Some Lessons Learned on Writing Predicate Logic Proofs in Isabelle/Isar

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at

October 30, 2014

Abstract

We describe our experience with the use of the proving assistant Isabelle and its proof development language Isar for formulating and proving formal mathematical statements. Our focus is on how to use classical predicate logic and well established proof principles for this purpose, bypassing Isabelle's meta-logic and related technical aspects as much as possible. By a small experiment on the proof of (part of a) verification condition for a program, we were able to identify a number of important patterns that arise in such proofs yielding to a workflow with which we feel personally comfortable; the resulting guidelines may serve as a starting point for a the application of Isabelle/Isar for the "average" mathematical user (i.e, a mathematical user who is not interested in Isabelle/Isar per se but just wants to use it as a tool for computer-supported formal theory development).

Contents

1 Introduction

Isabelle [3] is a very well known and widely used proof assistant; its proof development language Isar [8] allows to write formally checked proofs that are still readable by humans and indeed resemble to a certain extent manually developed proofs. However, as for most computer-supported theorem proving tools, the learning curve for mastering Isabelle/Isar is pretty steep.

This is the case although there exists an abundance of material on Isabelle/Isar, in particular the survey [6] on programming and proving in Isabelle/HOL, the book [7] that mostly serves as the "manual" for the tool, the reference manual [8] on Isabelle/Isar, papers [5, 4] on the practical use of the language, and on the automatic proof methods in the system [1]. However, this information is partially overlapping, partially reflects the state of the art at different times throughout the long history of the system (much information focuses on the application of Isabelle proof methods and tactics in proof development rather on the more recent Isar language), partially it is on different level of abstraction (much material focuses use of individual logical rules in Isabelle/HOL which is rarely needed in practical proof development any more); partially the intended target audience are people interested in Isabelle per se rather than average mathematical users that just want to get their job done.

The author of this paper experienced the situation that, despite the existence of a lot of learning material, he had a hard time to extract that information that really interested him: how to use Isabelle/Isar for the purpose of formulating and proving mathematical statements in classical predicate logic. Ultimately we started a small experimental learning session by formulating some proving problem that arose in the context of program verification [2]. We used this problem in order to investigate how to develop a proof with Isabelle/Isar in a way that corresponds to our manual proof development (rather than in that way that Isabelle/Isar would like us lead us). In this paper, we deliberately do not explain the content of this proving problem, since it does not contribute to the purpose of our explication but just serves as the "raw material" from which our observations were extracted. Of course, the results may be shaped a bit by this material, but actually we don't think too much.

This paper documents the (preliminary) state of our investigations. It is subsequently structured as follows: we start by setting out in Section 2 our premises on how we would like to use Isabelle/Isar. In Section 3, we describe the patterns we have found useful for the "top-down" decomposition of proof goals to subgoals, i.e., those steps that are usually performed first in a proof. In Section 4, we describe the patterns we have found useful for the "bottom-up" derivation of new knowledge from existing knowledge, i.e, those steps by which we ultimately discharge the various subgoals. In Section 5, we give our overall conclusions. Appendix A lists the Isabelle/HOL theory that we have elaborated in the context of this experiment in pretty-printed form; Appendix B lists the corresponding plain text.

Our experiments were performed with the Isabelle 2014 distribution using the recommended jEdit interface. Undoubtedly expert Isabelle users may be able to outline better practices than those outlined in this paper or correct some factual errors of our presentation; if so, we would be very much interested to hear about these.

2 Premises

The premises of our considerations are as follows:

1. We want to state theorems by formulas in predicate logic, not in the Isabelle meta-logic. For instance, we want to state and prove a predicate-logic theorem

   theorem *VC1* :
   "$\forall (t :: address)(s :: store).\, pre\ t\ s \wedge null(left(s\ t)) \longrightarrow post\ (right(s\ t))\ s\ t\ s$"

   without getting too much involved with the details (and even the very existence) of the Isabelle meta-logic. While for the goals and purposes of Isabelle, the existence of an object-logic independent meta-logic is important, the differentiation between two logical levels is irritating for the average mathematical user.

   In particular, we do *not* want to state and prove the corresponding meta-logic theorem

   theorem *VC1* :
   fixes t :: "*address*" **and** s :: "*store*"
   assumes "$pre\ t\ s$" **and** "$null(left(s\ t))$"
   shows "$post\ (right(s\ t))\ s\ t\ s$"

   even if this seems to be the recommendation of Isabelle experts (and most theorems in Isabelle-related documentation are formulated in that style).

   This kind of meta-logic statements may appear also in the syntactic form

   theorem *VC1* :
   "$\bigwedge(t :: address)\ (s :: store).\ [\![pre\ t\ s;\ null(left(s\ t))]\!] \Longrightarrow null(left(s\ t))$"

   which may be abbreviated to

   theorem *VC1* :
   "$\bigwedge(t :: address)\ (s :: store).\ pre\ t\ s \implies null(left(s\ t)) \implies null(left(s\ t))$"

   or even to

   theorem *VC1* :
   "$pre\ (t :: address)(s :: store) \implies null(left(s\ t)) \Longrightarrow null(left(s\ t))$"

   Still we are not interested in formulating and proving such statements which actually express rules of the Isabelle meta-logic, not theorems in the object-level predicate logic. We don't want to be bothered by having to explain to readers and users of our theorems the difference between the object-level logical connective $\longrightarrow$ and the meta-logic arrow $\Longrightarrow$. We just want to state and prove predicate logic statements. We are interested in Isabelle because (and as far as) it is a mean to achieve this goal; the use of Isabelle is for us not a mean in itself.

2. We would like to use the Isabelle/Isar proof language to develop proofs in the usual "top-down" style that is common in mathematical practice by repeatedly reducing goals to smaller subgoals ("backward proofs"); only if no further reduction is possible, "bottom-up" steps are applied to infer new knowledge from existing knowledge until the current subgoal can be proved ("forward proofs").

However, the typically presented Isar proof skeleton

> **theorem** *formula*
> **proof**
>   **assume** $\mathit{assumption}_1$ **and** ... **and** $\mathit{assumption}_m$
>   **from** ... **have** $\mathit{formula}_1$ **by** ...
>   **from** ... **have** $\mathit{formula}_2$ **by** ...
>   ...
>   **from** ... **have** $\mathit{formula}_n$ **by** ...
>   **from** ... **show** *goal* **by** ...
> **qed**

only presents one top-down step: the proof of *formula* is by an (typically implicitly selected) rule reduced to the proof of the meta-logic statement

$$\mathit{assumption}_1 \Longrightarrow \ldots \Longrightarrow \mathit{assumption}_m \Rightarrow \mathit{goal}$$

Subsequently, from the assumptions by bottom-up steps gradually additional knowledge $\mathit{formula}_1, \ldots, \mathit{formula}_n$ is derived from which ultimately *goal* can be shown. However, this presentation has some problems:

- The rule implicitly selected by Isabelle/Isar is a logical introduction rule that gets rid only of the outermost logical symbol (quantifier or connective); the further reduction of the goal is not shown.
- The goal formula is only mentioned in the last line of the derivation; we don't see from the beginning to which goal we (by the application of the rule) are actually heading.

Proof scripts in this style thus don't really convey much of the intuition that guided the development of the proof.

Our goal is to determine how Isabelle/Isar can be used in top-down proof development such that the generated proofs are actually presented also in the style in which they were originally developed.

3. We want to elaborate how with the *minimum* amount of knowledge about inference rules and proof methods available in Isabelle/Isar, the main types of logical inference used in mathematical practice can be performed. Apart from the top-down goal decomposition steps mentioned above, these compromise in particular
   - the usage of universally quantified knowledge by explicit instantiation,
   - the usage of existentially quantified knowledge for introducing new constants,
   - the proof of existentially quantified goals by explicit instantiation,
   - the replacement of defined functions and predicates in knowledge and goals.
4. In Isabelle/Isar proof scripts frequently formulas have to be stated that are instances of other formulas (definitions or knowledge) or parts of other formulas (goals). We would like to elaborate how to get hold of these formulas without manual derivation.

3 Decomposing Goals

In this section, we show how by the proof methods `auto` and `unfold` compound goals are decomposed to simpler goals and in atomic goals the definitions of predicates can be expanded. We can thus perform the usual first "top-down" steps in a proof.

3.1 Decomposition of Compound Goals

When starting the proof of a theorem such as

```
theorem VC1 :
 "∀(t::address)(s::store).
    pre t s ∧ null (left(s t)) ⟶ post (right(s t)) s t s"
```

whose formula still contains universal quantifiers (∀) and logical connectives (∧, ⟶), we do not use the automatically selected decomposition rule which just gets rid of a single logical symbol (in above case, the $\forall t$, leaving a formula of the form $\forall s. \ldots$). Rather we explicitly select the `auto` rule to apply all the usual decomposition steps at once:

```
theorem VC1 :
 "∀(t::address)(s::store).
    pre t s ∧ null (left(s t)) ⟶ post (right(s t)) s t s"
proof (auto)
  ...
qed
```

The "Output" window of the Isabelle/jEdit interface shows the derived goal

```
goal (1 subgoal):
 1. ⋀t s. pre t s ⟹ null (left (s t)) ⟹ post (right (s t)) s t s
```

By a sequence of `fix`, `assume`, and `show` steps

```
proof (auto)
  fix t s
  assume "pre t s"
  assume "null (left(s t))"
  show "post (right(s t)) s t s"
  ...
qed
```

we display the usual first steps of such a proof and also the ultimate goal to be shown (the corresponding subformulas can be copied and pasted from the "Output" window). The resulting proof state is

```
goal (1 subgoal):
 1. post (right (s t)) s t s
```

3.2 Unfolding Definitions in Goals

When proving a goal

```
show "post (right(s t)) s t s"
```

with a defined predicate such as

```
definition post :: "address ⇒ store ⇒ address ⇒ store ⇒ bool" where
 "post t s oldt olds =
    ((stree t s) ∧
     (∀a::address.¬(nodein a oldt olds) ⟶ s a = olds a) ∧
     (∀k::key.(keyin k t s) ⟶ keyin k oldt olds) ∧
     (∀k::key.(keyin k oldt olds) ⟶
                (keyin k t s ⟷ ¬minkeyin k oldt olds)))"
```

we apply the `unfold` rule to expand the definition. If the goal is a subgoal of another proof, we start a corresponding subproof:

```
show "post (right(s t)) s t s"
proof (unfold post_def)
  ...
qed
```

In the “Output” window, then the correspondingly expanded goal is displayed:

```
goal (1 subgoal):
 1. stree (right (s t)) s ∧
    (∀a. ¬ nodein a t s ⟶ s a = s a) ∧
    (∀k. keyin k (right (s t)) s ⟶ keyin k t s) ∧
    (∀k. keyin k t s ⟶ keyin k (right (s t)) s = (¬ minkeyin k t s))
```

We may copy and paste this goal into a corresponding `show` command. Since the goal is a compound formula, we may (as described in the previous section) again apply the `auto` method for automatic goal decomposition:

```
show "post (right(s t)) s t s"
proof (unfold post_def)
  show
   "stree (right (s t)) s ∧
    (∀a. ¬ nodein a t s ⟶ s a = s a) ∧
    (∀k. keyin k (right (s t)) s ⟶ keyin k t s) ∧
    (∀k. keyin k t s ⟶ keyin k (right (s t)) s = (¬ minkeyin k t s))"
  proof (auto)
    ...
  qed
qed
```

3.3 Proving Multiple Goals

The decomposition of a conjunctive goal formula

```
show
 "stree (right (s t)) s ∧
  (∀a. ¬ nodein a t s ⟶ s a = s a) ∧
  (∀k. keyin k (right (s t)) s ⟶ keyin k t s) ∧
  (∀k. keyin k t s ⟶ keyin k (right (s t)) s = (¬ minkeyin k t s))"
proof (auto)
   ...
qed
```

results in multiple goals:

```
goal (4 subgoals):
 1. stree (right (s t)) s
 2. ⋀k. keyin k (right (s t)) s ⟹ keyin k t s
 3. ⋀k. keyin k t s ⟹ keyin k (right (s t)) s ⟹ minkeyin k t s ⟹ False
 4. ⋀k. keyin k t s ⟹ ¬ minkeyin k t s ⟹ keyin k (right (s t)) s
```

Please note that by the decomposition, the second formula in the conjunction

```
(∀a. ¬ nodein a t s ⟶ s a = s a)
```

was automatically proved; on the other hand, the proof of the formula

```
(∀k. keyin k t s ⟶ keyin k (right (s t)) s = (¬ minkeyin k t s))
```

was split into the two subgoals 3 and 4.

The resulting four subgoals are proved by multiple proofs separated by the `next` command (the individual subformulas can be copied and pasted from the “Output” window):

```
show
 "stree (right (s t)) s ∧
  (∀a. ¬ nodein a t s ⟶ s a = s a) ∧
  (∀k. keyin k (right (s t)) s ⟶ keyin k t s) ∧
  (∀k. keyin k t s ⟶ keyin k (right (s t)) s = (¬ minkeyin k t s))"
proof (auto)
  show "stree (right (s t)) s"
  ...
next
  fix k
  assume "keyin k (right (s t)) s"
  show "keyin k t s"
  ...
next
  fix k
  assume "keyin k t s"
  assume "keyin k (right (s t)) s"
  assume "minkeyin k t s"
  show False
```

```
    ...
  next
    fix k
    assume "keyin k t s"
    assume "¬ minkeyin k t s"
    show "keyin k (right (s t)) s"
    ...
  qed
```

Each proof thus follows the previously outlined decomposition strategies.

3.4 Proving Existential Goals

When proving an existential goal such as

```
show "∃p n. nodepath p n t s ∧ p n = a"
```

we start the proof with `proof -` which does not decompose the goal any further but starts a "bottom-up" proof (see Section 4):

```
show "∃p n. nodepath p n t s ∧ p n = a"
proof -
...
qed
```

The proof requires the derivation of an instance of the formula where the existentially quantified variables are replaced by concrete witness terms. In order to avoid writing the formula instance we may introduce auxiliary schematic variables for these terms such that we can copy and paste the formula from the output window and just replace the quantified variables by the schematic variables:

```
show "∃p n. nodepath p n t s ∧ p n = a"
proof -
  let ?p = "cons t p"
  let ?n = "n+1"
  have "nodepath ?p ?n t s ∧ ?p ?n = a"
  ...
  from this show ?thesis by blast
qed
```

The part "…" denotes the proof of the formula instance, e.g. in above case by selecting the automatic goal decomposition

```
show "∃p n. nodepath p n t s ∧ p n = a"
proof -
  let ?p = "cons t p"
  let ?n = "n+1"
  have "nodepath ?p ?n t s ∧ ?p ?n = a"
  proof (auto)
    ...
```

```
    end
    from this show ?thesis by blast
  qed
```

Please note the last line where the procedure `blast` is invoked to prove that the derived formula is an instance of the existential goal formula (denoted by the predefined name `?thesis`). The procedure `auto` does here not suffice; in certain situations, even a stronger proof procedure might be required (see the following section).

In the case of a formula with a single existentially quantified variable one might also invoke the implicitly selected decomposition rule which replaces the existentially quantified variable by a schematic variable such that only the value of this variable and the truth of the resulting instance has to be established. For instance

```
show "∃a. nodein a t s ∧ key (s a) = k"
proof
  ...
qed
```

leads to the proof goal

```
goal (1 subgoal):
 1. nodein ?a t s ∧ key (s ?a) = k
```

such that we might continue the proof with

```
show "∃a. nodein a t s ∧ key (s a) = k"
proof
  let ?a = "..."
  show "nodein ?a t s ∧ key (s ?a) = k"
  proof (auto)
    ...
  qed
qed
```

However, then the witness term for `?a` must not depend on any value obtained during the proof. On the contrary, in the format described above we may for instance write

```
show "∃a. nodein a t s ∧ key (s a) = k"
proof -
  ...
  from ... obtain a where ...
  ...
  have "nodein a t s ∧ key (s a) = k"
  proof (auto)
    ...
  qed
  from this show ?thesis by auto
qed
```

which is not allowed in the more special format. In order to avoid to keep in mind a second proof format that only can be applied in certain situations, we don't use it any further.

3.5 Proving by Case Distinction

We may split a proof of a statement like

```
show "child (case i of 0 ⇒ t | Suc x ⇒ p x)
            (case i + 1 of 0 ⇒ t | Suc x ⇒ p x) s"
```

by the usual technique of "case distinction" as follows

```
show "child (case i of 0 ⇒ t | Suc x ⇒ p x)
            (case i + 1 of 0 ⇒ t | Suc x ⇒ p x) s"
proof (cases)
  assume cond: "i=0"
  ...
  from ... show ?thesis ...
next
  assume cond: "i≠0"
  ...
  from ... show ?thesis ...
qed
```

Since the goal is not changed by the application of case distinction, we don't mention it explicitly at the beginning of each subproof, unless some more decomposition step is to be applied. In that case, we may write the proof also in the format

```
show "child (case i of 0 ⇒ t | Suc x ⇒ p x)
            (case i + 1 of 0 ⇒ t | Suc x ⇒ p x) s"
proof (cases)
  assume cond: "i=0"
  show ?thesis
  proof ...
    ...
  qed
next
  assume cond: "i≠0"
  show ?thesis
  proof ...
    ...
  qed
qed
```

4 Deriving Knowledge

In this section, we describe how the usual "bottom-up" steps of expanding definitions and applying quantified knowledge can be performed to derive new knowledge; we also describe the application of the automatic proof methods provided by Isabelle to close the proofs.

4.1 Expanding Definitions

The need to expand function definitions did not arise in our example proofs, since the `auto` command expanded these automatically.

As for predicate definitions, in a situation where we know some atomic formula, e.g.

```
assume pre: "pre t s"
```

the definition of the corresponding predicate

```
definition pre :: "address ⇒ store ⇒ bool" where
 "pre t x = (stree t x ∧ ¬null t)"
```

may be expanded by applying the command

```
from pre pre_def[of "t" "s"] have
  "stree t x ∧ ¬null t" by auto
```

where the optional `of` clause may be used to give the actual arguments for the formal parameters of the predicate. Apart from simplifying the task of the proving procedure, the explicit instantiation has the advantage that by placing in the Isabelle/jEdit interface the cursor before the token `have` the knowledge

```
picking this:
    pre t s
    pre t s = (stree t s ∧ ¬ null t)
```

is displayed from which the instantiated definition can just be copied and pasted into the proof script. The resulting knowledge may be also immediately decomposed such as in

```
from pre pre_def[of "t" "s"] have
  pre1: "stree t s" and
  pre2: "¬ null t" by auto
```

Sometimes the procedure `auto` is too weak to infer the correctness of the claimed instantiation; in such situations, the more powerful procedure `metis` may be used instead. However, `metis` seems to have problems of deriving multiple goals simultaneously; we thus have to write

```
from pre3 tree_def[of "t" "s"] have
  pre5: "(∀p. ¬ ipath p t s)" by metis
from pre3 tree_def[of "t" "s"] have
  pre6: "(∀p1 p2 n1 n2. nodepath p1 n1 t s ∧
           nodepath p2 n2 t s∧ p1 n1 = p2 n2 ⟶
             n1 = n2 ∧ (∀i<n1. p1 i = p2 i))" by metis
```

to derive the two parts of the conjunctive formula that results from the instantiation of definition

```
definition tree :: "address ⇒ store ⇒ bool" where
 "tree t s =
    ((∀p::path. ¬ipath p t s) ∧
     (∀(p1::path) (p2::path) (n1::nat) (n2::nat).
       nodepath p1 n1 t s ∧ nodepath p2 n2 t s ∧
       p1 n1 = p2 n2 ⟶
         n1 = n2 ∧ (∀i::nat. i < n1 ⟶ p1 i = p2 i)))"
```

4.2 Instantiating Universal Knowledge

Given a universally quantified knowledge formula, we may obtain an instantiation of this formula only by constructing the instantiation manually. However, to reduce typing overhead schematic variables for the instantiation terms may be used. For instance, the commands

```
let ?q1 ="cons t p1" and ?q2 ="cons t p2" and ?m1 ="n1+1" and ?m2 ="n2+1"
from pre6 have
  "nodepath ?q1 ?m1 t s ∧ nodepath ?q2 ?m2 t s ∧ ?q1 ?m1 = ?q2 ?m2 ⟶
          ?m1 = ?m2 ∧ (∀i<?m1. ?q1 i = ?q2 i)" by blast
```

create an instance of the previously derived formula

```
from ... have
  pre6: "(∀p1 p2 n1 n2. nodepath p1 n1 t s ∧ nodepath p2 n2 t s ∧
          p1 n1 = p2 n2 ⟶
            n1 = n2 ∧ (∀i<n1. p1 i = p2 i))" by metis
```

By placing the cursor on the line `from pre6 have` we get in the output window a copy of the formula

```
picking this:
  ∀p1 p2 n1 n2. nodepath p1 n1 t s ∧ nodepath p2 n2 t s ∧
     p1 n1 = p2 n2 ⟶ n1 = n2 ∧ (∀i<n1. p1 i = p2 i)
```

whose body may be copied and pasted into the `have` clause and the variables be replaced by the schematic variables denoting the instantiation values.

4.3 Applying Existential Knowledge

Given an existentially quantified knowledge formula, we may obtain a constant corresponding to the quantified variable by the `obtain` command. For instance, we may write in a proof

```
from ... have
  "(∃a. nodein a (right (s t)) s ∧ key (s a) = k)" by ...
from this obtain a where "nodein a (right (s t)) s /\ "key (s a) = k" by auto
```

The knowledge derived about the formula may be also immediately decomposed respectively specialized such as in

```
from ... have
  "(∃a. nodein a (right (s t)) s ∧ key (s a) = k)" by ...
from this obtain a where
  a3: "nodein a (right (s t)) s" and
  a4: "key (s a) = k" by auto
```

By placing the cursor on the line `from this obtain a where` we e get in the output window a copy of the formula

```
picking this:
  ∃a. nodein a (right (s t)) s ∧ key (s a) = k
```

whose body may be copied and pasted into the `have` clause and the variables be replaced by the schematic variables denoting the instantiation values.

4.4 Closing Proofs

For closing proofs we may apply the following automatic procedures provided by Isabelle:

by auto This is a quick strategy that subsumes simplification (as provided by `simp`) with a small amount of proof search. It is the default strategy we use most of the time.

by blast If `auto` fails, we try `blast` which is a fast tableau prover directly written in ML.

sledgehammer/metis If also `blast` fails, we apply `sledgehammer` which uses external automated provers (notably E, SPASS, and Vampire) to discover a proof. When applied via the Isabelle/jEdit interface (buttons "Sledgehammer/Apply"), the command may take a minute or so: if a proof is discovered, a line of the form

```
by (metis rules)
```

is displayed where *rules* is a list of rules that were used in the proof.

The form of the result is that of a call of the internal resolution prover `metis` to which the list of rules is passed as an argument; by double-clicking on this line, the command is inserted into the proof script; the resulting call of `metis` is then typically able to quickly reconstruct the externally discovered proof and discharge of the goal.

To all these procedures the facts to be used are passed by the `from` clause of a proof command; `sledgehammer` additionally uses the definitions and lemmas in the current scope and is thus also able to detect missing facts that should be also listed (in addition to those facts that are needed from the standard Isabelle libraries).

5 Conclusions

By the small "self-exercise" sketched in this paper, we were able to determine a strategy by which we feel comfortable to elaborate proofs with the Isabelle proof assistant using the Isar proof development language. In particular, we were able to extract from the plenitude of sources available for Isabelle/Isar those parts that are most relevant for our purpose, the development of theorems and proofs in classical logic in a style that closely resembles our usually preferred practice. Undoubtedly there are still many further aspects that need to be elaborated for proving with special theories (we did not e.g. not discuss induction or special quantifiers). However, at least with the predicate logic aspect we now feel reasonably familiar. Further work with the tool will certainly bring additions/revisions to our personal preferences and recommendations; we will then update this paper correspondingly.

We hope that by the results presented in this document, others may experience a somewhat less steep path towards the use of Isabelle/Isar for common mathematical practice.

References

[1] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic Proof and Disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of*

Combining Systems (FroCoS 2011), volume 6989 of *LNCS*, pages 12–27. Springer, 2011. http://www21.in.tum.de/~nipkow/pubs/frocos11.pdf.

[2] Daniel Bruns, Wojciech Mostowski, and Mattias Ulbrich. Implementation-level Verification of Algorithms with KeY. *Software Tools for Technology Transfer*, 16, 2014. http://link.springer.com/article/10.1007%2Fs10009-013-0293-y.

[3] Isabelle, 2014. http://isabelle.in.tum.de/index.html.

[4] Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, pages 259–278. Springer, 2003. http://www21.in.tum.de/~nipkow/pubs/types02.pdf.

[5] Tobias Nipkow. A Tutorial Introduction to Structured Isar Proofs, 2008. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.220.8443.

[6] Tobias Nipkow. Programming and Proving in Isabelle/HOL, August 2014. http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2014/doc/prog-prove.pdf.

[7] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL — A Proof Assistant for Higher-Order Logic, August 2014. http://isabelle.in.tum.de/doc/tutorial.pdf.

[8] Makarius Wenzel. The Isabelle/Isar Reference Manual, August 2014. http://isabelle.in.tum.de/doc/isar-ref.pdf.

A Isabelle Theory (Formatted)

theory *Trees*
imports *Main*
begin

— ——————————————————————————————————————
— Trees.thy: verification of an imperative tree algorithm
—
— Given: a pointer t to the root of a non-empty binary search tree
— (not necessarily balanced).
— Verify that the procedure (deleteMin) removes the node with the
— minimal key from the tree.
—
— See Section 5 of Daniel Bruns et al:
— Implementation-level verification of algorithms with KeY
— Int J Softw Tools Technol Transfer, Springer, 17 November 2013
— DOI 10.1007/s10009-013-0293-y, http://dx.doi.org/10.1007/s10009-013-0293-y
—
— (c) 2014, Wolfgang Schreiner (Wolfgang.Schreiner@risc.jku.at)
— Research Institute for Symbolic Computation (RISC)
— Johannes Kepler University, Linz, Austria
— ——————————————————————————————————————

— keys
type_synonym *key ="int"*

— addresses
type_synonym *address ="nat"*
definition *null :: "address => bool"* **where**
"null x = (x = 0)"

— nodes: key, left and right pointer
type_synonym *node ="key * address * address"*
fun *key :: "node ⇒ key"* **where**
"key (x,_,_) = x"
fun *left :: "node => address"* **where**
"left (_,x,_) = x"
fun *right :: "node => address"* **where**
"right (_,_,x) = x"

— generic sequences
type_synonym *'a seq ="nat => 'a"*

— stores
type_synonym *store ="node seq"*

— paths
type_synonym *path = "address seq"*
fun *head :: "path ⇒ address"* **where**
"head p = p(0)"
fun *tail :: "path ⇒ path"* **where**
"tail p = (λn::nat. p (n+1))"
fun *cons :: "address ⇒ path ⇒ path"* **where**

```
 "cons a p = (λn::nat. case n of 0 ⇒ a | Suc(n0) ⇒ p n0)"
fun put :: "path ⇒ nat ⇒ address ⇒ path" where
 "put p k a = (λn::nat. if n = k then a else p k)"

— a2 is child of a1
definition child :: "address ⇒ address ⇒ store ⇒ bool" where
 "child a1 a2 s = (a2 = left(s a1) ∨ a2 = right(s a1))"

— path element p(i+1) is child of p(i)
definition children :: "path ⇒ nat ⇒ store ⇒ bool" where
 "children p i s = child (p i) (p (i+1)) s"

— p is an infinite path starting from t
definition ipath :: "path ⇒ address ⇒ store ⇒ bool" where
 "ipath p t s =
     (p(0) = t ∧
       (∀i::nat. ¬null(p i)) ∧
       (∀i::nat. children p i s))"

— p is a path of length n starting from t
definition path :: "path ⇒ nat ⇒ address ⇒ store ⇒ bool" where
 "path p n t s =
     (p(0) = t ∧
       (∀i::nat. i < n ⟶ ¬null(p i)) ∧
       (∀i::nat. i < n ⟶ children p i s))"

— p is a path of length n starting from t leading to a node
definition nodepath :: "path ⇒ nat ⇒ address ⇒ store ⇒ bool" where
 "nodepath p n t s = ((path p n t s) ∧ ¬null(p n))"

— t is a tree
definition tree :: "address ⇒ store ⇒ bool" where
— no infinite paths respectively cycles and
— no sharing of nodes in multiple paths, also prevents cycles
 "tree t s =
     ((∀p::path. ¬ipath p t s) ∧
      (∀(p1::path) (p2::path) (n1::nat) (n2::nat).
        nodepath p1 n1 t s ∧ nodepath p2 n2 t s ∧
        p1 n1 = p2 n2 ⟶
          n1 = n2 ∧ (∀i::nat. i < n1 ⟶ p1 i = p2 i)))"

— t is a search tree with unique keys
definition stree :: "address ⇒ store ⇒ bool" where
 "stree t s =
     ((tree t s) ∧
      (∀(p::path)(n::nat).
         nodepath p (n+1) t s ⟶
           (let a1 = (p n) in let a2 = (p (n+1)) in
            let n1 = s(a1) in let n2 = s(a2) in
           (a2 = left(n1) ⟷ key(n2) < key(n1)))))"

— a is the address of a node in tree t
definition nodein :: "address ⇒ address ⇒ store ⇒ bool" where
 "nodein a t s = (∃(p::path)(n::nat). (nodepath p n t s) ∧ p n = a)"
```

— k is a key in tree t
definition *keyin :: "key ⇒ address ⇒ store ⇒ bool"* **where**
"keyin k t s = (∃(a::address). nodein a t s ∧ key (s a) = k)"

— k is the minimum key in tree t
definition *minkeyin :: "key ⇒ address ⇒ store ⇒ bool"* **where**
"minkeyin k t s = (keyin k t s ∧ ¬(∃k0::key. k0 < k ∧ keyin k0 t s))"

— precondition of deleteMin
definition *pre :: "address ⇒ store ⇒ bool"* **where**
— input is a non-null search tree
"pre t x = (stree t x ∧ ¬null t)"

— postcondition of deleteMin
definition *post :: "address ⇒ store ⇒ address ⇒ store ⇒ bool"* **where**
— the output is a search tree
— only the nodes in the input tree may have been changed
— the only keys in the output tree are those from the input tree
— a key from the input tree is in the output tree iff it is not the minimum key

```
"post t s oldt olds =
   ((stree t s) ∧
    (∀a::address.¬(nodein a oldt olds) ⟶ s a = olds a) ∧
    (∀k::key.(keyin k t s) ⟶ keyin k oldt olds) ∧
    (∀k::key.(keyin k oldt olds) ⟶
               (keyin k t s ⟷ ¬minkeyin k oldt olds)))"
```

— ——————————————————————————-
— Predecessor function and corresponding lemmas
— ——————————————————————————-

fun *Pre :: "nat ⇒ nat"* **where**
"Pre i = (THE x::nat.(Suc x=i))"

lemma *haspre :*
"∀i::nat. i ≠ 0 ⟶ (∃x::nat. Suc x=i)"
proof *(auto)*
fix *i::nat*
assume *"0 < i"*
show *"∃x. Suc x = i"* **by** *(metis ‘0 < i‘ gr0_conv_Suc)*
qed

lemma *ispre :*
"∀i::nat. i ≠ 0 ⟶ Suc(Pre i) = i"
proof *(auto)*
fix *i::nat*
assume *"0 < i"*
moreover from *this haspre* **have** *"(∃x. Suc x = i)"* **by** *auto*
ultimately show *"Suc (THE x. Suc x = i) = i"* **by** *auto*
qed

— ——————————————————————————
— Lemma (childpath)
— ——————————————————————————

```
lemma childpath:
 "∀ (t::address)(s::store)(p::path)(n::nat)(t0::address).
     ¬null t ∧ child t t0 s ∧ nodepath p n t0 s  ⟶ nodepath (cons t p) (n+1) t s"
proof (auto)
  fix t s p n t0
  assume pre2: "¬ null t"
  assume pre3: "child t t0 s"
  assume pre4: "nodepath p n t0 s"
  show "nodepath (case_nat t p) (Suc n) t s"
  proof (unfold "nodepath_def")
    show "path (case_nat t p) (Suc n) t s ∧
          ¬ null (case Suc n of 0 ⇒ t | Suc x ⇒ p x)"
    proof (auto)
      show "path (case_nat t p) (Suc n) t s"
      proof (unfold path_def)
        show "(case 0 of 0 ⇒ t | Suc x ⇒ p x) = t ∧
              (∀ i<Suc n. ¬ null (case i of 0 ⇒ t | Suc x ⇒ p x)) ∧
              (∀ i<Suc n. children (case_nat t p) i s)"
        proof (auto)
          fix i
          assume "i < Suc n"
          assume "null (case i of 0 ⇒ t | Suc x ⇒ p x)"
          from this pre2 pre4 nodepath_def show "False"
            by (metis ‘i < Suc n‘ less_Suc_eq_0_disj old.nat.simps(4) old.nat.simps(5) path_def)
        next
          fix i
          assume "i < Suc n"
          show "children (case_nat t p) i s"
          proof (unfold "children_def")
            from pre3 pre4 nodepath_def[of "p" "n" "t0" "s"] have
            4: "path p n t0 s" and
            5: "¬ null (p n)" by auto
            show "child (case i of 0 ⇒ t | Suc x ⇒ p x)
                        (case i + 1 of 0 ⇒ t | Suc x ⇒ p x) s"
            proof (cases)
              assume cond: "i=0"
              from pre3 cond 4 path_def children_def child_def have
                "child t (p 0) s" by auto
              from this cond show ?thesis by auto
            next
              assume cond: "i≠0"
              from pre3 cond 4 path_def children_def child_def ispre have
               "child (p (Pre i)) (p i) s"
              by (metis Suc_eq_plus1 ‘i < Suc n‘ less_Suc_eq_0_disj old.nat.inject)
              from this cond show ?thesis by (metis Suc_eq_plus1 ispre old.nat.simps(5))
            qed
          qed
        qed
      qed
    next
      from pre4 nodepath_def show "null (p n) ⟹ False" by metis
    qed
  qed
qed
```

— ————————————————————————————

— Lemma (treeright)

— ————————————————————————————

```
lemma treeright :
 "∀(t::address)(s::store). tree t s ∧ ¬null t  ⟶ tree (right(s t)) s"
proof (auto)
  fix t s
  assume pre1: "tree t s"
  assume pre2: "¬null t"
  show "tree (right(s t)) s"
  proof (unfold tree_def)

    from pre1 tree_def[of "t" "s"] have
      pre5: "(∀p. ¬ ipath p t s)" by metis
    from pre1 tree_def[of "t" "s"] have
      pre6: "(∀p1 p2 n1 n2. nodepath p1 n1 t s ∧ nodepath p2 n2 t s ∧
                p1 n1 = p2 n2 ⟶
                  n1 = n2 ∧ (∀i<n1. p1 i = p2 i))" by metis

    show "(∀p. ¬ ipath p (right (s t)) s) ∧
          (∀p1 p2 n1 n2.
             nodepath p1 n1 (right (s t)) s ∧ nodepath p2 n2 (right (s t)) s ∧
               p1 n1 = p2 n2 ⟶
                 n1 = n2 ∧ (∀i<n1. p1 i = p2 i))"

    proof (auto)
      fix p
      assume a1: "ipath p (right (s t)) s"
      show False
      proof -
        from a1 ipath_def[of "p" "right (s t)" "s"] have
          n1: "(p 0 = right (s t) ∧ (∀i. ¬ null (p i)) ∧
                (∀i. children p i s))" by auto
        let ?p0 ="cons t p"
        have "ipath ?p0 t s"
        proof (unfold ipath_def)
          show "cons t p 0 = t ∧ (∀i. ¬ null (cons t p i)) ∧
                (∀i. children (cons t p) i s)"
          proof (auto)
            fix i
            assume "null (case i of 0 ⇒ t | Suc x ⇒ p x)"
            from this pre2 n1 show False by (metis Nitpick.case_nat_unfold)
          next
            fix i
            show "children (case_nat t p) i s"
            proof (unfold children_def)
              from n1 children_def child_def show
              "child (case i of 0 ⇒ t | Suc x ⇒ p x)
                     (case i + 1 of 0 ⇒ t | Suc x ⇒ p x) s"
              by (metis Suc_eq_plus1 lessI less_Suc_eq_0_disj old.nat.simps(4) old.nat.simps(5))
            qed
          qed
        qed
```

```
    from this pre5 show False by auto
  qed
next
  fix p1 p2 n1 n2
  assume 1: "nodepath p1 n1 (right (s t)) s"
  assume 2: "nodepath p2 n2 (right (s t)) s"
  assume 3: "p1 n1 = p2 n2"
  show "n1 = n2"
  proof -
    let ?q1 ="cons t p1" and ?q2 ="cons t p2" and ?m1 ="n1+1" and ?m2 ="n2+1"
    from pre6 have
     "nodepath ?q1 ?m1 t s ∧ nodepath ?q2 ?m2 t s ∧ ?q1 ?m1 = ?q2 ?m2 ⟶
         ?m1 = ?m2 ∧ (∀i<?m1. ?q1 i = ?q2 i)" by blast
    from this show "n1 = n2"
    proof (auto)
      assume "n1 ≠ n2"
      from pre2 1 childpath child_def show "nodepath (case_nat t p1) (Suc n1) t s"
      by (metis One_nat_def add_Suc_right cons.simps monoid_add_class.add.right_neutral)

    next
      assume "n1 ≠ n2"
      from pre2 2 childpath child_def show "nodepath (case_nat t p2) (Suc n2) t s"
      by (metis One_nat_def add_Suc_right cons.simps monoid_add_class.add.right_neutral)

    next
      assume "n1 ≠ n2"
      from 3 show "p1 n1 = p2 n2" by auto
    qed
  qed
next
  fix p1 p2 n1 n2 i
  assume 1: "nodepath p1 n1 (right (s t)) s"
  assume 2: "nodepath p2 n2 (right (s t)) s"
  assume 3: "p1 n1 = p2 n2"
  assume 4: "i < n1"
  show "p1 i = p2 i"
  proof -
    let ?q1 ="(cons t p1)" and ?q2 ="(cons t p2)"
    let ?m1 ="n1+1" and ?m2 ="n2+1"
    from pre6 have
     "nodepath ?q1 ?m1 t s ∧ nodepath ?q2 ?m2 t s ∧ ?q1 ?m1 = ?q2 ?m2 ⟶
         ?m1 = ?m2 ∧ (∀i<?m1. ?q1 i = ?q2 i)" by blast
    from this show ?thesis
    proof (auto)
      assume "p1 i ≠ p2 i"
      from 1 pre2 childpath child_def show "nodepath (case_nat t p1) (Suc n1) t s"
      by (metis One_nat_def add_Suc_right cons.simps
                monoid_add_class.add.right_neutral pre2)
    next
      assume "p1 i ≠ p2 i"
      from 2 pre2 childpath child_def show "nodepath (case_nat t p2) (Suc n2) t s"
      by (metis One_nat_def add_Suc_right cons.simps
                monoid_add_class.add.right_neutral pre2)
    next
```

```
          assume "p1 i ≠ p2 i"
          from 3 show "p1 n1 = p2 n2" by auto
        next
          assume 5: "∀i<Suc n2. (case i of 0 ⇒ t | Suc x ⇒ p1 x) =
                                (case i of 0 ⇒ t | Suc x ⇒ p2 x)"
          assume 6: "n1 = n2"
          from 5 6 show "p1 i = p2 i" by (metis "4"Suc_less_eq old.nat.simps(5))
        qed
      qed
    qed
  qed
qed
```

— ———————————————————————————————

— Lemma (keyinright)

— ————————————————————————————————-

```
lemma keyinright:
 "∀(t::address)(s::store)(k::key). ¬null t ∧ keyin k (right (s t)) s ⟶ keyin k t s"
proof (auto)
  fix t s k
  assume pre2: "¬null t"
  assume a1: "(keyin k (right(s t)) s)"
  show "keyin k t s"
  proof (unfold keyin_def)
    show "∃a. nodein a t s ∧ key (s a) = k"
    proof -
      from a1 keyin_def[of "k" "(right(s t))" "s"] have
        a2: "(∃a. nodein a (right (s t)) s ∧ key (s a) = k)" by auto
      from a2 obtain a where
        a3: "nodein a (right (s t)) s" and
        a4: "key (s a) = k" by auto
      from a3 nodein_def[of "a" "right (s t)" "s"] obtain "p" "n" where
        a5: "nodepath p n (right (s t)) s" and
        a6: "p n = a" by auto
      have "nodein a t s ∧ key (s a) = k"
      proof (auto)
        show "nodein a t s"
        proof (unfold nodein_def)
          show "∃p n. nodepath p n t s ∧ p n = a"
          proof -
            let ?p = "cons t p"
            let ?n = "n+1"
            have "nodepath ?p ?n t s ∧ ?p ?n = a"
            proof (auto)
              from childpath nodepath_def children_def child_def pre2 a5 a6 show
              "nodepath (case_nat t p) (Suc n) t s" by (metis Suc_eq_plus1 cons.simps)
            next
              from cons_def a6 show "p n = a" by auto
            qed
            from this show ?thesis by blast
          qed
        qed
      next
        from a4 show "key (s a) = k" by auto
```

```
      qed
      from this show ?thesis by auto
    qed
  qed
qed
```

```
  static Tree deleteMin(Tree t) {
      Tree p = t.left;
      if (p == null)
        t = t.right;
      else {
        Tree p2 = t;
        Tree tt = p.left;
        while (tt != null) {
          p2 = p; p = tt; tt = p.left;
        }
        p2.left = p.right;
      }
      return t;
  }
```

— ———————————————————————————————
— Verification Condition 1
— correctness of first program path in which store is not changed
— assume pre; p = t.left; assume p == null; t = t.right; assert post;
— ———————————————————————————————

```
theorem VC1 :
 "∀ (t::address)(s::store). pre t s ∧ null (left(s t)) ⟶ post (right(s t)) s t s"

proof (auto)
  fix t s
  assume pre: "pre t s"
  assume "null (left(s t))"
  show "post (right(s t)) s t s"

  proof (unfold post_def)

    from pre pre_def[of "t" "s"] have
      pre1: "stree t s" and
      pre2: "¬ null t" by auto
    from pre1 stree_def[of "t" "s"] have
      pre3: "tree t s" and
      pre4: "∀ p n. nodepath p (n + 1) t s ⟶
                (let a1 = p n; a2 = p (n + 1); n1 = s a1; n2 = s a2 in
                (a2 = left n1) = (key n2 < key n1))" by auto
    from pre3 tree_def[of "t" "s"] have
      pre5: "(∀ p. ¬ ipath p t s)" by metis
    from pre3 tree_def[of "t" "s"] have
      pre6: "(∀ p1 p2 n1 n2. nodepath p1 n1 t s ∧ nodepath p2 n2 t s∧ p1 n1 = p2 n2 ⟶
                n1 = n2 ∧ (∀ i<n1. p1 i = p2 i))" by metis
    show
    "stree (right (s t)) s ∧
```

```
      (∀a. ¬ nodein a t s ⟶ s a = s a) ∧
      (∀k. keyin k (right (s t)) s ⟶ keyin k t s) ∧
      (∀k. keyin k t s ⟶ keyin k (right (s t)) s = (¬ minkeyin k t s))"

    proof (auto)

      — subproof
      show "stree (right(s t)) s"
      proof (unfold stree_def)
        show
        "tree (right (s t)) s ∧
          (∀p n. nodepath p (n + 1) (right (s t)) s ⟶
          (let a1 = p n; a2 = p (n + 1); n1 = s a1; n2 = s a2 in
            (a2 = left n1) = (key n2 < key n1)))"
        proof (auto)
          from treeright pre2 pre3 show "tree (right (s t)) s" by auto
        next
          fix p n
          assume "nodepath p (Suc n) (right (s t)) s"
          from this pre4 nodepath_def childpath child_def pre2
          show "let a2 = p (Suc n); n1 = s (p n) in (a2 = left n1) = (key (s a2) < key n1)"
          by (metis Suc_eq_plus1 cons.simps old.nat.simps(5))
        qed
      qed

    — subproof
    next
      fix k
      assume "keyin k (right (s t)) s"
      from keyinright pre2 this show "keyin k t s" by auto

    — subproof
    next
      fix k
      assume "keyin k t s"
      assume "keyin k (right (s t)) s"
      assume "minkeyin k t s"
      show False sorry

    — subproof
    next
      fix k
      assume "keyin k t s"
      assume "¬ minkeyin k t s"
      show "keyin k (right (s t)) s" sorry
    qed
  qed
qed

end
```

B Isabelle Theory (Plain)

```
theory Trees
imports Main
begin

-- "------------------------------------------------------------------------------------------  "
-- "Trees.thy: verification of an imperative tree algorithm                                     "
-- "                                                                                            "
-- "Given: a pointer t to the root of a non-empty binary search tree"
-- "       (not necessarily balanced)."
-- "Verify that the procedure (deleteMin) removes the node with the"
-- "minimal key from the tree."
-- "                                                                                            "
-- "See Section 5 of Daniel Bruns et al: "
-- "Implementation-level verification of algorithms with KeY "
-- "Int J Softw Tools Technol Transfer, Springer, 17 November 2013"
-- "DOI 10.1007/s10009-013-0293-y, http://dx.doi.org/10.1007/s10009-013-0293-y
-- "                                                                                            "
-- "(c) 2014, Wolfgang Schreiner (Wolfgang.Schreiner@risc.jku.at)                               "
-- "Research Institute for Symbolic Computation (RISC)"
-- "Johannes Kepler University, Linz, Austria "
-- "------------------------------------------------------------------------------------------  "

-- "keys"
type_synonym key ="int"

-- "addresses"
type_synonym address ="nat"
definition null :: "address => bool" where
 "null x = (x = 0)"

-- "nodes: key, left and right pointer"
type_synonym node ="key * address * address"
fun key :: "node \<Rightarrow> key" where
 "key (x,_,_) = x"
fun left :: "node => address" where
 "left (_,x,_) = x"
fun right :: "node => address" where
 "right (_,_,x) = x"

-- "generic sequences"
type_synonym 'a seq ="nat => 'a"

-- "stores"
type_synonym store ="node seq"

-- "paths"
type_synonym path = "address seq"
fun head :: "path \<Rightarrow> address" where
 "head p = p(0)"
fun tail :: "path \<Rightarrow> path" where
 "tail p = (\<lambda>n::nat. p (n+1))"
fun cons :: "address \<Rightarrow> path \<Rightarrow> path" where
 "cons a p = (\<lambda>n::nat. case n of 0 \<Rightarrow> a | Suc(n0) \<Rightarrow> p n0)"
fun put :: "path \<Rightarrow> nat \<Rightarrow> address \<Rightarrow> path" where
 "put p k a = (\<lambda>n::nat. if n = k then a else p k)"

-- "a2 is child of a1"
definition child :: "address \<Rightarrow> address \<Rightarrow> store \<Rightarrow> bool" where
 "child a1 a2 s = (a2 = left(s a1) \<or> a2 = right(s a1))"
```

```
-- "path element p(i+1) is child of p(i)"
definition children :: "path \<Rightarrow> nat \<Rightarrow> store \<Rightarrow> bool" where
 "children p i s = child (p i) (p (i+1)) s"

-- "p is an infinite path starting from t"
definition ipath :: "path \<Rightarrow> address \<Rightarrow> store \<Rightarrow> bool" where
 "ipath p t s =
     (p(0) = t \<and>
       (\<forall>i::nat. \<not>null(p i)) \<and>
       (\<forall>i::nat. children p i s))"

-- "p is a path of length n starting from t"
definition path :: "path \<Rightarrow> nat \<Rightarrow> address \<Rightarrow> store \<Rightarrow> bool" where
 "path p n t s =
     (p(0) = t \<and>
       (\<forall>i::nat. i < n \<longrightarrow> \<not>null(p i)) \<and>
       (\<forall>i::nat. i < n \<longrightarrow> children p i s))"

-- "p is a path of length n starting from t leading to a node"
definition nodepath :: "path \<Rightarrow> nat \<Rightarrow> address \<Rightarrow> store \<Rightarrow> bool" where
 "nodepath p n t s = ((path p n t s) \<and> \<not>null(p n))"

-- "t is a tree"
definition tree :: "address \<Rightarrow> store \<Rightarrow> bool" where
-- "no infinite paths respectively cycles and"
-- "no sharing of nodes in multiple paths, also prevents cycles"
 "tree t s =
     ((\<forall>p::path. \<not>ipath p t s) \<and>
      (\<forall>(p1::path) (p2::path) (n1::nat) (n2::nat).
        nodepath p1 n1 t s \<and> nodepath p2 n2 t s \<and>
        p1 n1 = p2 n2 \<longrightarrow>
          n1 = n2 \<and> (\<forall>i::nat. i < n1 \<longrightarrow> p1 i = p2 i)))"

-- "t is a search tree with unique keys"
definition stree :: "address \<Rightarrow> store \<Rightarrow> bool" where
 "stree t s =
     ((tree t s) \<and>
      (\<forall>(p::path)(n::nat).
         nodepath p (n+1) t s \<longrightarrow>
           (let a1 = (p n) in let a2 = (p (n+1)) in
            let n1 = s(a1) in let n2 = s(a2) in
           (a2 = left(n1) \<longleftrightarrow> key(n2) < key(n1)))))"

-- "a is the address of a node in tree t"
definition nodein :: "address \<Rightarrow> address \<Rightarrow> store \<Rightarrow> bool" where
 "nodein a t s = (\<exists>(p::path)(n::nat). (nodepath p n t s) \<and> p n = a)"

-- "k is a key in tree t"
definition keyin :: "key \<Rightarrow> address \<Rightarrow> store \<Rightarrow> bool" where
 "keyin k t s = (\<exists>(a::address). nodein a t s \<and> key (s a) = k)"

-- "k is the minimum key in tree t"
definition minkeyin :: "key \<Rightarrow> address \<Rightarrow> store \<Rightarrow> bool" where
 "minkeyin k t s = (keyin k t s \<and> \<not>(\<exists>k0::key. k0 < k \<and> keyin k0 t s))"

-- {* precondition of deleteMin *}
definition pre :: "address \<Rightarrow> store \<Rightarrow> bool" where
  -- "input is a non-null search tree"
 "pre t x = (stree t x \<and> \<not>null t)"

-- "postcondition of deleteMin"
definition post :: "address \<Rightarrow> store \<Rightarrow> address \<Rightarrow> store \<Rightarrow> bool" where
```

```
  -- "the output is a search tree"
  -- "only the nodes in the input tree may have been changed"
  -- "the only keys in the output tree are those from the input tree"
  -- "a key from the input tree is in the output tree iff it is not the minimum key"
 "post t s oldt olds =
    ((stree t s) \<and>
     (\<forall>a::address.\<not>(nodein a oldt olds) \<longrightarrow> s a = olds a) \<and>
     (\<forall>k::key.(keyin k t s) \<longrightarrow> keyin k oldt olds) \<and>
     (\<forall>k::key.(keyin k oldt olds) \<longrightarrow>
                (keyin k t s \<longleftrightarrow> \<not>minkeyin k oldt olds)))"

-- "-----------------------------------------------------------------------------------------"
-- "Predecessor function and corresponding lemmas"
-- "-----------------------------------------------------------------------------------------"

fun Pre :: "nat \<Rightarrow> nat" where
 "Pre i = (THE x::nat.(Suc x=i))"

lemma haspre :
 "\<forall>i::nat. i \<noteq> 0 \<longrightarrow> (\<exists>x::nat. Suc x=i)"
proof (auto)
  fix i::nat
  assume "0 < i"
  show "\<exists>x. Suc x = i" by (metis ‘0 < i‘ gr0_conv_Suc)
qed

lemma ispre :
 "\<forall>i::nat. i \<noteq> 0 \<longrightarrow> Suc(Pre i) = i"
proof (auto)
  fix i::nat
  assume "0 < i"
  moreover from this haspre have "(\<exists>x. Suc x = i)" by auto
  ultimately show "Suc (THE x. Suc x = i) = i" by auto
qed

-- "-------------------------------------------------------------------------------------------"
-- "Lemma (childpath)"
-- "-------------------------------------------------------------------------------------------"
lemma childpath:
 "\<forall>(t::address)(s::store)(p::path)(n::nat)(t0::address).
     \<not>null t \<and> child t t0 s \<and> nodepath p n t0 s  \<longrightarrow> nodepath (cons t p) (n+1) t s"
proof (auto)
  fix t s p n t0
  assume pre2: "\<not> null t"
  assume pre3: "child t t0 s"
  assume pre4: "nodepath p n t0 s"
  show "nodepath (case_nat t p) (Suc n) t s"
  proof (unfold "nodepath_def")
    show "path (case_nat t p) (Suc n) t s \<and>
          \<not> null (case Suc n of 0 \<Rightarrow> t | Suc x \<Rightarrow> p x)"
    proof (auto)
      show "path (case_nat t p) (Suc n) t s"
      proof (unfold path_def)
        show "(case 0 of 0 \<Rightarrow> t | Suc x \<Rightarrow> p x) = t \<and>
              (\<forall>i<Suc n. \<not> null (case i of 0 \<Rightarrow> t | Suc x \<Rightarrow> p x)) \<and>
              (\<forall>i<Suc n. children (case_nat t p) i s)"
        proof (auto)
          fix i
          assume "i < Suc n"
          assume "null (case i of 0 \<Rightarrow> t | Suc x \<Rightarrow> p x)"
          from this pre2 pre4 nodepath_def show "False"
            by (metis ‘i < Suc n‘ less_Suc_eq_0_disj old.nat.simps(4) old.nat.simps(5) path_def)
```

```
        next
          fix i
          assume "i < Suc n"
          show "children (case_nat t p) i s"
          proof (unfold "children_def")
            from pre3 pre4 nodepath_def[of "p" "n" "t0" "s"] have
            4: "path p n t0 s" and
            5: "\<not> null (p n)" by auto
            show "child (case i of 0 \<Rightarrow> t | Suc x \<Rightarrow> p x)
                        (case i + 1 of 0 \<Rightarrow> t | Suc x \<Rightarrow> p x) s"
            proof (cases)
              assume cond: "i=0"
              from pre3 cond 4 path_def children_def child_def have
                "child t (p 0) s" by auto
              from this cond show ?thesis by auto
            next
              assume cond: "i\<noteq>0"
              from pre3 cond 4 path_def children_def child_def ispre have
               "child (p (Pre i)) (p i) s"
              by (metis Suc_eq_plus1 ‘i < Suc n‘ less_Suc_eq_0_disj old.nat.inject)
              from this cond show ?thesis by (metis Suc_eq_plus1 ispre old.nat.simps(5))
            qed
          qed
        qed
      qed
    next
      from pre4 nodepath_def show "null (p n) \<Longrightarrow> False" by metis
    qed
  qed
qed

-- "-------------------------------------------------------------------------------------------"
-- "Lemma (treeright)"
-- "------------------------------------------------------------------------------------------"
lemma treeright :
 "\<forall>(t::address)(s::store). tree t s \<and> \<not>null t  \<longrightarrow> tree (right(s t)) s"
proof (auto)
  fix t s
  assume pre1: "tree t s"
  assume pre2: "\<not>null t"
  show "tree (right(s t)) s"
  proof (unfold tree_def)

    from pre1 tree_def[of "t" "s"] have
      pre5: "(\<forall>p. \<not> ipath p t s)" by metis
    from pre1 tree_def[of "t" "s"] have
      pre6: "(\<forall>p1 p2 n1 n2. nodepath p1 n1 t s \<and> nodepath p2 n2 t s \<and>
                p1 n1 = p2 n2 \<longrightarrow>
                  n1 = n2 \<and> (\<forall>i<n1. p1 i = p2 i))" by metis

    show "(\<forall>p. \<not> ipath p (right (s t)) s) \<and>
          (\<forall>p1 p2 n1 n2.
             nodepath p1 n1 (right (s t)) s \<and> nodepath p2 n2 (right (s t)) s \<and>
               p1 n1 = p2 n2 \<longrightarrow>
                 n1 = n2 \<and> (\<forall>i<n1. p1 i = p2 i))"

    proof (auto)
      fix p
      assume a1: "ipath p (right (s t)) s"
      show False
      proof -
        from a1 ipath_def[of "p" "right (s t)" "s"] have
```

```
      n1: "(p 0 = right (s t) \<and> (\<forall>i. \<not> null (p i)) \<and>
           (\<forall>i. children p i s))" by auto
    let ?p0 ="cons t p"
    have "ipath ?p0 t s"
    proof (unfold ipath_def)
      show "cons t p 0 = t \<and> (\<forall>i. \<not> null (cons t p i)) \<and>
            (\<forall>i. children (cons t p) i s)"
      proof (auto)
        fix i
        assume "null (case i of 0 \<Rightarrow> t | Suc x \<Rightarrow> p x)"
        from this pre2 n1 show False by (metis Nitpick.case_nat_unfold)
      next
        fix i
        show "children (case_nat t p) i s"
        proof (unfold children_def)
          from n1 children_def child_def show
          "child (case i of 0 \<Rightarrow> t | Suc x \<Rightarrow> p x)
                 (case i + 1 of 0 \<Rightarrow> t | Suc x \<Rightarrow> p x) s"
          by (metis Suc_eq_plus1 lessI less_Suc_eq_0_disj old.nat.simps(4) old.nat.simps(5))
        qed
      qed
    qed
    from this pre5 show False by auto
  qed
next
  fix p1 p2 n1 n2
  assume 1: "nodepath p1 n1 (right (s t)) s"
  assume 2: "nodepath p2 n2 (right (s t)) s"
  assume 3: "p1 n1 = p2 n2"
  show "n1 = n2"
  proof -
    let ?q1 ="cons t p1" and ?q2 ="cons t p2" and ?m1 ="n1+1" and ?m2 ="n2+1"
    from pre6 have
     "nodepath ?q1 ?m1 t s \<and> nodepath ?q2 ?m2 t s \<and> ?q1 ?m1 = ?q2 ?m2 \<longrightarrow>
         ?m1 = ?m2 \<and> (\<forall>i<?m1. ?q1 i = ?q2 i)" by blast
    from this show "n1 = n2"
    proof (auto)
      assume "n1 \<noteq> n2"
      from pre2 1 childpath child_def show "nodepath (case_nat t p1) (Suc n1) t s"
      by (metis One_nat_def add_Suc_right cons.simps monoid_add_class.add.right_neutral)
    next
      assume "n1 \<noteq> n2"
      from pre2 2 childpath child_def show "nodepath (case_nat t p2) (Suc n2) t s"
      by (metis One_nat_def add_Suc_right cons.simps monoid_add_class.add.right_neutral)
    next
      assume "n1 \<noteq> n2"
      from 3 show "p1 n1 = p2 n2" by auto
    qed
  qed
next
  fix p1 p2 n1 n2 i
  assume 1: "nodepath p1 n1 (right (s t)) s"
  assume 2: "nodepath p2 n2 (right (s t)) s"
  assume 3: "p1 n1 = p2 n2"
  assume 4: "i < n1"
  show "p1 i = p2 i"
  proof -
    let ?q1 ="(cons t p1)" and ?q2 ="(cons t p2)"
    let ?m1 ="n1+1" and ?m2 ="n2+1"
    from pre6 have
     "nodepath ?q1 ?m1 t s \<and> nodepath ?q2 ?m2 t s \<and> ?q1 ?m1 = ?q2 ?m2 \<longrightarrow>
         ?m1 = ?m2 \<and> (\<forall>i<?m1. ?q1 i = ?q2 i)" by blast
```

```
        from this show ?thesis
        proof (auto)
          assume "p1 i \<noteq> p2 i"
          from 1 pre2 childpath child_def show "nodepath (case_nat t p1) (Suc n1) t s"
          by (metis One_nat_def add_Suc_right cons.simps
                    monoid_add_class.add.right_neutral pre2)
        next
          assume "p1 i \<noteq> p2 i"
          from 2 pre2 childpath child_def show "nodepath (case_nat t p2) (Suc n2) t s"
          by (metis One_nat_def add_Suc_right cons.simps
                    monoid_add_class.add.right_neutral pre2)
        next
          assume "p1 i \<noteq> p2 i"
          from 3 show "p1 n1 = p2 n2" by auto
        next
          assume 5: "\<forall>i<Suc n2. (case i of 0 \<Rightarrow> t | Suc x \<Rightarrow> p1 x) =
                              (case i of 0 \<Rightarrow> t | Suc x \<Rightarrow> p2 x)"
          assume 6: "n1 = n2"
          from 5 6 show "p1 i = p2 i" by (metis "4"Suc_less_eq old.nat.simps(5))
        qed
      qed
    qed
  qed
qed

-- " ---------------------------------------------------------------------------------------------- "
-- "Lemma (keyinright)"
-- "---------------------------------------------------------------------------------------------- "
lemma keyinright:
 "\<forall>(t::address)(s::store)(k::key). \<not>null t \<and> keyin k (right (s t)) s \<longrightarrow> keyin k t s"
proof (auto)
  fix t s k
  assume pre2: "\<not>null t"
  assume a1: "(keyin k (right(s t)) s)"
  show "keyin k t s"
  proof (unfold keyin_def)
    show "\<exists>a. nodein a t s \<and> key (s a) = k"
    proof -
      from a1 keyin_def[of "k" "(right(s t))" "s"] have
        a2: "(\<exists>a. nodein a (right (s t)) s \<and> key (s a) = k)" by auto
      from a2 obtain a where
        a3: "nodein a (right (s t)) s" and
        a4: "key (s a) = k" by auto
      from a3 nodein_def[of "a" "right (s t)" "s"] obtain "p" "n" where
        a5: "nodepath p n (right (s t)) s" and
        a6: "p n = a" by auto
      have "nodein a t s \<and> key (s a) = k"
      proof (auto)
        show "nodein a t s"
        proof (unfold nodein_def)
          show "\<exists>p n. nodepath p n t s \<and> p n = a"
          proof -
            let ?p = "cons t p"
            let ?n = "n+1"
            have "nodepath ?p ?n t s \<and> ?p ?n = a"
            proof (auto)
              from childpath nodepath_def children_def child_def pre2 a5 a6 show
              "nodepath (case_nat t p) (Suc n) t s" by (metis Suc_eq_plus1 cons.simps)
            next
              from cons_def a6 show "p n = a" by auto
            qed
            from this show ?thesis by blast
```

```
          qed
        qed
      next
        from a4 show "key (s a) = k" by auto
      qed
      from this show ?thesis by auto
    qed
  qed
qed

text {*
{\small
\begin{verbatim}
  static Tree deleteMin(Tree t) {
      Tree p = t.left;
      if (p == null)
        t = t.right;
      else {
        Tree p2 = t;
        Tree tt = p.left;
        while (tt != null) {
          p2 = p; p = tt; tt = p.left;
        }
        p2.left = p.right;
      }
      return t;
  }
\end{verbatim}
}
*}

-- "-----------------------------------------------------------------------------------------------"
-- "Verification Condition 1"
-- "correctness of first program path in which store is not changed "
-- "assume pre; p = t.left; assume p == null; t = t.right; assert post;"
-- "-----------------------------------------------------------------------------------------------"
theorem VC1 :
 "\<forall>(t::address)(s::store). pre t s \<and> null (left(s t)) \<longrightarrow> post (right(s t)) s t s"

proof (auto)
  fix t s
  assume pre: "pre t s"
  assume "null (left(s t))"
  show "post (right(s t)) s t s"

  proof (unfold post_def)

    from pre pre_def[of "t" "s"] have
      pre1: "stree t s" and
      pre2: "\<not> null t" by auto
    from pre1 stree_def[of "t" "s"] have
      pre3: "tree t s" and
      pre4: "\<forall>p n. nodepath p (n + 1) t s \<longrightarrow>
                (let a1 = p n; a2 = p (n + 1); n1 = s a1; n2 = s a2 in
                (a2 = left n1) = (key n2 < key n1))" by auto
    from pre3 tree_def[of "t" "s"] have
      pre5: "(\<forall>p. \<not> ipath p t s)" by metis
    from pre3 tree_def[of "t" "s"] have
      pre6: "(\<forall>p1 p2 n1 n2. nodepath p1 n1 t s \<and> nodepath p2 n2 t s\<and> p1 n1 = p2 n2 \<longrightarrow>
                n1 = n2 \<and> (\<forall>i<n1. p1 i = p2 i))" by metis
    show
    "stree (right (s t)) s \<and>
```

```
      (\<forall>a. \<not> nodein a t s \<longrightarrow> s a = s a) \<and>
      (\<forall>k. keyin k (right (s t)) s \<longrightarrow> keyin k t s) \<and>
      (\<forall>k. keyin k t s \<longrightarrow> keyin k (right (s t)) s = (\<not> minkeyin k t s))"

    proof (auto)

      -- "subproof "
      show "stree (right(s t)) s"
      proof (unfold stree_def)
        show
        "tree (right (s t)) s \<and>
          (\<forall>p n. nodepath p (n + 1) (right (s t)) s \<longrightarrow>
          (let a1 = p n; a2 = p (n + 1); n1 = s a1; n2 = s a2 in
            (a2 = left n1) = (key n2 < key n1)))"
        proof (auto)
          from treeright pre2 pre3 show "tree (right (s t)) s" by auto
        next
          fix p n
          assume "nodepath p (Suc n) (right (s t)) s"
          from this pre4 nodepath_def childpath child_def pre2
          show "let a2 = p (Suc n); n1 = s (p n) in (a2 = left n1) = (key (s a2) < key n1)"
          by (metis Suc_eq_plus1 cons.simps old.nat.simps(5))
        qed
      qed

    -- "subproof "
    next
      fix k
      assume "keyin k (right (s t)) s"
      from keyinright pre2 this show "keyin k t s" by auto

    -- "subproof "
    next
      fix k
      assume "keyin k t s"
      assume "keyin k (right (s t)) s"
      assume "minkeyin k t s"
      show False sorry

    -- "subproof "
    next
      fix k
      assume "keyin k t s"
      assume "\<not> minkeyin k t s"
      show "keyin k (right (s t)) s" sorry
    qed
  qed
qed

end
```