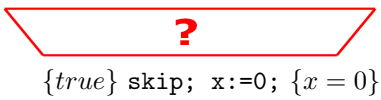
4.3.1 Meta-variables and holes

To achieve this goal, the assistant makes use of meta-variables and *holes*. Meta-variables are used to give the user the flexibility to apply rules without at the same time needing to specifying the assertions. holes on the other hand are indicators which show users where there is still more work to do, e.g. where the proof outline is incomplete. How the assistant uses meta-variables and holes is illustrated in the following subsection.

4.3.1.1 How holes and meta-variables are used - an illustration

To illustrate how meta-variables and holes are used, we show how the proof previously done on paper would have looked like if it were done using the assistant.

At the beginning



$\{true\}$ `skip; x:=0;` $\{x = 0\}$

Figure 10: Representation as a derivation tree

$\{true\}$
?
`skip;`
`x:=0;`
?
$\{x = 0\}$

Figure 11: Representation as a proof outline

If the assistant were used to do the proof, the outline at the beginning would look like Figure 11. The red question marks are the indicators that there is a hole. The reason why there is a hole can be seen when looking at the corresponding derivation tree in Figure 10, which at this stage consists of only one triple. This triple is not valid yet, there is more work to do which is indicated by the hole.

After the application of the rule for sequential composition

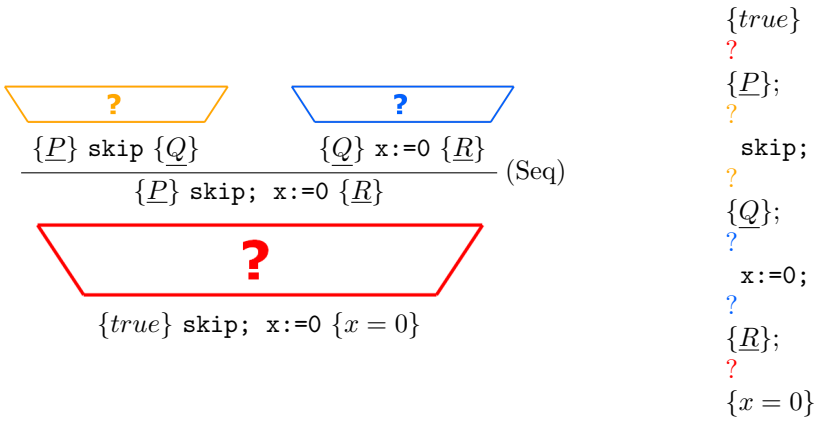


Figure 12: The proof after applying the rule for sequential composition

After we applied the rule for sequential composition, Figure 12 illustrates how our proof outline and our derivation tree would look like. When applying a rule, the assistant introduces meta-variables for all newly created assertions. As users can decide when to instantiate a meta-variable, and they can remove each hole by introducing the rule of consequence, they still have the same degree of freedom as if they did the proof on paper.

One important remark is the one, that the assistant allows the user to use $\{true\}$ instead of $\{\underline{P}\}$ and $\{x := 0\}$ instead of $\{\underline{R}\}$ in the newly introduced rule, and if the user actually wanted to do that, introducing the meta-variables $\underline{P}$ and $\underline{Q}$ would not be necessary and the red hole would not exist. How this is actually done is described in chapter 5 step 5. But let us for the sake of explanation assume that the user wanted the assistant to introduce the meta-variables $\underline{P}$ and $\underline{Q}$.

Therefore, as the assistant introduces new meta-variables for all assertions, there is a hole in the derivation tree between the triple we applied the rule to and the rule that was introduced. To see why there is a hole, let us have a look at the rule for sequential composition rule that was introduced. The sequential rule, as it was introduced by the assistant is the following one

$$\frac{\{\underline{P}\}\ \texttt{skip}\ \{\underline{Q}\} \qquad \{\underline{Q}\}\ \texttt{x:=0}\ \{\underline{R}\}}{\{\underline{P}\}\ \texttt{skip;}\ \texttt{x:=0}\ \{\underline{R}\}}\ (\text{Seq})$$

Let us have a closer look at the triple on the bottom of the rule, which is

$$\{\underline{P}\}\ \texttt{skip;}\ \texttt{x:=0}\ \{\underline{R}\}$$

For the sake of simplicity, let us refer to the precondition of such a triple on the bottom of a rule as the precondition of the rule and to the postcondition of such a the triple on the bottom of a rule as the postcondition of the rule.

One thing that we can observe when looking at Figure 12 is that the triple on the bottom of the rule

$$\{\underline{P}\}\ \texttt{skip; x:=0}\ \{\underline{R}\}$$

doesn't match the triple which we applied the rule to

$$\{true\}\ \texttt{skip; x:=0}\ \{x = 0\}$$

Therefore, as the precondition of the triple we applied the rule to ($\{true\}$) doesn't match the precondition of the rule for sequential composition ($\{\underline{P}\}$) and the postcondition of the triple ($\{x = 0\}$) doesn't match the postcondition of the rule ($\{\underline{R}\}$), there is the red hole in between, which can be seen in Figure 12.

What we observe as well is that in order to make the proof outline more readable, $\{\underline{P}\}$ and $\{\underline{R}\}$ are only printed once in the proof outline although they appear multiple times in the corresponding derivation tree.

The orange and the blue holes are again there because the triples that they are above in the derivation tree are not valid yet and there is more work to be done.

After unification

If we want to remove the red hole, we need to unify the precondition of the triple with the precondition of the rule and the postcondition of the triple with the postcondition of the rule.

To unify two assertions means that we instantiate the meta-variables that are used in the given assertions in such a way that these two assertions become syntactically equal. For more information please refer to subsection 6.2.

So if we want to unify $\{true\}$ and $\{\underline{P}\}$, we can instantiate $\underline{P}$ by $true$ and unifying $\{x = 0\}$ and $\{\underline{R}\}$ can be done by instantiating $\underline{R}$ by $x = 0$.

After we did unify those assertion, the proof assistant automatically removes the hole between the triple and the rule, which leads us to the representation of the derivation tree and the proof outline which can be seen in Figure 13. For a longer explanation, how the assistant removes the hole exactly, please refer to chapter **??** step 4.

$$\dfrac{\overset{?}{\{\mathit{true}\}\ \texttt{skip}\ \{\underline{Q}\}} \qquad \overset{?}{\{\underline{Q}\}\ \texttt{x:=0}\ \{x = 0\}}}{\{\mathit{true}\}\ \texttt{skip;}\ \texttt{x:=0}\ \{x = 0\}}\ (\text{Seq})$$

$\{\mathit{true}\}$
?
`skip;`
?
$\{\underline{Q}\}$;
?
`x:=0;`
?
$\{x = 0\}$

Figure 13: The proof after unification

After applying the assignment rule

If we applied the assignment rule as a next step, the proof assistant would again introduce meta-variables and holes, which can be seen in Figure 14.

$$\dfrac{\overset{?}{\{\mathit{true}\}\ \texttt{skip}\ \{\underline{Q}\}} \qquad \dfrac{}{\{\underline{S}[x \mapsto 0]\}\ \texttt{x:=0}\ \{\underline{S}\}}\ (\text{Ass})}{\{\mathit{true}\}\ \texttt{skip;}\ \texttt{x:=0}\ \{x = 0\}}\ (\text{Seq})$$

$\{\mathit{true}\}$
?
`skip;`
?
$\{\underline{Q}\}$;
?
$\{\underline{S}[x \mapsto 0]\}$
`x:=0;`
$\{\underline{S}\}$
?
$\{x = 0\}$

Figure 14: Representation after applying the assignment rule

4.3.1.2 What is achieved by using meta-variable and holes

With the help of meta-variables and holes, the assistant is able to provide the user with a high degree of freedom but also has the possibility of indicating where more work needs to be done.

As the user now sees the holes in the proof outline and as the rules are always correctly applied by the assistant, the mistakes mentioned before can't be made anymore. In fact, unless the user introduces entailments which don't hold, the proof outline is always going to be correct.

5 A tour through the assistant

Showing the features that the proof assistant provides is done by illustrating how an example proof outline could be built up using the assistant. Concretely, we want to prove that

$$\{x = N \wedge x > 0\}$$
```
y:=1;
while not x=1 do
   y:=y·x;
   x:=x-1;
end
```
$$\{y = N! \wedge N > 0\}$$

which is nothing other than a proof that the program actually implements the factorial function. Note that N is a logical variable (i.e. a constant), which means that N is only used in assertions and never in program statements. Here, N denotes the initial value of the program variable x.

Step 1: Entering the program

Before the user can actually start building a proof outline, they have to enter the pre- and postcondition and the IMP program which they want to work with, which is done using the start page (Figure 15). When the user enters an assertion, the proof assistant automatically tries to parse this assertion and if that succeeds the entered assertion gets pretty-printed right below the input box where it was entered.

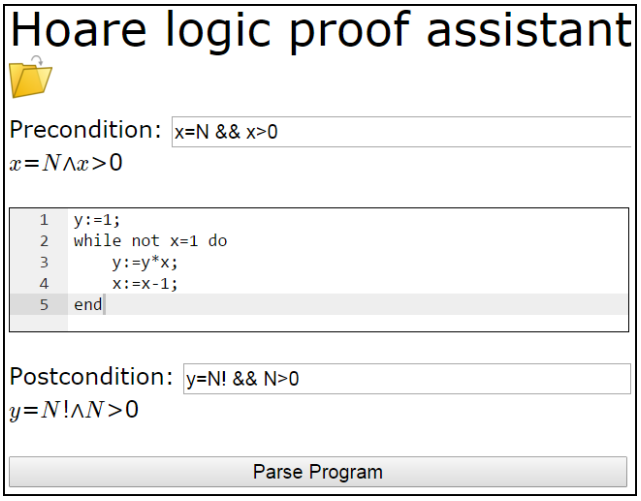


Figure 15: Screenshot of the start page

After the entered precondition, postcondition and IMP program were successfully parsed by the assistant, the user can proceed by clicking on the "Parse Program" button.

Step 2: Parsing

After the user has entered a new program, the assistant creates the proof outline and switches to the main page (Figure 16), where the proof outline can be edited.

In order to build a proof outline, the user can use the various elements provided by the GUI, which are briefly described below.

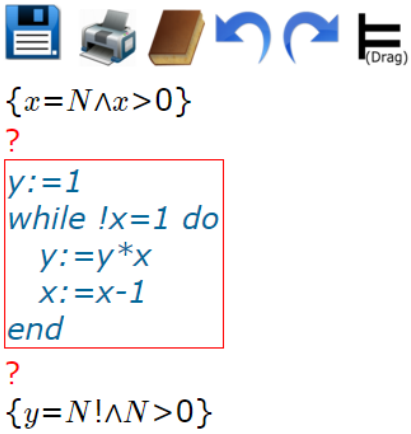


Figure 16: Screenshot of the main page

A proof outline can be downloaded and stored locally using the save button (see step 13).

A proof outline can be printed using the print button (see step 13).

The history view button enables us to review previous steps (see step 9).

The undo button enables the user to undo the last step.

The redo button enables the user to redo the step undone before.

The rule of consequence can be introduced by dragging the entailment symbol (see step 10).

The question mark is the indicator for a hole.

If there is a triple which has not yet been proven, its statement is printed in a blue font and inside a red frame. By clicking on such a statement, we open a dialog in which we can specify which rule we want to apply, (i.e. which rule we want to have above this triple). More information can be found in step 3.

Step 3: Applying the rule for sequential composition

After clicking on the sequential composition statement which is printed in a blue font and has a red border (see Figure 16), the dialog shown in Figure 18 is opened. Using this dialog, the user can specify which rule to apply by clicking on the corresponding button. As one can see, the dialog also comes with two text and two check boxes, but we don't worry about them now, they are explained in step 5 and 6.

If the user tried to apply a rule which doesn't match the statement and therefore would be wrong to apply, they would get an error message.

After the user clicked on the "Apply Sequential Rule" button, the assistant applies the rule for sequential composition. The resulting proof outline can be seen in Figure 17.

$\{x{=}N \wedge x{>}0\}$
?
$\{\underline{P}\}$
?
```
y:=1;
```
?
$\{\underline{Q}\}$
?
```
while !(x=1) do
   y:=y*x;
   x:=x-1;
end
```
?
$\{\underline{R}\}$
?
$\{y{=}N! \wedge N{>}0\}$

Figure 17: Result of the sequential composition rule application

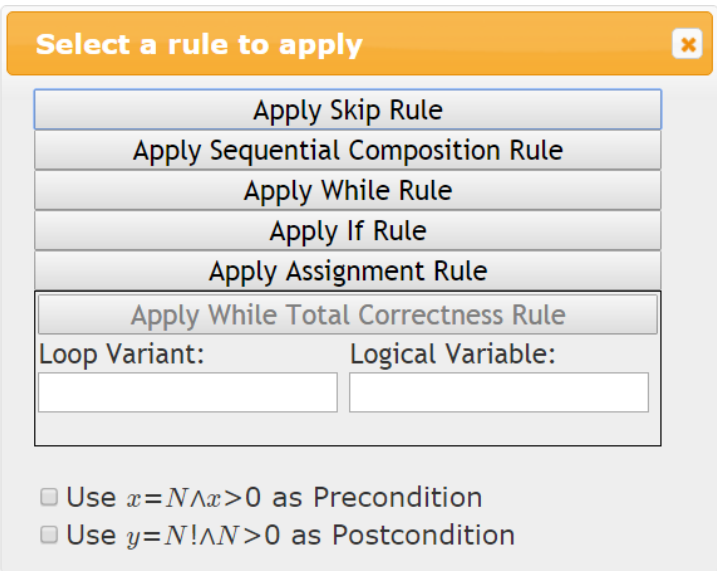


Figure 18: Screenshot of the rule application dialog

Step 4: Removing a hole

After the rule for sequential composition was applied, our proof outline now is the one displayed in Figure 17. When we have a look at how the derivation tree looks at this stage (Figure 19), we can see that we have a hole between the root triple and the sequential rule.

As we have seen earlier, such a rule can be removed by unifying the precondition of the triple with the precondition of the rule and the postcondition of the triple with the postcondition of the rule.

$$\dfrac{\overset{?}{\{\underline{P}\}\ \texttt{y:=1}\ \{\underline{Q}\}} \qquad \overset{?}{\{\underline{Q}\}\ \texttt{while !(x=1) do s end}\ \{\underline{R}\}}}{\{\underline{P}\}\ \texttt{skip; x:=0}\ \{\underline{R}\}}\ (\text{Seq})$$

$$\overset{?}{\{x = N \wedge x > 0\}\ \texttt{skip; x:=0}\ \{y = N! \wedge N > 0\}}$$

Figure 19: Corresponding derivation tree

Therefore, we instantiate $\underline{P}$ with $x = N \wedge x > 0$ and $\underline{R}$ with $y = N! \wedge N > 0$. In the GUI, this can be done by dragging the assertion $\{x = N \wedge x > 0\}$ and dropping it on the assertion $\{\underline{P}\}$, and dragging the assertion $\{y = N! \wedge N > 0\}$ and dropping it on the assertion $\{\underline{R}\}$.

This leads to the result, that our tree is going to like Figure 20.

$$\dfrac{\overset{?}{\{x = N \wedge x > 0\}\ \texttt{y:=1}\ \{\underline{Q}\}} \qquad \overset{?}{\{\underline{Q}\}\ \texttt{while !(x=1) do s end}\ \{y = N! \wedge N > 0\}}}{\{x = N \wedge x > 0\}\ \texttt{skip; x:=0}\ \{y = N! \wedge N > 0\}}\ (\text{Seq})$$

$$\overset{?}{\{x = N \wedge x > 0\}\ \texttt{skip; x:=0}\ \{y = N! \wedge N > 0\}}$$

Figure 20: Derivation tree after unifying $\underline{P}$ and $\underline{R}$ with concrete assertions

As the precondition of the root triple and the precondition of the rule for sequential composition are syntactically equal, and the postcondition of the root triple and the postcondition of the rule for sequential composition are syntactically equal, the assistant recognizes that the hole between the root triple and the rule for sequential composition can be removed.

In order to do that, the assistant automatically introduces the rule of consequence, with the side condition that $\{x = N \wedge x > 0\} \vDash \{x = N \wedge x > 0\}$ and $\{y = N! \wedge N > 0\} \vDash \{y = N! \wedge N > 0\}$, which is trivially true, which leads to the result which is illustrated in Figure 21.

$$\dfrac{\dfrac{\overset{?}{\{x = N \wedge x > 0\}\ \texttt{y:=1}\ \{Q\}} \qquad \overset{?}{\{Q\}\ \texttt{while !(x=1) do s end}\ \{y = N! \wedge N > 0\}}}{\{x = N \wedge x > 0\}\ \texttt{skip; x:=0}\ \{y = N! \wedge N > 0\}}\ (\text{Seq})}{\{x = N \wedge x > 0\}\ \texttt{skip; x:=0}\ \{y = N! \wedge N > 0\}}\ (\text{Cons})$$

Figure 21: Derivation tree after introducing the rule of consequence

Finally, as an optimization, the rule of consequence can be removed and the final result of this step can be seen in Figure 22.

$$\dfrac{\overset{?}{\{x = N \wedge x > 0\}\ \texttt{y:=1}\ \{Q\}} \qquad \overset{?}{\{Q\}\ \texttt{while !(x=1) do s end}\ \{y = N! \wedge N > 0\}}}{\{x = N \wedge x > 0\}\ \texttt{skip; x:=0}\ \{y = N! \wedge N > 0\}}\ (\text{Seq})$$

Figure 22: Derivation tree after removing the rule of consequence

Step 5: Applying the while rule

After the hole was removed, the proof outline is the one shown in Figure 23. As a next step, the user might want to apply the while rule for total correctness, which is the one displayed below.

$$\frac{\{\underline{b} \wedge \underline{P} \wedge \underline{e} = \underline{Z}\}\ \underline{s}\ \{\Downarrow \underline{P} \wedge \underline{e} < \underline{Z}\}}{\{\underline{P}\}\ \texttt{while}\ \underline{b}\ \texttt{do}\ \underline{s}\ \texttt{end}\ \{\Downarrow \neg\underline{b} \wedge \underline{P}\}}\ (\text{WhTot})$$

As can be seen, in order to apply the while rule for total correctness, the user has to introduce a new logical variable which isn't used yet and has to specify what our loop variant should be.

The user can do that using the text boxes that come with the rule application dialog. Figure 24 shows how the rule application dialog would look if the user chose to use Z as the logical variable and x as the loop variant.

```
{x=N∧x>0}
?
y:=1;
?
{Q}
?
while !(x=1) do
   y:=y*x;
   x:=x-1;
end
?
{y=N!∧N>0}
```

Figure 23: Proof outline after removing a hole

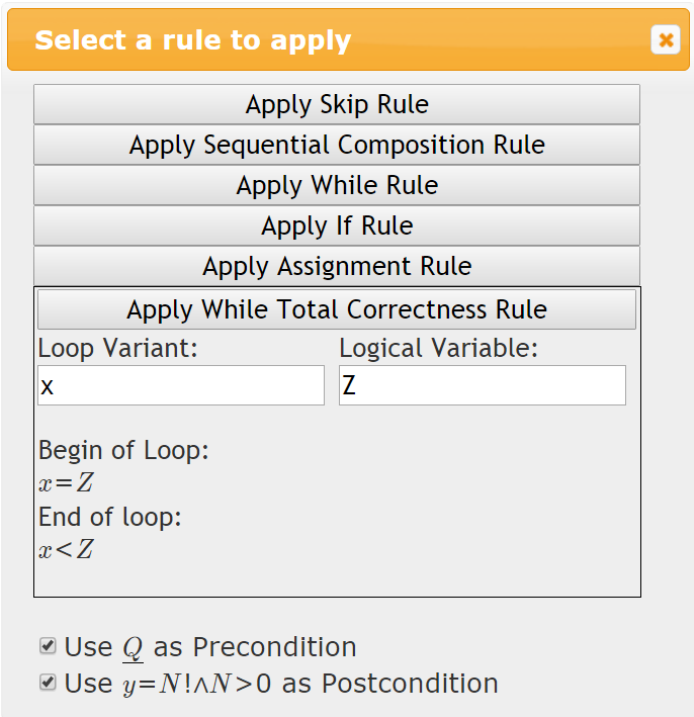


Figure 24: Rule application dialog for applying the while rule for total correctness

If the user wants the assistant to unify the pre-/postcondition of the rule with the pre-/postcondition of the triple to which they applied the rule, they can make use of the two check boxes which are on the bottom of the rule application dialog.

If the user clicked on the "Apply While Total Correctness Rule" button and the two check boxes at the bottom of the dialog were checked, the assistant would do the following three steps:

At first, the assistant introduces the while rule for total correctness, which leads to the result that our derivation tree now looks like Figure 25.

$$\dfrac{\{x = N \wedge x > 0\}\ \texttt{y:=1}\ \{\underline{Q}\} \qquad \dfrac{\overset{?}{\{\neg(x = 1) \wedge \underline{S} \wedge x = Z\}\ \texttt{s}\ \{\Downarrow \underline{S} \wedge x < Z\}}}{\{\underline{S}\}\ \texttt{while !(x=1) do s end}\ \{\Downarrow \neg\neg(x = 1) \wedge \underline{S}\}}\ \text{(WhTot)}}{\{x = N \wedge x > 0\}\ \texttt{skip; x:=0}\ \{y = N! \wedge N > 0\}}\ \text{(Seq)}$$

Figure 25: Derivation tree after introducing the while rule for total correctness

As a second step, the assistant unifies the precondition of the triple ($\{\underline{Q}\}$) with the precondition of the rule ($\{\underline{S}\}$), which leads to the result that $\underline{Q}$ is instantiated with $\underline{S}$, which leads to the derivation tree which is displayed in Figure 26.

$$\dfrac{\{x = N \wedge x > 0\}\ \texttt{y:=1}\ \{\underline{Q}\} \qquad \dfrac{\overset{?}{\{\neg(x = 1) \wedge \underline{S} \wedge x = Z\}\ \texttt{s}\ \{\Downarrow \underline{S} \wedge x < Z\}}}{\{\underline{S}\}\ \texttt{while !(x=1) do s end}\ \{\Downarrow \neg\neg(x = 1) \wedge \underline{S}\}}\ \text{(WhTot)}}{\{x = N \wedge x > 0\}\ \texttt{skip; x:=0}\ \{y = N! \wedge N > 0\}}\ \text{(Seq)}$$

Figure 26: Derivation tree after unifying the preconditions

As a third step, the assistant tries to unify $\{y = N! \wedge N > 0\}$ and $\{\Downarrow \neg\neg(x = 1) \wedge \underline{S}\}$, which is not possible because no matter how $\underline{S}$ is instantiated, the two assertions won't be syntactically equal .

Therefore, the assistant informs the user that this is not possible by displaying the failure dialog that can be seen in Figure 27 on the next page.

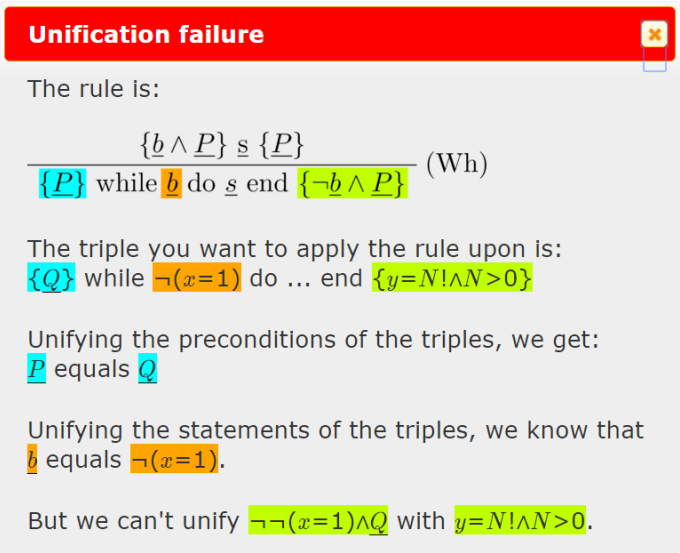


Figure 27: Failure dialog

As the user can't use both the precondition and the postcondition, they might decide that the assistant should neither use the precondition nor the postcondition, which would lead to the proof outline displayed in Figure 28. Note that the user could have used only the postcondition, but let us assume for the sake of demonstration that the user preferred using neither the precondition nor the postcondition.

What also can be seen in Figure 28 is that the assistant printed a star next to the while statement, to indicate that there is a side condition and to give the user the possibility to argue why this side condition holds (for more information, refer to step 12).

$\{x=N\wedge x>0\}$
?
y:=1;
?
$\{\underline{Q}\}$
?
$\{\underline{INV_0}\}$
while !(x=1) do *0
  $\{\neg(x=1)\wedge\underline{INV_0}\wedge x=Z\}$
  ?
  y:=y*x;
  x:=x-1;
  ?
  $\{\Downarrow\underline{INV_0}\wedge x<Z\}$
end
$\{\neg(\neg(x=1))\wedge\underline{INV_0}\}$
?
$\{y=N!\wedge N>0\}$

Figure 28: Proof outline resulting from applying the while rule for total correctness

Step 6: Applying several rules

What the user could do now is to instantiate the meta-variable INV_0, but as finding the a loop invariant which is not too strong and not to weak is rather difficult, and because we have the freedom to leave the loop invariant unspecified, we keep on working on the proof outline without specifying the loop invariant. The next steps are not explained in detail anymore, as they are similar to the steps before, we just show how the proof outline looks after every single step.

$\{x=N \wedge x>0\}$
?
y:=1;
?
$\{\underline{Q}\}$
?
$\{\underline{INV_0}\}$
while !(x=1) do*0
  $\{\neg(x=1) \wedge \underline{INV_0} \wedge x=Z\}$
  ?
  y:=y*x;
  ?
  $\{\underline{U}\}$
  ?
  x:=x-1;
  ?
  $\{\Downarrow \underline{INV_0} \wedge x<Z\}$
end
$\{\neg(\neg(x=1)) \wedge \underline{INV_0}\}$
?
$\{y=N! \wedge N>0\}$

Figure 29: Proof outline after applying the rule for sequential composition to the statement which is inside the while loop and unifying the corresponding preconditions and postconditions

$\{x=N \wedge x>0\}$
?
y:=1;
?
$\{\underline{Q}\}$
?
$\{\underline{INV_0}\}$
while !(x=1) do*0
  $\{\neg(x=1) \wedge \underline{INV_0} \wedge x=Z\}$
  ?
  y:=y*x;
  ?
  $\{\underline{INV_0}[x \mapsto x\text{-}1] \wedge x\text{-}1<Z\}$
  x:=x-1
  $\{\Downarrow \underline{INV_0} \wedge x<Z\}$
end
$\{\neg(\neg(x=1)) \wedge \underline{INV_0}\}$
?
$\{y=N! \wedge N>0\}$

Figure 30: Proof outline after working backwards by applying the assignment rule to `x:=x-1` and unifying the corresponding preconditions and postconditions

$\{x{=}N \wedge x{>}0\}$

?

$\boxed{y{:=}1;}$

?

$\{\underline{Q}\}$

?

$\{\underline{INV_0}\}$

$\boxed{\text{while !(x=1) do}}$*0

$\{\neg(x{=}1) \wedge \underline{INV_0} \wedge x{=}Z\}$

?

$\{\underline{INV_0}[x \mapsto x\text{-}1][y \mapsto y{\cdot}x] \wedge x\text{-}1{<}Z\}$

$\boxed{\text{y:=y*x}}$

$\{\underline{INV_0}[x \mapsto x\text{-}1] \wedge x\text{-}1{<}Z\}$

$\boxed{\text{x:=x-1}}$

$\{\Downarrow \underline{INV_0} \wedge x{<}Z\}$

$\boxed{\text{end}}$

$\{\neg(\neg(x{=}1)) \wedge \underline{INV_0}\}$

?

$\{y{=}N! \wedge N{>}0\}$

Figure 31: Proof outline after working backwards by applying the assignment rule to `y:=y*1` and unifying the corresponding postconditions

$\{x{=}N \wedge x{>}0\}$

?

$\{\underline{Q}[y \mapsto 1]\}$

$\boxed{\text{y:=1}}$

$\{\underline{Q}\}$

?

$\{\underline{INV_0}\}$

$\boxed{\text{while !(x=1) do}}$*0

$\{\neg(x{=}1) \wedge \underline{INV_0} \wedge x{=}Z\}$

?

$\{\underline{INV_0}[x \mapsto x\text{-}1][y \mapsto y{\cdot}x] \wedge x\text{-}1{<}Z\}$

$\boxed{\text{y:=y*x}}$

$\{\underline{INV_0}[x \mapsto x\text{-}1] \wedge x\text{-}1{<}Z\}$

$\boxed{\text{x:=x-1}}$

$\{\Downarrow \underline{INV_0} \wedge x{<}Z\}$

$\boxed{\text{end}}$

$\{\neg(\neg(x{=}1)) \wedge \underline{INV_0}\}$

?

$\{y{=}N! \wedge N{>}0\}$

Figure 32: Proof outline after working backwards by applying the assignment rule on `y:=1` and unifying the corresponding postconditions

$\{x{=}N \wedge x{>}0\}$

?

$\{\underline{INV_0}[y{\mapsto}1]\}$

$\boxed{\texttt{y:=1}}$

$\{\underline{INV_0}\}$

$\boxed{\texttt{while !(x=1) do}}^{*0}$

  $\{\neg(x{=}1) \wedge \underline{INV_0} \wedge x{=}Z\}$

  ?

  $\{\underline{INV_0}[x{\mapsto}x\text{-}1][y{\mapsto}y{\cdot}x] \wedge x\text{-}1{<}Z\}$

  $\boxed{\texttt{y:=y*x}}$

  $\{\underline{INV_0}[x{\mapsto}x\text{-}1] \wedge x\text{-}1{<}Z\}$

  $\boxed{\texttt{x:=x-1}}$

  $\{\Downarrow \underline{INV_0} \wedge x{<}Z\}$

$\boxed{\texttt{end}}$

$\{\neg(\neg(x{=}1)) \wedge \underline{INV_0}\}$

?

$\{y{=}N! \wedge N{>}0\}$

Figure 33: Proof outline after instantiating $\underline{Q}$ with $\underline{INV_0}$

Step 7: Instantiating the loop invariant

At this point, the proof outline consists of three holes and one meta-variable, namely the loop invariant. If we wanted to instantiate the loop invariant now, we could do that by double clicking on the INV_0 and entering the concrete boolean formula in the dialog that is displayed (Figure 34).

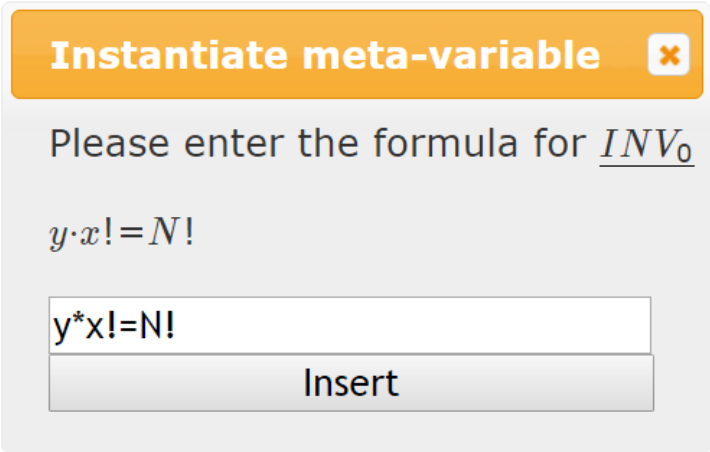


Figure 34: Instantiation dialog

As can be seen, we want to instantiate $\underline{INV_0}$ by $y \cdot x! = N!$. As we entered the formula, the assistant again tried to parse it immediately and pretty-printed it above the input box.

If we now click on the insert button, $\underline{INV_0}$ is going to be instantiated by $y \cdot x! = N!$ in all assertions. Additionally, if this meta-variable was occurs a substitution, the substitution us automatically applied because the concrete formula is now known.

As a result, our proof outline now looks like Figure 35, which can be found on the next page.

Step 8: Adding entailments

After having instantiated the loop invariant, we recognize that the only thing left to be done is the introduction of the rule of consequence, such that the remaining three holes can be removed. Introducing the rule of consequence can be done by dragging the entailment symbol ($\models$) and dropping it onto a hole (e.g. a red question mark). But there is a problem: Consider the last two assertions in Figure 35, namely

$$\neg(\neg(x = 1)) \wedge y \cdot x! = N!$$

and

$$y = N! \wedge N > 0$$

When we use the rule of consequence, we have to argue why

$$\begin{gathered}\neg(\neg(x = 1)) \wedge y * x! = N! \\ \models \\ y = N! \wedge N > 0\end{gathered}$$

is true, which cannot be done, because our loop invariant is too weak. Therefore, we have to go back and instantiate the loop invariant with another formula

$\{x=N \wedge x>0\}$
?
$\{1 \cdot x!=N!\}$
y:=1
$\{y \cdot x!=N!\}$
while !(x=1) do ^{*0}
  $\{\neg(x=1) \wedge y \cdot x!=N! \wedge x=Z\}$
  ?
  $\{y \cdot x \cdot (x-1)!=N! \wedge x-1<Z\}$
  y:=y*x
  $\{y \cdot (x-1)!=N! \wedge x-1<Z\}$
  x:=x-1
  $\{\Downarrow y \cdot x!=N! \wedge x<Z\}$
end
$\{\neg(\neg(x=1)) \wedge y \cdot x!=N!\}$
?
$\{y=N! \wedge N>0\}$

Figure 35: Proof outline after instantiating INV_0

Step 9: Undo the previously done steps

When we want to undo and redo steps previously done, we could use the buttons for undo and redo which can be found in the menu, use the shortcuts CTRL + Z / CTRL + Y, or, if we want to jump to a specific step we can use the history view which can be shown by clicking on the the button which shows a book icon.

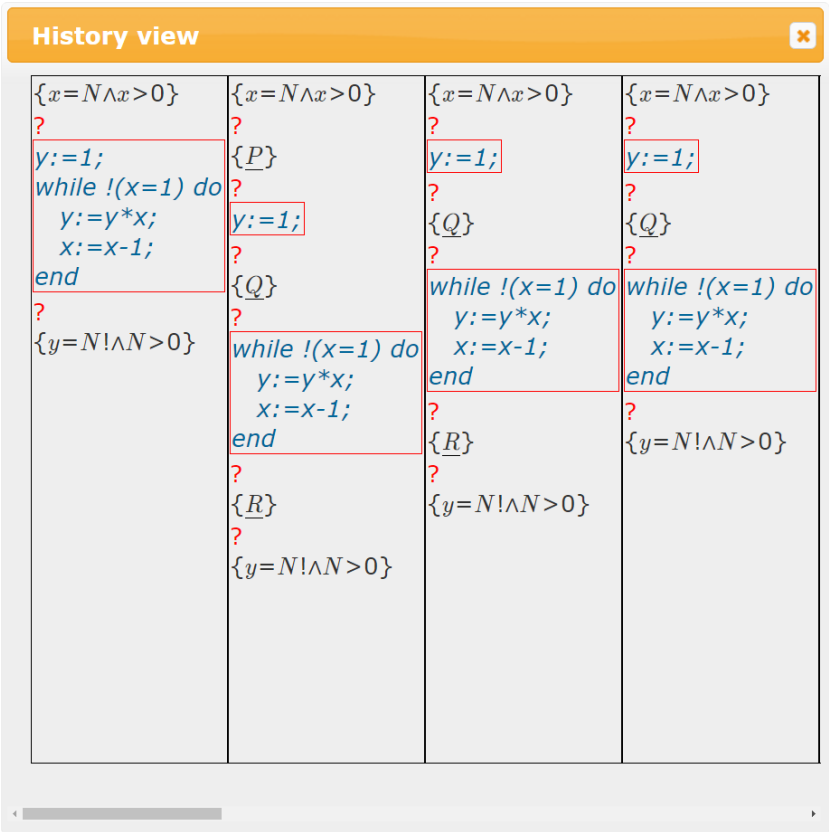


Figure 36: Screenshot of the history view

As can be seen in Figure 36, the whole history back to the first step can be viewed and by clicking on the corresponding outline we can jump to any previous step we want.

After we jumped back to the step before we instantiated the invariant, i.e. return to after performing step 6, we can this time use an invariant that is strong enough, namely

$$y * x! = N! \wedge N \geq x \wedge x > 0$$

and as the invariant is now strong enough, we can proceed with the proof by adding entailments, i.e. continue with the previously failed step 8.

Step 10: Adding entailments

After we removed the three holes by introducing the rule of consequence three times, we get the outline which can be seen in Figure 38.

$\{x=N \wedge x>0\}$
$\vDash_0$
$\{1 \cdot x!=N! \wedge N \geq x \wedge x>0\}$
$\boxed{\texttt{y:=1}}$
$\{y \cdot x!=N! \wedge N \geq x \wedge x>0\}$
$\boxed{\texttt{while !(x=1) do}}^{*0}$
  $\{\neg(x=1) \wedge y \cdot x!=N! \wedge N \geq x \wedge x>0 \wedge x=Z\}$
  $\vDash_1$
  $\{y \cdot x \cdot (x-1)!=N! \wedge N \geq x-1 \wedge x-1>0 \wedge x-1<Z\}$
  $\boxed{\texttt{y:=y*x}}$
  $\{y \cdot (x-1)!=N! \wedge N \geq x-1 \wedge x-1>0 \wedge x-1<Z\}$
  $\boxed{\texttt{x:=x-1}}$
  $\{\Downarrow y \cdot x!=N! \wedge N \geq x \wedge x>0 \wedge x<Z\}$
$\boxed{\texttt{end}}$
$\{\neg(\neg(x=1)) \wedge y \cdot x!=N! \wedge N \geq x \wedge x>0\}$
$\vDash_2$
$\{y=N! \wedge N>0\}$

Figure 37: The finished proof outline

Step 11: Justify the entailments

What is left to do is to justify the side conditions of the applications of the rule of consequence. This can be done by double-clicking on the inserted entailment symbols and entering the justification into the text area of the dialog that is displayed (Figure 38).

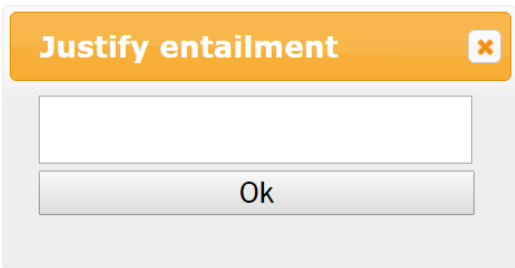


Figure 38: Dialog in which the entailment can be justified

Note that the assistant does not check whether or not the entailment is actually true, if this were an exercise, the teaching assistants would need to check the entailments. But if the entailments are correct, the whole proof outline is going to be correct as well.

Step 12: Justify the total correctness rule

In addition to arguing why the side conditions of the applications of the rule of consequence hold, we also have to argue why the side condition of the while rule for total correctness is true, which can be done by double-clicking on the star symbol near the while statement, and by filling in the dialog displayed in Figure 39.

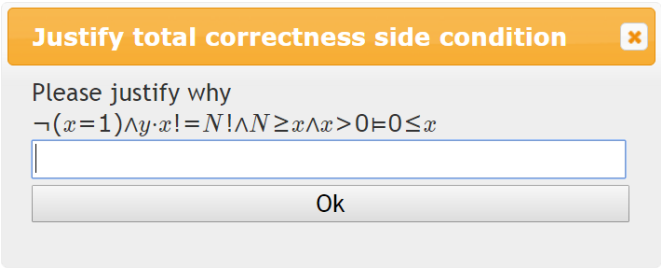


Figure 39: Dialog to argue why the side condition of the while rule for total correctness holds

Similar to before, the assistant also does not check whether or not the side condition is actually true.

Step 13: Storing and printing the proof outline

As the proof is now finished, it can be saved such that it can be reviewed later. Additionally, the proof outline can also be printed on paper or stored as a PDF file. The justifications of the entailments and of the while rule for total correctness side condition are printed as well. The proof outline as it is created by the assistant can be found in appendix C.

If the user forgot to provide some justifications, they get a warning which tells them which justifications are missing.

6 Implementation

6.1 The Derivation Tree

6.1.1 The Model

Figure 40 shows an ER model of how the derivation tree is modeled by the program. In the implementation, the entities are classes and the relations indicate the references between the objects of the classes. The following subsections describe how the different classes are used to represent a derivation tree.

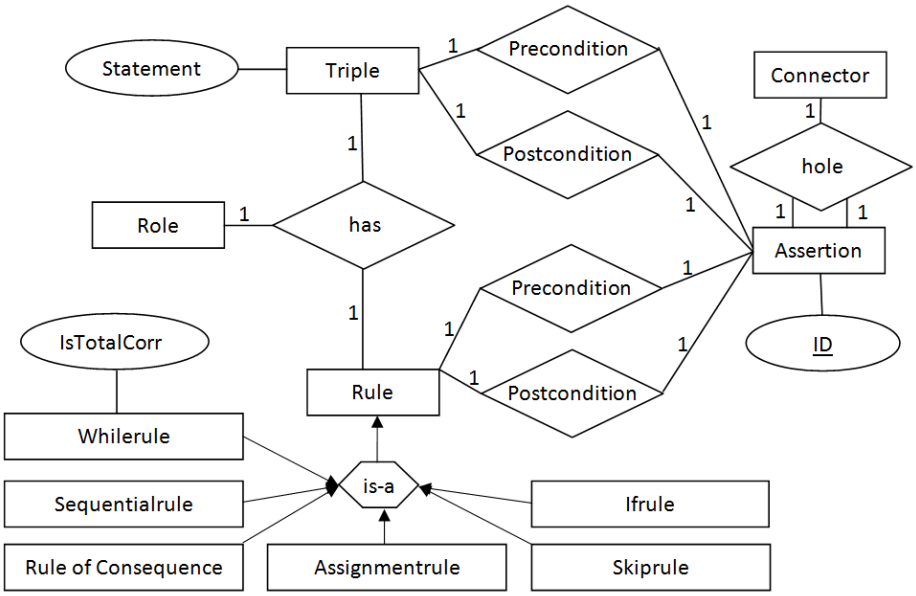


Figure 40: ER model of the derivation tree

6.1.1.1 Triple

The triple class is used to represent a Hoare triple, therefore it has a precondition, a postcondition and a statement. A triple always has a reference to the rule below it in the derivation tree and also to the rule above it, if there is actually a rule above.

There is one special triple, namely the root triple (the triple from which we start building the proof outline), it does not have a rule below.

6.1.1.2 Rule

A rule has a reference of the triple below it in the derivation tree and depending on the type of the rule one or two references to the triple(s) above (the rule for sequential composition and the if rule have two triples above, all others have one).

As mentioned earlier, a rule also has a pre- and postcondition, which are used to determine whether there is a hole between a rule and its triple below (if the pre- and postcondition of the triple are different from the pre- and postcondition of the triple below, there is a hole between them).

6.1.1.3 Assertion

Assertions are stored as an AST (abstract syntax tree). Each assertion has a unique id. Note that it is possible for assertions with different ids to have the same AST, even if two assertions are unified, each of them keeps their id and just the AST is exchanged. The reason, why it is done like this is that we need to know where an assertion in the tree is located to make the user interactions provided by the assistant possible. If the user for example drags the entailment symbol on a hole between two assertions, we have to make sure that the rule of consequence is introduced between those two assertions and not between arbitrary assertions who have the same AST.

6.1.1.4 Connector

In Figure 41, you can see a proof represented as a derivation tree and as a proof outline.

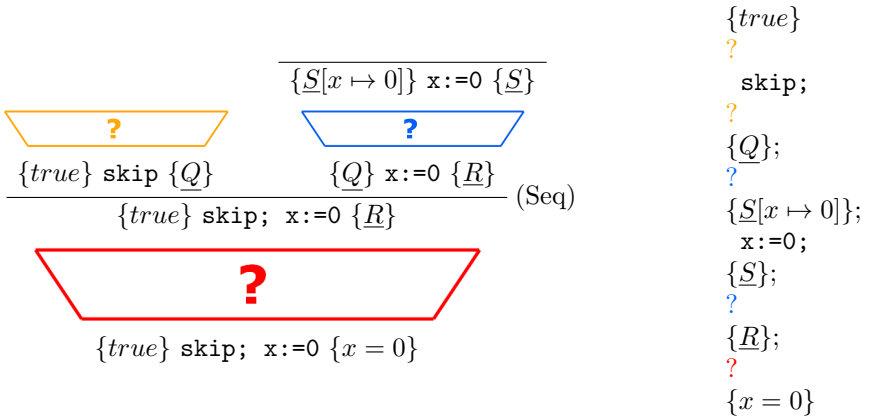


Figure 41: Representation of a proof as derivation tree and as proof outline

What can be observed is that the red hole in the derivation tree appears only once in the proof outline, while the blue hole in the derivation tree appears twice in the proof outline (there are two blue question marks in the proof outline).

When it comes to modeling holes of the proof outline, the assistant makes use of connectors. One connector represents one hole in the proof outline, therefore it is possible that one hole in the derivation tree is modeled by two connectors in the proof outline. For example, in Figure 41, only one connector for the red hole is needed, as the red hole appears only once in the proof outline, but two connectors for the blue hole and two connectors for the orange hole are needed, as they both appear twice in the proof outline.

6.1.2 The Abstract Syntax Tree and the Visitor Pattern

In the implementation, assertions and statements are stored as abstract syntax trees. Pretty-printing the tree in the browser, manipulating the tree and other tasks on the trees are then done by using the visitor pattern [3].

6.2 Unification

When it comes to unification of two formulas, the implementation manages to reduce the problem of unifying two AST's to the problem of unifying a conjunction of terms.

Therefore, we can view a formula f as $\bigwedge_i t_i$. Additionally, the implementation of assistant ensures that the following properties of a formula are true:

1. Only one meta-variable occurs in each formula
2. Each term t_i is either a meta-variable or doesn't contain a meta-variable

Note that ensuring point 1 is not difficult for the assistant, as meta-variables can only be created by applying a rule and no rule introduces an assertion that has more than one meta-variable.

When it comes to unifying two formulas now, the unification algorithm that was used in the assistant makes use of the guarantees given and manages to unify two formulas in O(n) (n the number of terms in the formulas), which is optimal and would be difficult to be achieved if a standard unification algorithm were used. The pseudo code of the algorithm used by the assistant can be found in appendix D

Additionally, we want to illustrate how the algorithm works by showing how the algorithm would unify the formulas

$$a \wedge b \wedge c \wedge \underline{P} \wedge g \wedge h \wedge i \wedge j \wedge k \wedge l$$
and
$$a \wedge b \wedge c \wedge d \wedge e \wedge f \wedge \underline{Q} \wedge l$$

where $\underline{P}$ and $\underline{Q}$ are meta-variables and the other letters are terms that don't contain any meta-variables.

The unification algorithm works the following way:

Step 1: Match terms at the beginning of the formulas

As a first step we start from the beginning of each formula and look at the first terms.

- If two terms are equal, we consume these terms and look at the next two terms.
- If the two terms are not equal and none of them is a meta variable, we output that the formulas can't be unified and are finished.
- If they are not equal and one of them is a meta-variable, we go to step 2.

After having executed the second step, the formulas that we keep working on are

$$\underline{P} \wedge g \wedge h \wedge i \wedge j \wedge k \wedge l$$
$$\text{and}$$
$$d \wedge e \wedge f \wedge \underline{Q} \wedge l$$

Step 2: Match terms at the end of the formulas

In the second step, the same algorithm as described in step 1 is applied, with the difference that we start at the end of both formulas and work backwards.

After having executed the second step, the formulas that we keep working on are

$$\underline{P} \wedge g \wedge h \wedge i \wedge j \wedge k$$
$$\text{and}$$
$$d \wedge e \wedge f \wedge \underline{Q}$$

Step 3: Replace meta-variables the terms

After having done the first two steps, we know that now one formula has the meta-variable at the end and the other has the meta-variable at the beginning.

Let f_1 be the formula which has the meta-variable ($\underline{v_1}$) at the start and f_2 be the formula which has the meta-variable ($\underline{v_2}$) at the end. What we now can do to get a correct unification is to instantiate $\underline{v_1}$ with all but the last term from f_2 and $\underline{v_2}$ with all but the first term from f_1.

In our example, this would mean that

$$\underline{P} \equiv d \wedge e \wedge f$$
$$\text{and}$$
$$\underline{Q} \equiv g \wedge h \wedge i \wedge j \wedge k$$

6.3 Redo/Undo

When a user builds a proof outline, the assistant records every step done by the user by storing the functions that were invoked as a result of the user interaction. If the user uses the history view to jump back to a certain step, the program resets everything to the initial state and invokes all the functions that were called until this step.

The advantage of this implementation is that the amount of information which needs to be stored to reconstruct a proof outline is quite small, but the downside is that all functions that were called from the start have to be re-executed when a user does an undo or a redo, but fortunately re-executing the necessary functions is fast such that the user experience is not affected at all.

6.4 Statistics

At the moment, all steps done by the user are stored in a MySQL database. Concretely, for each step a triple is stored in the data base, which consists of the ID of the session, the time the user did the interaction and a description of the interaction. With this data available, it is possible to find out how many users the tool has, how much time a user spent using the proof outline in one session, what the most common mistakes are that were done an so forth. To make sure that the privacy of the users is maintained, the data is stored anonymously.

7 Quality assurance

At the moment, there are in total 32 test available. There are two different types of tests, there are Unit tests and tests that recreate the proof outline.

7.1 Testing by recreation of the proof outline

21 of the tests that are available, are tests which work by recreating a previously build proof outline. Each test case consists of the invoked functions that were used to build the proof outline and a representation of the final derivation tree obtained from executing the recorded functions. What happens when the test case is invoked is that the assistant is redoing all the steps by invoking the functions that we used to build the proof outline and the resulting proof outline is then compared to the proof outline stored in the test case. Therefore, if one would like to write a test case for a part of the implementation of the assistant, one could build a proof outline which makes only use of that part of the assistant.

The test case set can easily be extended, as the assistant allows to store a created proof outline as a test case.

7.2 Unit testing

To test the implemented unification algorithm in the assistant, there are also 11 unit test cases available, which has can be used to test the unification algorithm.

8 Used tools

8.1 jQuery

jQuery [4] is a powerful JavaScript library. It provides easy ways of manipulating DOM elements, create animations, handling events and also handling requests via AJAX.

In the thesis, jQuery is used to create the dialogs, manipulate the GUI and to handle the connection between the client-side JavaScript code and the server.

8.2 Ace editor

The Ace editor [5] is an embeddable code editor written in JavaScript, which can easily be included in any web page.

In this thesis, the Ace editor was used, because it provides an easy way of enabling the user to enter a program in an editor-like textbox and getting the entered text with JavaScript.

8.3 Peg.js

Peg.js [6] is a parser generator, which generates parsers written in JavaScript. In addition to the grammar, from which the parser should be generated, a piece of JavaScript code can also be specified for each pattern, which is executed when the pattern matches successfully.

Peg.js uses Parsing Expression Grammars (PEG) [7], which are an alternative to context-free grammars and regular expressions. PEGs solve ambiguities and the problems that come with it like the dangling else problem by a prioritized choice operator. In PEGs, the dangling else problem can be solved by the following grammar:

```
Statement <- IF Cond THEN Statement ELSE Statement
           /  IF Cond THEN Statement
           /  ...
```

If the first pattern is matched, all following patterns are ignored, therefore ambiguity is resolved by priority. This means, that the longest patterns should have the highest priorities, otherwise it could be that a shorter pattern is matched, although we have a longer pattern, that should be matched, which could lead to a parsing error.

The advantage of this approach is that ambiguity resolution is precisely defined, but the downside is that language syntax design gets more complicated. If patterns aren't ordered well, no warning and no hint is going to be displayed, only at runtime will be detected that some inputs don't get parsed. This means that such a parser should be tested well, which this thesis does by having corresponding test cases. In this thesis, Peg.js is used to parse the IMP language and mathematical formulas.

8.4 Rendering library

From the start of the thesis it was clear, that a rendering library needs to be used for pretty-printing the formulas. Two rendering libraries, namely MathJax [8] and KaTeX [9] were taken into consideration, both are JavaScript based rendering libraries, which work in all major browsers, are easy to include and don't need a server in the background.

When deciding, which rendering library is the best fit for this thesis, three criteria were taken into consideration:

8.4.1 Readability of formula

Below is the same formula rendered with MathJax and KaTeX. Both renderings are acceptable.

$$\sqrt{\left\lceil \frac{1^2}{2^{42}} \right\rceil} + \sum_{x=1}^{2} x!$$

Figure 42: MathJax rendering

$$\sqrt{\left\lceil \frac{1^2}{2^{42}} \right\rceil} + \sum_{x=1}^{2} x!$$

Figure 43: KaTeX rendering

8.4.2 Supported functions

MathJax supports all functions that are needed for the thesis. KaTeX on the other side, does for example not support \leq (≤), \geq (≥), \Downarrow (⇓) and \underline (_), which is needed in the thesis.

8.4.3 Speed

Another important criterion is the speed, with which formulas get rendered. If the formulas get rendered too slow, the users might loose their patience and stop using the tool. Therefore, the speed of the two libraries was compared. The differences between the two libraries are huge: While KaTeX manages to render approximately 10'000 formulas per second, MathJax only manages to render 3-4. Even with caching the results, MathJax would need up to ten seconds to render a complete proof outline.

8.4.4 Conclusion

Because of the huge speed advantage of KaTeX and the fact, that there is a workaround for the functions which are not supported by KaTeX, the decision was made to use KaTeX in this thesis. If MathJax were used, the waiting time would be too long and could frustrate the user.

9 Conclusion and Future Work

The result of this thesis is a proof assistant, which helps the students building proof outlines. Although the proof assistant is ready to be used, there is also room for future work.

One thing that would be useful to have is an automatic verifier, which automatically checks whether the entailments from the instances of the rule of consequence that were used in the proof outline are actually valid.

It would also be nice to have a GUI in which the data from the statistics get displayed in a nice way. At the moment, doing an evaluation of the stored data can only be done using MySQL queries.

In addition to that, the GUI of the assistant could be further improved. One improvement that could be done is to enable user to view, how the derivation tree corresponding to a given proof outline looks, as the relation between proof outlines and derivation trees is not always straightforward.

What also could be done is to extend the language IMP. Possible features that could be added are:

- Heap allocation
- Pointers
- Variable declarations
- Procedures
- Concurrency

And finally, as the assistant is there to improve the learning phase and there was no lecture that introduced Hoare logics in the time the thesis was done, it was not possible to do a field studies to find out how beneficial the assistant actually is and what could be improved to increase the gain that comes with the assistant even further.

Acknowledgements

Finally, I would like to thank all the people that supported my during my thesis, a special thank goes to Malte Schwerhoff, who was a great support during this thesis.

Additionally, I also want to thank Prof. Dr. Peter Müller for making this thesis possible.

References

[1] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[2] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[4] jquery. `https://jquery.com/`.

[5] Ace editor. `https://ace.c9.io/`.

[6] Peg.js. `http://pegjs.org/`.

[7] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, January 2004.

[8] Mathjax. `https://www.mathjax.org/`.

[9] Katex. `https://khan.github.io/KaTeX/`.

Appendices

A Comparison derivation trees vs proof outlines

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{y = Y \wedge x = X\}\texttt{z:=x}\{y = Y \wedge z = X\}}\,(\text{Ass})}{\{y = Y \wedge x = X\}\texttt{z:=x;x:=y}\{x = Y \wedge z = X\}}\,(\text{Seq})}{\{y = Y \wedge z = X\}\texttt{x:=y}\{x = Y \wedge z = X\}}\,(\text{Ass})}{\dfrac{\{y = Y \wedge x = X\}\texttt{(z:=x;x:=y);y:=z}\{x = Y \wedge y = X\}}{\{x = X \wedge y = Y\}\texttt{(z:=x;x:=y);y:=z}\{x = Y \wedge y = X\}}\,(\text{Cons})}\,(\text{Seq})$$

Proof presented as a derivation tree

$\{x = X \wedge y = Y\}$
$\models$
$\{y = Y \wedge x = X\}$
`z:=x`
$\{y = Y \wedge z = X\}$
`x:=y`
$\{x = Y \wedge z = X\}$
`y:=z`
$\{x = Y \wedge y = X\}$

Proof presented as a proof outline

B Supported syntax for assertions

Syntax	**Symbol**	**Comment**
Boolean operators		
&&	$\wedge$	and
||	$\vee$	or
=>	$\Rightarrow$	implies
!	$\neg$	not
Boolean values		
true	$true$	true
false	$false$	false
Relational operators		
=	$=$	equals
!=	$\neq$	not equals
<	$<$	less than
<=	$\leq$	less or equal
>	$>$	greater than
>=	$\geq$	greater or equal
Arithmetic operators		
+	$+$	plus
-	$-$	minus
*	$\cdot$	multiply
/	$/$	divide
^	^	pow
Functions		
a!	$a!$	factorial
sqrt(a)	$\sqrt{a}$	square root
ceil(a)	$\lceil a \rceil$	ceiling
floor(a)	$\lfloor a \rfloor$	floor
foo(a,b,c)	$foo(a, b, c)$	arbitrary function
Sums, Products		
sum(n=0,N,i)	$\sum_{n=0}^{N} i$	sum over i from n to N
prod(n=0,N,i)	$\prod_{n=0}^{N} i$	product over i from n to N

C Printed proof outline

Solution

$\{x=N \wedge x>0\}$
$\vDash_0$
$\{1 \cdot x!=N! \wedge N \geq x \wedge x>0\}$
$\boxed{\text{y:=1}}$
$\{y \cdot x!=N! \wedge N \geq x \wedge x>0\}$
$\boxed{\text{while !(x=1) do}}^{*0}$
  $\{\neg(x=1) \wedge y \cdot x!=N! \wedge N \geq x \wedge x>0 \wedge x=Z\}$
  $\vDash_1$
  $\{y \cdot x \cdot (x\text{-}1)!=N! \wedge N \geq x\text{-}1 \wedge x\text{-}1>0 \wedge x\text{-}1<Z\}$
  $\boxed{\text{y:=y*x}}$
  $\{y \cdot (x\text{-}1)!=N! \wedge N \geq x\text{-}1 \wedge x\text{-}1>0 \wedge x\text{-}1<Z\}$
  $\boxed{\text{x:=x-1}}$
  $\{\Downarrow y \cdot x!=N! \wedge N \geq x \wedge x>0 \wedge x<Z\}$
$\boxed{\text{end}}$
$\{\neg(\neg(x=1)) \wedge y \cdot x!=N! \wedge N \geq x \wedge x>0\}$
$\vDash_2$
$\{y=N! \wedge N>0\}$

Entailment Justifications:
0:
from x=N follows 1*x! = N!
from x=N follows N>=x
from x>0 follows x>0
1:
from !(x=1) and x>0 follows that x >=2, therefore
from y*x!=N! follows y*x*(x-1)! = N!,
from N>=x-1 follows N>=x-1,
from x>=2 follows x-1>0,
from x=Z follows x-1 < Z
2:
from !!(x=1) follows x=1, therefore
from y*x! = N!, which is y*1! = N! follows y=N!
and from N>=x and x>0 follows N>0

Total Correctness sidecondition:
0:
from x>0 follows 0<=x

D Unification algorithm

```
 Step 1:

while true
{
  t_1 = first term of formula f;
  t'_1 = first term of formula f';
  if t_1 == t_2
  {
    remove t_1 from f;
    remove t'_1 from f';
  }
  else
  {
    if(!isMeta(t_1) && !isMeta(t'_1))
    {
      return "Not unifiable"
    }
    else
    {
      goto Step2
    }
  }
} return;

  Step 2:

while true
{
  t_m = last term of formula f;
  t'_n = last term of formula f';
  if t_m == t'_n
  {
    remove t_m from f;
    remove t'_n from f';
  }
  else
  {
    if(!isMeta(t_m) && !isMeta(t'_n))
    {
      return "Not unifiable"
    }
    else
    {
      goto Step3
    }
  }
} return;
```

```
Step 3:

if(f is only of one term (t_1) and this term is a meta-variable )
{
  instantiate (t_1) with t'_1 ∧ ··· ∧ t'_n
  return done;
}

if(f' is only of one term (t'_1) and the term is a meta-variable )
{
  instantiate (t'_1) with t_1 ∧ ··· ∧ t_m
  return done;
}

Step 4:

if(first term (t_1) of f_1 is a meta-variable)
{
  i = t_2 ∧ ··· ∧ t_m
  i' = t'_1 ∧ ··· ∧ t'_{n-1}
  if(t'_n == t_1 && i!=i')
  {
    return "Not unifiable"
  }
  else
  {
    Instantiate t_1 with i'
    Instantiate t'_n with i
    return done;
  }
}
else
{
  i = t_1 ∧ ··· ∧ t_{m-1}
  i' = t'_2 ∧ ··· ∧ t'_n
  if(t_m == t'_1 && i!=i')
  {
    return "Not unifiable"
  }
  else
  {
    Instantiate t_m with i'
    Instantiate t'_1 with i
    return done;
  }
}
```

E Declaration of originality

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Developing a Web-Based Hoare Logic Proof Assistant

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Goldener	Flavio

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
23.03.2016	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.