A Short Isabelle/HOL Tutorial for the Functional Programmer

Thomas Genet

Four Pages Isabelle/HOL Tutorial for the Functional Programmer

Thomas Genet

1 Introduction

The objective of this (very) short tutorial is to help any functional programmer to quickly put its hand on Isabelle/HOL and catch a glimpse of its power. Then, if you want some more, you should refer to the extensive Isabelle/HOL tutorial and documentation available in the tool.

2 Installing and Starting Isabelle/HOL

Download Isabelle/HOL from `https://isabelle.in.tum.de/`. This tutorial was done using the 2015 version. If your operating system is neither old or exotic, installation should be easy. Otherwise, you should read the installation instructions `https://isabelle.in.tum.de/installation.html`.

The first time you start it, Isabelle/HOL compiles all the basic theories. It may take some time but this is done once for all. Starting Isabelle will be faster the next time.

When Isabelle/HOL main window opens it should be divided in three frames. The larger frame on the top left is used to type programs and lemmas. Let us call it the **Theory** frame. The frame on the rightmost position gather all built-in Isabelle documentation. The frame at the bottom is for output of all program runs and proof goals. If the **Output** frame is not visible, you should click on the "Output" tab on the bottom left of the window to make it visible.

3 Defining and Running Programs

In the Theory frame type the following:

```
theory Scratch
imports Main
begin
```

where the name of the theory (here `Scratch`) should match with the file name (check that it is `Scratch.thy` in the leftmost topmost bar over the Theory frame). Then, under `begin` we can type our first program. For instance, type in the following code:

```
fun remove:: "'a => 'a list => 'a list"
where
"remove x [] = []" |
"remove x (y#ys) = (if x=y then ys else y#(remove x ys))"
```

While typing this program you may have noticed two things. First, Isabelle/HOL replaces all combinations `=>` by the nicer symbol ⇒. Second, in the Output frame, Isabelle/HOL has automatically proven the termination of your program by finding a termination order. About the syntax, we use the `fun` construct to define the function named `remove`, taking two parameters of type `'a` and `'a list` and whose result is of type `'a list`. The function is then defined using recursive equations. Note that the order on equations matters: to evaluate a function all, the first matching equation is used. Lists can be built using the empty list `[]` and the "cons" operator denoted by `#`. Lists can also be defined using shortcuts. For instance, the list `[1,2,3]` is a shortcut for `1#(2#(3#[]))`. Now let's try our function `remove`, which should remove an element from a list. Below the definition of `remove` type a simple test:

```
value "remove 2 [1,2,3]"
```

In the Output frame, you can see the result, which is not the expected one: `"remove (1 + 1) [1, 1 + 1, 1 + 1 + 1]"::"'a list"`. Isabelle/HOL has decomposed the integers but not evaluated the function. However, the type of the expression `'a list` shows that the type of integers have not been correctly inferred. This is due to the face that, by default in Isabelle, symbols "0", "1", etc. are not uniquely associated to the type `int`. Thus, evaluating `value "1"` results into the output `"1"::"'a"` meaning that the type of "1" is not fixed. To denote the integer 1, we need to annotate the symbol "1" by the desired type: `value "(1::nat)"` results in the integer 1. Let's try again on our example:

```
value "remove (2::int) [1,2,3]"
```

Now, we have the expected result. Note that typing explicitely one parameter was enough for the desired type to be inferred on the whole expression.

4 Automate the Search for Bugs

The next step is to check if our function contains errors. We could write some more tests using `value` and verify, test after test, that the result fulfills our expectations. Instead, we can define a property that our function is supposed to have, using a logical formula. For instance, type in the following sentence, where the ¬ logical symbol stands for "not". To get the ¬ operator type ~ character, a selection box opens and then type the Tab key to select the symbol[1].

```
lemma "¬ (List.member (remove e l) e)"
```

The function `List.member::'a list ⇒ 'a ⇒ bool` is part of Isabelle's `List` library and results in the boolean `True` if the list contains the given element, and `False` otherwise. The formula states that the result of `remove e l` should **not** contain the element `e`. Since we provide no information on `e` or `l`, they are free variables. Thus, the formula we just defined is supposed to hold for all possible values of `e` (of type `'a`) and `l` (of type `'a list`).

The effect of the `lemma` command is to define the property and, at the same time, to switch Isabelle/HOL in proof mode. Below `lemma` Isabelle/HOL waits for proof commands. In this mode you can no longer define functions nor state new lemmas. On the current lemma, we can first search for small finite counterexamples. This can be done by typing the `nitpick` command below the `lemma` line. While typing the first letters of the word `nitpick` a selection box where you can select the command you want to insert using the arrows of your keyboard and then typing on the Tab key. Once the `nitpick` command is inserted, it finds a counterexample to our lemma. With `l = [a1,a1]` and `e = a1` the lemma we stated is false. In other words, for any list `l` with two occurences of the same element, the lemma is false. Let's check `remove`'s behavior on a similar example. Typing in `value "remove (1::int) [1,1]"` permits to see that `remove` does not do the job: the result is `[1]`. Our function is wrong. Click on the definition of `remove` and correct it. In the "then" part of the second equation, one should recursively call `remove` on `ys`, *i.e.* the second equation should be: `"remove x (y#ys) = (if x=y then (remove x ys) else y#(remove x ys))"`. Once corrected, you can click on the `value` lines to check that the result is the expected one, and click on the `nitpick` line to check that no more counterexamples are found.

Now, let's try to define a second property. You first have to close the previous one to get out of Isabelle's proof mode. There are three ways to close the proof of a lemma: `done` when the proof is finished, `sorry` when you think that the lemma is correct but do not have completed the proof, and `oops` to close the proof mode when the lemma was shown to be false (if there are still counterexamples, for instance). In our case, we corrected the function and the lemma is likely to be true but we did not yet have the proof, we thus type `sorry` below the nitpick command. With our last lemma we expect to show that `remove x l` removes all occurences of `x` in `l` but what about the other elements in the list? Below `sorry`, let's type the following lemma which intends to check that all the other elements of the list have not disappeared:

```
lemma "(length l) = (length (remove e l)) + (count l e)"
```

Note that you have to type a space character between `(length l)` and "`=`" to avoid Isabelle/HOL to replace the combination "`)=`" by the symbol "⊇". In this lemma, the function `length::'a list ⇒ nat` returns the number of elements in a list and `count::'a list ⇒ 'a ⇒ nat` counts the number of occurrences of a given element in a list. Thus, this lemma states that the number of elements in `l` is equal to the number of elements in `(remove x l)` plus the number of occurrences of `x` in `l`. Again, the first thing to check is that there is no counterexample. Below the lemma, type `nitpick`. It finds a counterexample where `count` is considered has a

[1] If you look for more logical symbols like ∧, ∨, ∀ or ∃, click on the "Symbols" tab at the bottom of the main frame and then on the "Logic" or "Arrow" tabs. When you have found the symbol you are looking for, you can either click on it to insert it or hoover the mouse pointer over it to see what are the abbreviations you can type.

free variable, as `l` and `x`. Note that this is not the case for `length` and `remove`. The reason is that the function `count` is unknown in this theory. You can check this by typing on separate lines, above the current lemma: `value "remove"`, `value "length"`, and `value "count"`. The first two are known and the expected type of the function is returned, but the last is unknown and `"count"::'a` is returned. By default, `count` is not visible from outside of the List library. This time, the counterexample is not due to a bug in the function but to an error in the lemma: it states a property on an unknown function `count` and nitpicks invents a value for this function, falsifying the lemma. To denote the `count` function of the List library, we can use the `List.count` notation. Back to your lemma, modify it as follows:

```
lemma "(length l) = (length (remove e l)) + (List.count l e)"
```

Now, `nitpick` no longer finds counterexamples.

5 No more bugs? let's go for a proof

The fact that `nitpick` finds no counterexample does not imply that the property is true. To be sure that it holds we need to prove it using Isabelle/HOL proof commands. First, we prove the last lemma on `remove`, `length` and `List.count`. Since we know that there are no more simple counterexamples on the lemma, you can remove the `nitpick` command. Click back on the lemma line. The Output frame presents the proof state that consists, for the moment, of a unique proof subgoal which is exactly the property that we want to prove. We are going to apply the main Isabelle proof command `auto` to see if this goal can be automatically simplified as it is. Under the lemma, type `apply auto`. In the output frame, we can see that `auto` failed to simplify the proof goal. This is not surprising since `e` and `l` are free variables. In particular, we do not know if we can apply the first equation of the definition of `remove` since we do not know whether `l` is `[]` or not. To give more information to Isabelle about `l`, before the `apply auto` line, type `apply (induct l)`. Applying induction on `l` has split the proof goal in two subgoals:

```
1. length [] = length (remove e []) + List.count [] e
2. ⋀a l. length l = length (remove e l) + List.count l e
                    ⟹ length (a # l) = length (remove e (a # l)) + List.count (a # l) e
```

The first subgoal corresponds to the case where `l` is known to be `[]`, *i.e.* `l` has been replaced by `[]` in the subgoal. In the second subgoal, `a` and `l` become new fresh variables, local to this formula. This is the meaning of the notation "`⋀a l.`" on the leftmost part of the formula. This second subgoal is built with Isabelle's deduction symbol "`⟹`". To solve a goal of the form $A \Longrightarrow B$, the objective is to prove B using the additional hypothesis A. This time, `apply auto` knows how to simplify those two goals. Click on the `apply auto` line below `apply (induct l)` to see its effect: there are no proof subgoals left. This concludes your first proof. You can now close it using the `done` command.

Now, let's go back to the first lemma we wrote on `List.member` and `remove`, and prove it. You can remove the nitpick command. As above, an `apply auto` would not succeed because the definition of `remove` cannot be used to simplify the proof goal. Type `apply (induct l)` and then `apply auto`. This time, `apply auto` did not succeed to fully solve the subgoals. If you click on the `apply (induct l)` line and the `apply auto` line you can compare the subgoals to see the effect of `apply auto`. In the two subgoals, the expressions built on `remove` have been simplified but not the expressions built on `List.member`. Solving those goals is a job for `sledgehammer`. Type `sledgehammer` below `apply auto`. After some seconds, in the output frame you can see the results obtain by some external automated theorem provers to solve this goal. In particular, you should see this result:

```
Try this: apply (simp add: member_rec(2))
```

by simpliy clicking on the proposed proof command `apply (simp add: member_rec(2))` it is added to your proof. It has solved the first subgoal. Below, for the last goal, you can simply type again `sledgehammer` and click on the proof it provides to complete this second proof. The last proof command should be of the form `by (simp add: member_rec(1))`, where `by` is a proof command that also close the proof mode like `done`. To clean up the proof you can remove the `sledgehammer` commands that are no longer necessary. Let's finish this part by proving a last lemma, where the implication symbol $\longrightarrow$ can be obtained by typing "`-->`":

```
lemma "(length (remove e l)) = (length l)  ⟶ ¬ (List.member l e)"
```

This property means that if the length of `(remove e l)` and `l` is equal, then `e` does not belong to `l`. Nitpick does not find any counterexample. We can start the proof by `apply (induct l)` and `apply auto` as usual. We end up with four subgoals. Sledgehammer can solve the first one, but not the second one. So we are stuck with 3 subgoals. Let's have a more careful look at the first remaining subgoal:

```
1. ⋀l. length (remove e l) = Suc (length l) ⟹ List.member (e # l) e ⟹ False
```

This goal is of the form $A \Longrightarrow B \Longrightarrow False$. In other words, with assumptions A and B we have to prove $False$. The only solution for this subgoal to hold is if the assumptions, separately or together, are false. Here, the first assumption is clearly false: `length (remove e l)` cannot be equal to `(length l)+1` (`Suc i` stands for the successor of natural `i`). In fact, we already know, from the previous lemma how, do `length(remove e l)` and `length l` relate. However, for Sledgehammer to use this lemma, we need to name it. Click back on the definition of the previous lemma and add it a name. For instance:

```
lemma count_remove: "(length l) = (length (remove e l)) + (List.count l e)"
```

Come back to the proof and click on the failing `sledgehammer` line. Now sledgehammer succeeds on this goal and you can see that the found proofs use the `count_remove` lemma. Simlarly Sledgehammer can prove the two remaining subgoals, concluding the proof.

6 Datatypes

Let's define the algebraic datatype of polymorphic binary trees with two constructors `Leaf` and `Node`:

```
datatype 'a tree= Leaf | Node 'a  "'a tree"  "'a tree"
```

You have to pay attention to the double quotes used to avoid ambiguity when giving the type parameters to the `Node` constructor. Let's define an object of this type and associate it to the constant name `myTree`:

```
definition "myTree= Node (1::int) (Node 2 Leaf Leaf) (Node 3 Leaf Leaf)"
```

This object has type `int tree`. Now let's define the following two functions:

```
fun numNodes:: "'a tree ⇒ nat"
where
"numNodes Leaf = 0" |
"numNodes (Node e t1 t2) = 1+(numNodes t1)+(numNodes t2)"

fun subTree:: "'a tree ⇒ 'a tree ⇒ bool"
where
"subTree Leaf  Leaf            = True" |
"subTree t     Leaf            = False" |
"subTree t     (Node e t1 t2)  = ((t=(Node e t1 t2)) ∨ (subTree t t1) ∨ (subTree t t2))"
```

The function `numNodes` counts the number of nodes in a binary tree and the function `subTree` checks if a tree is a subtree of another, where the symbol "∨" (denoting "or") is obtained by typing `\/`. We can test those functions on the constant we just defined:

```
value "numNodes myTree"
value "subTree (Node 3 Leaf Leaf) myTree"
```

Besides, an expected property of those functions is the following lemma stating that if a tree `t1` is a subtree of `t2`, then the number of nodes of `t1` is lesser or equal to the number of nodes of `t2`.

```
lemma subNum: "(subTree t1 t2) ⟶ (numNodes t1) <= (numNodes t2)"
```

To prove this lemma, we need to perform a proof by induction on trees because functions `numNodes` and `subTree` are recursively defined on trees. We have the choice between an induction on `t1` or on `t2`. However, since `subTree` is recursively defined on the second parameter, induction on `t2` is more likely to succeed. The proof can be carried out using `apply (induct t2)`, `apply auto` and `sledgehammer`.

7 Code exportation

You can export the code of Isabelle/HOL functions into various programming languages (SML, Haskell, OCaml and Scala) using the `code_export` directive. For instance, below all the function definitions and outside a proof, you can type: `export_code remove numNodes subTree in Scala` whose effect is to export all the necessary material for the above functions to run in Scala.