Automated Reasoning: Coursework 1
Proving and Reasoning in Isabelle/HOL

Released 2 February 2016
Updated 9 February 2016

1 Introduction

This is the first coursework assignment for the Automated Reasoning course. It is divided into two parts. In the first part, you will get some experience with the rules of natural deduction, building on the exercises that are on the course web page. In the second part, you will work on a verification task, using Isabelle's more powerful proof tools in order to prove theorems about an inductively defined data structure. You will also get to formalise your own definitions.

There are two (essentially identical) versions of this document. The source version, which you can load into Isabelle to complete your proofs:

http://www.inf.ed.ac.uk/teaching/courses/ar/coursework/Coursework1.thy

and the PDF version, which is easier to read:

http://www.inf.ed.ac.uk/teaching/courses/ar/coursework/Coursework1.pdf

You should fill in the source version with your proofs for submission. Submission instructions are at the end of this document.

The deadline for submission is **4pm, Friday 26th February 2016**.

2 Marks

You will only earn marks for an unfinished proof if you provide some explanation as to your proof strategy or an explanation as to why you are stuck. You may also earn marks if you can prove the theorem by asserting a sensible lemma with `lemma` and "proving" it with the "cheat" command `sorry`. Be careful to note the restrictions on the proof methods that your are allowed to use for each question.

3 Background Reading

This assignment uses the interactive theorem prover Isabelle, introduced in the lectures and exercises.

As you will be using Isabelle interactively, you will need to be familiar with the system before you start. Formalized mathematics is not trivial! You will find this assignment much easier if you attend the lectures, attempt the various Isabelle exercises given on the course webpages, and ask questions about using Isabelle before you start. It is recommended that you read Chapter 5 of the Isabelle/HOL tutorial located at:

http://isabelle.in.tum.de/doc/tutorial.pdf

We also recommend that you use the Isabelle Cheat Sheet from Jeremy Avigad, which can be found at:

http://www.inf.ed.ac.uk/teaching/courses/ar/FormalCheatSheet.pdf

4 Part 1 : Natural Deduction in Isabelle/HOL (30 marks)

In this section, you will get some practice with natural deduction by proving some theorems from propositional and first-order logic. Each of these theorems could be solved directly with Isabelle's automatic tactics, but here, you are asked to use only the following basic introduction and elimination rules:

conjI	:	$[\![?P;\ ?Q]\!] \Longrightarrow\ ?P \wedge ?Q$
conjE	:	$[\![?P \wedge ?Q;\ [\![?P;\ ?Q]\!] \Longrightarrow\ ?R]\!] \Longrightarrow\ ?R$
conjunct1	:	$?P \wedge ?Q \Longrightarrow\ ?P$
conjunct2	:	$?P \wedge ?Q \Longrightarrow\ ?Q$
disjI1	:	$?P \Longrightarrow\ ?P \vee ?Q$
disjI2	:	$?Q \Longrightarrow\ ?P \vee ?Q$
disjE	:	$[\![?P \vee ?Q;\ ?P \Longrightarrow\ ?R;\ ?Q \Longrightarrow\ ?R]\!] \Longrightarrow\ ?R$
impI	:	$(?P \Longrightarrow\ ?Q) \Longrightarrow\ ?P \longrightarrow\ ?Q$
impE	:	$[\![?P \longrightarrow\ ?Q;\ ?P;\ ?Q \Longrightarrow\ ?R]\!] \Longrightarrow\ ?R$
mp	:	$[\![?P \longrightarrow\ ?Q;\ ?P]\!] \Longrightarrow\ ?Q$
notI	:	$(?P \Longrightarrow \mathit{False}) \Longrightarrow \neg\ ?P$
notE	:	$[\![\neg\ ?P;\ ?P]\!] \Longrightarrow\ ?R$
iffI	:	$[\![?P \Longrightarrow\ ?Q;\ ?Q \Longrightarrow\ ?P]\!] \Longrightarrow\ ?P =\ ?Q$
iffE	:	$[\![?P =\ ?Q;\ [\![?P \longrightarrow\ ?Q;\ ?Q \longrightarrow\ ?P]\!] \Longrightarrow\ ?R]\!] \Longrightarrow\ ?R$
iffD1	:	$[\![?Q =\ ?P;\ ?Q]\!] \Longrightarrow\ ?P$
iffD2	:	$[\![?P =\ ?Q;\ ?Q]\!] \Longrightarrow\ ?P$
allI	:	$(\bigwedge x.\ ?P\ x) \Longrightarrow \forall x.\ ?P\ x$
allE	:	$[\![\forall x.\ ?P\ x;\ ?P\ ?x \Longrightarrow\ ?R]\!] \Longrightarrow\ ?R$
spec	:	$\forall x.\ ?P\ x \Longrightarrow\ ?P\ ?x$
exI	:	$?P\ ?x \Longrightarrow \exists x.\ ?P\ x$
exE	:	$[\![\exists x.\ ?P\ x;\ \bigwedge x.\ ?P\ x \Longrightarrow\ ?Q]\!] \Longrightarrow\ ?Q$

You may also use the following *classical rules*:

excluded_middle	:	$\neg\ ?P \vee\ ?P$
ccontr	:	$(\neg\ ?P \Longrightarrow \mathit{False}) \Longrightarrow\ ?P$

Note that you can display any of these theorems, and any other named theorem, even while in the middle of a proof. For instance, to display the rule `conjI`, just use the following command:

```
thm conjI
```

In each step of the proof, you may use apply with any of the methods `rule`, `erule`, `drule`, `frule`, and their variants `rule_tac`, `erule_tac`, `drule_tac` and `frule_tac`. You will also need to use the method `assumption`, and you may also use the commands `defer` and `prefer` to manipulate the goal-stack during a proof. You are **not** permitted to use any other proof methods for this part.

Prove the following lemmas by replacing the placeholder proof `oops` with a real proof in each case:

lemma $P \longrightarrow P$
oops

(1 marks)

lemma $P \wedge Q \longrightarrow Q \wedge P$
oops

(1 mark)

lemma $(Q \wedge R) \wedge P \longrightarrow (P \wedge R) \wedge Q$
oops

(1 mark)

lemma $(Q \vee R) \wedge P \longrightarrow \neg P \longrightarrow Q$
oops

(1 mark)

lemma $(\forall x.\ P\ x \longrightarrow Q\ x) \longrightarrow \neg (\exists x.\ P\ x \wedge \neg Q\ x)$
oops

(5 marks)

lemma $(\exists x.\ \forall y.\ P\ x\ y) \longrightarrow (\forall y.\ \exists x.\ P\ x\ y)$
oops

(5 marks)

lemma $\neg (\exists \mathit{barber}.\ \mathit{man}\ \mathit{barber} \wedge (\forall x.\ \mathit{man}\ x \wedge \neg\ \mathit{shaves}\ x\ x \longleftrightarrow \mathit{shaves}\ \mathit{barber}\ x))$
oops

(8 marks)

lemma $(\forall (x{::}\mathit{int}).\ (\exists y.\ P\ x\ y) \longrightarrow Q\ x) \wedge (\forall R\ (x{::}\mathit{int})\ y.\ R\ x\ y \longrightarrow R\ y\ x) \longrightarrow (\forall z.\ P\ a\ z \longrightarrow Q\ z)$
oops

(8 marks)

5 Part 2. Binary Space Partitioning (70 marks)

In this part of the assignment, we will look at a software verification exercise, and formally verify properties of a binary tree data structure (see http://en.wikipedia.org/wiki/Binary_tree).

Our binary trees will be used to define a region of space by recursively performing *binary partitions.* This technique is typically used in computer graphics and simulations, to provide a fast way to compute *collisions* with the defined regions, and to determine which objects are potentially visible to a virtual camera.

Our binary trees will consider a one-dimensional case. They recursively partition a line segment into disjoint pieces. At each branch of the tree, the line-segment is divided in half, so that for a tree of depth n, we are able to consider 2^n possible subdivisions. At the leaves of the tree, we specify whether the corresponding partition is Empty of points, or whether it is Filled with points. See Figure 1 for an example.

For more information on partitioning in this way, see http://en.wikipedia.org/wiki/Quadtree, where the two-dimensional case is considered.

The following defines the *datatype* of our binary partition trees. A tree is either an Empty or a Filled *leaf*, or else it is a *Branch*, splitting the space into two partitions.

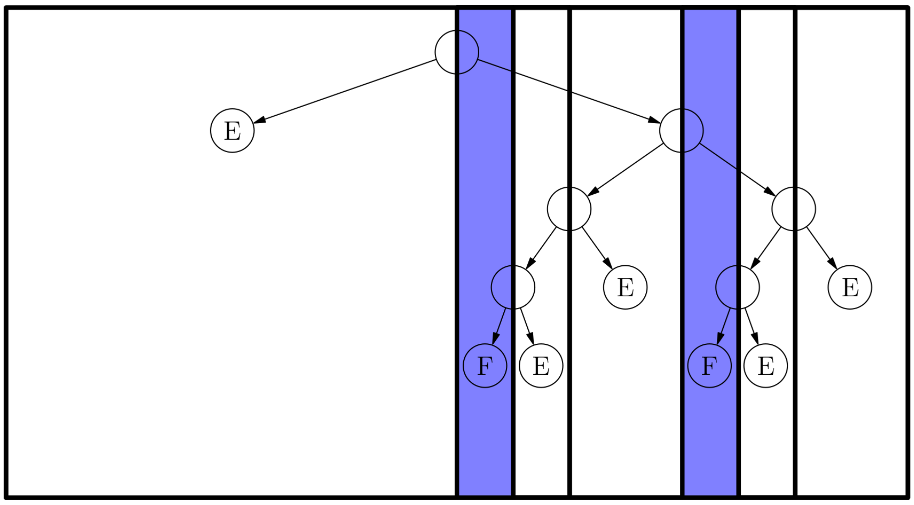

Figure 1: Binary space partitioning in one dimension

datatype *partition* = *Empty* | *Filled* | *Branch partition partition*

Since partitions define parts of a line-segment, it is possible to define a union and intersection operation on them, where we understand the arguments to these functions as partitions of the same arbitrary segment. Here is the definition of the union function, which we can also write with the infix notation ⊔, written as `\<squnion>` in Isabelle.

fun *union* :: *partition* ⇒ *partition* ⇒ *partition* (**infixr** ⊔ *65*) **where**
  union (*Empty*) *q* = *q*
| *union* (*Filled*) *q* = *Filled*
| *union p* (*Empty*) = *p*
| *union p* (*Filled*) = *Filled*
| *union* (*Branch l1 r1*) (*Branch l2 r2*) = *Branch* (*union l1 l2*) (*union r1 r2*)

Here is the definition of an intersection function, which we can also write as ⊓ (`\<sqinter>`).

fun *intersect* :: *partition* ⇒ *partition* ⇒ *partition* (**infixr** ⊓ *65*) **where**
  intersect (*Empty*) *q* = *Empty*
| *intersect* (*Filled*) *q* = *q*
| *intersect p* (*Empty*) = *Empty*
| *intersect p* (*Filled*) = *p*
| *intersect* (*Branch l1 r1*) (*Branch l2 r2*) = *Branch* (*intersect l1 l2*) (*intersect r1 r2*)

Finally, here is a function to *invert* or *complement* a partition.

fun *invert* :: *partition* ⇒ *partition* **where**
  invert (*Empty*) = *Filled*
| *invert* (*Filled*) = *Empty*
| *invert* (*Branch l r*) = *Branch* (*invert l*) (*invert r*)

5.1 Simple Proofs by Induction (5 marks)

For this first proof, you will use the rules of natural deduction from Part 1 and equational rules given below. In addition, you will have some rules which are automatically proven by Isabelle, based on the data-type and function definitions above. These rules will help you simplify expressions involving these functions, and consider the cases which define each of their equations.

The rules are named *partition.induct*, *union.simps*, *intersect.simps*, *invert.simps*, *union.cases*, *intersect.cases* and *invert.cases*. You can use the *thm* command to display these rules. Now, with these new rules, and the same basic proof methods used in Part 1, prove that the operation *invert* is self-inverse.

Additionally, you might find useful these rules which help reasoning about equality:

refl	:	$?t = ?t$
sym	:	$?s = ?t \Longrightarrow ?t = ?s$
trans	:	$[\![?r = ?s;\ ?s = ?t]\!] \Longrightarrow ?r = ?t$
subst	:	$[\![?s = ?t;\ ?P\ ?s]\!] \Longrightarrow ?P\ ?t$

For this question, you cannot use the *auto*, *blast* or *simp* methods.

You are provided below with a proof for this theorem, written in an earlier version of Isabelle. For this question, you have the possibility of trying to fix this proof or to try to write a proof of your own.

theorem *invert-invert*: $\forall\, p.\ invert\ (invert\ p) = p$

```
  apply (rule allI)
  apply (rule partition.induct)
  apply (rule_tac ?t="invert Empty" in subst)
  apply (rule sym)
  apply (rule invert.simps)+
  apply (rule_tac ?t="invert Filled" in subst)
  apply (rule sym)
  apply (rule invert.simps)+
  apply (rule_tac ?t="invert (Branch partition1 partition2)" in subst)
  apply (rule sym)
  apply (rule invert.simps)+
  apply (rule_tac ?t="invert (Branch (invert partition1) (invert partition2))" in subst)
  apply (rule sym)
  apply (rule invert.simps)+
  apply (drule sym)
  apply (drule_tac ?t="partition2" in sym)
  apply (erule subst)+
  by (rule refl)
```

oops

(5 marks)

5.2 Using More Powerful Proof Methods (10 marks)

In proofs such as these, it is usually much easier to use the following tactics. From here on, you may use them freely:
case-tac : perform case-analysis on a given variable.
induct-tac : apply an induction rule on a given variable.
simp : the (contextual) rewriter
auto : the classical reasoner.
blast : a powerful tableau-prover for first-order reasoning.

The *simplifier* deserves particular mention. It is a powerful automated tool that tries to rewrite terms as much as possible. It is particularly useful because it already knows a basic set of equations with which to rewrite terms. Though you can explicitly supply your own theorems with *add*, the simplifier already knows plenty of rewrites, including *union.simps*, *intersect.simps* and *invert.simps*. With this tool, proofs become much shorter:

```
theorem invert-invert-simp: ∀ p. invert (invert p) = p
apply auto
apply (induct-tac p)
by simp+
```

Note that *auto* also performs some simplification, before attempting to solve the goal. As with (*simp add*: <*lemma*>), you can supply your own simplification rules when using *auto*, with the command (*auto simp*: <*lemma*>).

A proof of the De Morgan law for our functions is provided below:

```
theorem demorgan: ∀ p1 p2. (invert p1) ⊓ (invert p2) = invert (p1 ⊔ p2)
  apply (rule allI)
  apply (induct-tac p1)
  apply auto
  apply (case-tac p2)
  by simp+
```

Using the lemma *demorgan* and any other appropriate lemmas, prove the following theorem (without using the *metis* tactic). Note that you *must* make use of *demorgan* in your proof.

```
theorem demorgan2: ∀ p1 p2. (invert p1) ⊔ (invert p2) = invert (p1 ⊓ p2)
oops
```

(10 marks)

5.3 Formalising and Proving Properties of Partitions (10 marks)

So far, definitions for the *partition* type and various functions and lemmas about them have been given to you and you were asked to prove them. This part of the coursework is a small formalisation exercise in which you are asked to write some definitions and lemmas of your own before attempting to prove them. When doing formalised mathematics, laying down these definitions is often the hardest part. The choices made at this stage can have a significant impact on the difficulty of proving the subsequent theorems.

First, define a function *mirror*, which, given a *partition*, returns the same partition, in which the order of the children in branches has been swapped.

(2 marks)

Then, write down and prove theorems which verify the following properties of *mirror*:

1. Mirroring a partition twice results in the original partition.
2. First mirroring and then inverting a partition is the same as first inverting and then mirroring it.
3. The mirror of the union of two partitions is the same as the union of the mirrors of each partition.
4. The same property as (3), but for intersection.

(8 marks)

5.4 Simplifying Partitions (45 marks)

We'll now consider a function which will *simplify* (or normalise) a partition. This function is useful because, as you may have noticed, it is possible for there to be many different but *equivalent* ways to express a partition as a tree (see Figure 2).

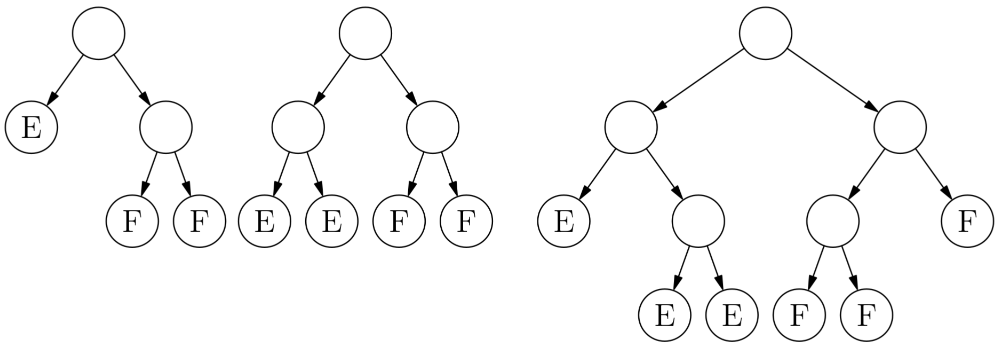

Figure 2: Different binary trees representing equivalent space partitions

Let us define two functions to reduce these equivalent trees to the same tree:

```
fun simplify1 :: partition ⇒ partition where
simplify1 (Filled)  = Filled
| simplify1 (Empty)  = Empty
| simplify1 (Branch (Filled) (Filled)) = Filled
| simplify1 (Branch (Empty) (Empty)) = Empty
| simplify1 p = p

fun simplify :: partition ⇒ partition where
simplify (Filled) = Filled
| simplify (Empty) = Empty
| simplify (Branch (Filled) (Filled)) = Filled
| simplify (Branch (Empty) (Empty)) = Empty
| simplify (Branch p q) = simplify1 (Branch (simplify p) (simplify q))
```

How can we be sure that these functions are correct? One obvious property is that they must preserve the effects of all of our operations. We can make a start towards formally verifying this by proving some theorems.

Prove that, under normalisation, a partition unified with its inverse will always result in the identity element for unification, which is *Filled*:

theorem $\forall\, p.$ *simplify* $(p \sqcup (\mathit{invert}\ p)) = \mathit{Filled}$
oops

(5 marks)

Note that intersection and *Empty* behave in the same way. Now, prove that simplifying the inverse of a partition has the same result as inverting its simplification:

lemma *simplify-invert*: $\forall\, p.$ *simplify* $(\mathit{invert}\ p) = \mathit{invert}\ (\mathit{simplify}\ p)$
oops

(10 marks)

Next, we will consider the union operation. Here is a conjecture we might, mistakenly, try to prove:

theorem *simplify-union-wrong*: *simplify* $p \sqcup$ *simplify* $q =$ *simplify* $(p \sqcup q)$
oops

However, Isabelle's QuickCheck tool should automatically provide you with a counterexampe which demonstrates that this conjecture is false.

The correct theorem requires an extra simplification on the left-hand side. Your next task is to formally verify it. However, you will probably need to prove additional lemmas to make a start. There are multiple paths when it comes to proving this theorem. Think carefully about the lemmas that you will need in order to construct a proof.

theorem *simplify-union*: $\forall\, p1\ p2.$ *simplify* $(\mathit{simplify}\ p1 \sqcup \mathit{simplify}\ p2) =$ *simplify* $(p1 \sqcup p2)$
oops

(30 marks)

end

6 Demonstrator Hours

The TA for the first half of the course, Victor Dumitrescu, will be available in **FH 1.B31** between **3pm and 5pm on Thursdays** to help with any problems with course material and coursework. There will be 4 lab sessions, from Week 3 (Thu 28/01) to Week 6 (Thu 25/02). Note that there will be no lab session during Innovative Learning Week (15-19 Feb).

You may also send questions directly to `victor.dumitrescu@ed.ac.uk` or post questions to the course mailing list at `ar-students@inf.ed.ac.uk`.

7 Submission

By **4pm** on **Friday 26th February 2016**, you must submit your theory file by typing the following command at a DICE terminal:

```
submit ar 1 Coursework1.thy
```

Late coursework will be penalised in accordance with the Informatics standard policy. Please consult your course guide for specific information about this. Also note that, while we encourage students to discuss the practical among themselves, we take plagarism seriously and any such case will be treated appropriately. Please consult your student guide for more information about this matter.