Isabelle coursework exercises

John Wickerson

Autumn term 2023

There are five tasks, all worth the same number of marks. They appear in roughly increasing order of difficulty. The tasks are independent from each other, so failure to complete one task should have no bearing on later tasks. Tasks labelled ($\star$) are expected to be reasonably straightforward. Tasks labelled ($\star\star$) should be manageable but may require quite a bit of thinking, and it may be necessary to consult additional sources of information, such as the Isabelle manual and Stack Overflow. Tasks labelled ($\star\star\star$) are more ambitious still.

Marking principles. It is not expected that students will complete all parts of all the tasks. Partial credit will be given to partial answers. If you are unable to complete a proof, partial credit will be given for explaining your thinking process in the form of `(*comments*)` in the Isabelle file.

Submission process. You are expected to produce a single Isabelle theory file called `Surname1Surname2.thy`, where `Surname1` and `Surname2` are the surnames of the two students in the pair. This file should contain all of the definitions and proofs for all of the tasks below that you have attempted.

Plagiarism policy. You **are** allowed to consult the coursework tasks from previous years – the questions and model solutions for these are available. You **are** allowed to consult internet sources like Isabelle tutorials. You **are** allowed to work together with the other student in your pair. You **are** allowed to ask questions on Stack Overflow or the Isabelle mailing list, but make your questions generic (e.g. "Why isn't the `subst` method working as I expected?"); please **don't** ask for solutions to these specific tasks! And please **don't** share your answers to these tasks outside of your own pair.

Task 1 (⋆) This task involves extending our logic synthesiser to handle XOR gates. To do this, we first extend our `circuit` datatype (from the Isabelle worksheet) to include a constructor for XOR gates, as follows (line 5 is new):

```
1 datatype circuit =
2   NOT circuit
3 | AND circuit circuit
4 | OR circuit circuit
5 | XOR circuit circuit
6 | TRUE
7 | FALSE
8 | INPUT ℤ
```

It also involves extending the `simulate` function so that it provides a semantics for this new kind of gate, as follows (lines 6 and 7 are new):

```
1  fun simulate where
2    simulate (AND c1 c2) ρ =
3        ((simulate c1 ρ) ∧ (simulate c2 ρ))
4  | simulate (OR c1 c2) ρ =
5        ((simulate c1 ρ) ∨ (simulate c2 ρ))
6  | simulate (XOR c1 c2) ρ =
7        ((simulate c1 ρ) ≠ (simulate c2 ρ))
8  | simulate (NOT c) ρ = (¬ (simulate c ρ))
9  | simulate TRUE ρ = True
10 | simulate FALSE ρ = False
11 | simulate (INPUT i) ρ = ρ i
```

We are now able to write and simulate circuits that include XOR gates. Next, we shall provide a function that passes over a given circuit and *removes* all the XOR gates that it encounters, replacing them with an equivalent combination of AND, OR, and NOT gates.

```
1  fun elim_xor where
2    "elim_xor (AND c1 c2) =
3           AND (elim_xor c1) (elim_xor c2)"
4  | "elim_xor (OR c1 c2) =
5           OR (elim_xor c1) (elim_xor c2)"
6  | "elim_xor (XOR c1 c2) = (
7           let c1 = elim_xor c1; c2 = elim_xor c2 in
8           AND (OR c1 c2) (NOT (AND c1 c2)))"
9  | "elim_xor (NOT c) = NOT (elim_xor c)"
10 | "elim_xor TRUE = TRUE"
11 | "elim_xor FALSE = FALSE"
```

```
| "elim_xor (INPUT i) = INPUT i"
```

Use Isabelle to prove that `elim_xor` is correct. That is, prove for any circuit c that `elim_xor(`c`)` has the same input/output behaviour as c.

Task 2 ($\star\star$) Continuing with the theme of XOR gates, we shall now provide a function that passes over a given circuit, seeking to *introduce* XOR gates where possible. For instance, if it spots the pattern $(a \vee b) \wedge \neg(a \wedge b)$, then it replaces it with a `XOR` b.

```
fun intro_xor where
  "intro_xor (AND (OR a b) (NOT (AND c d))) = (
   let a = intro_xor a; b = intro_xor b;
       c = intro_xor c; d = intro_xor d in
   if a=c /\ b=d \/ a=d /\ b=c then XOR a b else
   (AND (OR a b) (NOT (AND c d))))"
| "intro_xor (AND (NOT (AND a b)) (OR c d)) = (
   let a = intro_xor a; b = intro_xor b;
       c = intro_xor c; d = intro_xor d in
   if a=c /\ b=d \/ a=d /\ b=c then XOR a b else
   (AND (NOT (AND a b)) (OR c d)))"
| "intro_xor (NOT c) = NOT (intro_xor c)"
| "intro_xor (AND c1 c2) =
           AND (intro_xor c1) (intro_xor c2)"
| "intro_xor (OR c1 c2) =
           OR (intro_xor c1) (intro_xor c2)"
| "intro_xor (XOR c1 c2) =
           XOR (intro_xor c1) (intro_xor c2)"
| "intro_xor TRUE = TRUE"
| "intro_xor FALSE = FALSE"
| "intro_xor (INPUT i) = INPUT i"
```

Use Isabelle to prove that `intro_xor` is correct. That is, prove for any circuit c that `intro_xor(`c`)` has the same input/output behaviour as c. *Note: it doesn't matter if you did not manage to prove the* `elim_xor` *function correct in Task 1, because the* `intro_xor` *function is completely independent from it.*

Task 3 ($\star$) Here is a function that generates a list containing the same repeated element.

```
fun clone :: "nat => 'a => 'a list"
where
  "clone 0 x = []"
| "clone (Suc n) x = x # clone n x"
```

For instance, `clone 5 "Ni"` produces the list `["Ni", "Ni", "Ni", "Ni", "Ni"]`.

Prove using Isabelle that reversing a cloned list has no effect. That is, prove the following lemma.

```
lemma rev_clone: "rev (clone n x) = clone n x"
```

Note: you may find this lemma useful in the next task. So, if you do not manage to complete this task, you are advised to 'prove' `rev_clone` *via* `sorry`*, so that you can rely on it when proving other lemmas and theorems.*

Task 4 (⋆⋆) This task is about an algorithm for performing analogue-to-digital conversion (ADC). The algorithm essentially performs a binary search using a value stored in a 'successive approximation register' (SAR).[1] To see the idea, consider the 4-bit version, which works as follows. The SAR begins with the 'middle' value that it can represent, and the first bit is coloured red to indicate that it is the bit currently being considered.

1	0	0	0

The input to the ADC is a real number i in the range $0 \leq i < 16$. Let us suppose $i = 9.4$ for this example. The SAR is currently representing the number 8, which is less than or equal to 9.4, so we keep the `1` (and colour it green to indicate that it is now fixed) and move on to the second SAR bit, setting it to `1` too.

1	1	0	0

The SAR is now representing the number 12, which is greater than 9.4, so we revert the second SAR bit from `1` back to `0`, and move on to the third SAR bit.

1	0	1	0

The SAR is now representing the number 10, which is greater than 9.4, so we revert the third SAR bit from `1` back to `0`, and move on to the fourth and final SAR bit.

1	0	0	1

[1] https://en.wikipedia.org/wiki/Successive-approximation_ADC

The SAR is now representing the number 9, which is less than or equal to 9.4, so we keep the fourth SAR bit as `1`. The value in the SAR is then presented as the output of the ADC.

1	0	0	1

We shall verify the correctness of this ADC algorithm in Isabelle. First, we need an implementation. The following function performs one 'step' of the ADC.

```
1 fun adc_step :: "real => sar => sar"
2 where
3   "adc_step i (r1, r2) = (
4     let r = r1 @ (True # tl r2) in
5     let cmp = real (bits_to_nat r) <= i in
6     (r1 @ [cmp], tl r2))"
```

It takes a real number `i` as the analogue input to be converted, and the current contents of the SAR. We split the contents of the SAR into two lists, `r1` holding the more significant bits and `r2` holding the less significant bits. At the beginning, `r1` is empty, and then each step moves a bit from `r2` into `r1`, and the algorithm is finished when `r2` is empty.[2] On line 4, the first bit of `r2` is set to `1` and then the two lists are combined to get the current contents of the whole SAR. On line 5, this contents is converted to a natural number and compared against the input `i`. On line 6, the new contents of the SAR is created, with one bit removed from the start of `r2`, and one bit added to the end of `r1` depending on the result of the comparison.

The following function calls `adc_step` repeatedly until `r2` is empty, at which point it returns the contents of `r1`.

```
1 fun adc_helper :: "real => sar => bool list"
2 where
3   "adc_helper i (r1, []) = r1"
4 | "adc_helper i (r1, r2) =
5              adc_helper i (adc_step i (r1, r2))"
```

The following function is the top-level entry point to the ADC. It zeroes the SAR and then calls `adc_helper`. The first parameter, `w`, is the width of the ADC, which can be any natural number. In our worked example above, we had `w` set to 4.

[2] In an imperative language like C, something similar could be accomplished by storing the contents of the SAR in an array and using an integer to track the index of the array element that is currently being considered.

```
definition adc :: "nat => real => bool list"
where
  "adc w i = adc_helper i ([], clone w False)"
```

The following theorem says that if the input i is non-negative, then the output from the ADC, when intepreted as a binary number via the `bits_to_nat` function, will not exceed i, regardless of the bitwidth w.[3]

Theorem 1.

$$0 \leq i \longrightarrow \texttt{bits\_to\_nat}(\texttt{adc}\ w\ i) \leq i$$

Prove Theorem 1 using Isabelle.

Hint: find an invariant for the `adc_helper` *function – a property that holds every time* `adc_helper` *is called. This can be proved by rule induction. Once that proof is complete, it follows that the property will hold when* `adc_helper` *is called for the* first *time (by the top-level* `adc` *function). If your invariant is strong enough, you should be able to then deduce the correctness of* `adc`*.*

Task 5 ($\star\star\star$) This task relates to Fermat's Last Theorem, which states that if a, b, c, and n are positive integers and $n > 2$, then $a^n + b^n \neq c^n$.

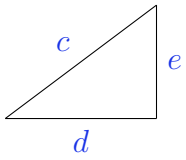
We're not going to prove the whole theorem – proof assistants are still not ready for proofs *that* complex! – instead, we will prove one special case of the theorem: the case when $n = 4$.

Theorem 2 (Fermat's Last Theorem when $n = 4$)**.** *If a, b, and c are positive integers then*

$$a^4 + b^4 \neq c^4.$$

The proof of Theorem 2 rests on another of Fermat's theorems: the so-called 'Fermat's Right Triangle Theorem', which is as follows. Suppose we have an integer-sided right triangle with hypotenuse c and other sides d and e (i.e. $d^2 + e^2 = c^2$ by Pythagoras).

[3] The theorem is, by the way, quite weak, because it only bounds `adc`'s output from above. To capture the `adc`'s behaviour more fully, we would also like to bound it from below. But that property is more subtle, because it only holds if the bitwidth w is large enough. So, we'll leave that for another day.



Then d and e cannot both be squares (i.e. there do not exist integers a and b such that $d = a^2$ and $e = b^2$). This can be stated more concisely as follows:

Theorem 3 (Fermat's Right Triangle Theorem)**.** *If a, b, and c are positive integers then*

$$\gcd(a, b) = 1 \wedge \mathrm{odd}(a) \longrightarrow a^4 + b^4 \neq c^2.$$

Note that the statement above includes two extra assumptions, both of which can be made without loss of generality: (1) that any such triangle is in a 'reduced' form having removed any common factors between a and b (i.e. that the GCD of a and b is 1), and (2) that a is odd (since at least one of a and b must be odd, otherwise they'd have a common factor of 2, it might as well be a).

Your task is to use Isabelle to prove Theorem 2. You may use Theorem 3 without proof.