Isabelle coursework exercises

John Wickerson

Autumn term 2024

There are six tasks. Tasks 1–5 are worth 15 marks each. Task 6 is worth 25 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 (⋆) are expected to be reasonably straightforward. Tasks labelled (⋆⋆) 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 (⋆⋆⋆) 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 your solutions to all of the tasks below that you have attempted. You are welcome to show your working on incomplete tasks by decorating your file with `(*comments*)`. Some of the tasks contain questions that require short written answers; these answers can be provided as comments.

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?");

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

```
1  datatype circuit =
2     NOT circuit
3   | AND circuit circuit
4   | OR circuit circuit
5   | NAND 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 (NAND 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 NAND gates. Next, we shall provide a function that passes over a given circuit and removes all the AND/OR/NOT gates that it encounters, replacing them with an equivalent combination of NAND gates. (This is possible because NAND gates enjoy a property called *functional completeness*.)

```
1  fun intro_nand where
2    "intro_nand (AND c1 c2) =
3          NAND (NAND (intro_nand c1) (intro_nand c2))
4                TRUE"
5  | "intro_nand (OR c1 c2) =
```
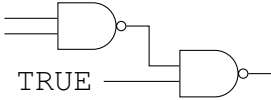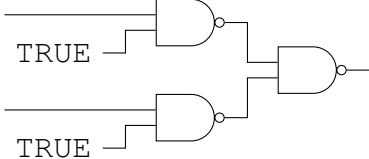
Table 1: Replacing AND and OR gates with NAND gates

```
6            NAND (NAND (intro_nand c1) TRUE)
7                 (NAND (intro_nand c2) TRUE)"
8  | "intro_nand (NAND c1 c2) = (
9            NAND (intro_nand c1) (intro_nand c2))"
10 | "intro_nand (NOT c) = NAND (intro_nand c) TRUE"
11 | "intro_nand TRUE = TRUE"
12 | "intro_nand FALSE = NAND TRUE FALSE"
13 | "intro_nand (INPUT i) = INPUT i"
```

For example, Table 1 shows how AND and OR gates can be replaced with two or three NAND gates. In all cases, we assume a constant `TRUE` is always available.

1. Use Isabelle to prove that `intro_nand` is sound. That is, prove for any circuit c that `intro_nand(`c`)` has the same input/output behaviour as c. Note that there is a (deliberate) bug in the definition above, which you will need to fix before you can prove the correctness theorem.

2. Then use Isabelle to prove that `intro_nand` only produces circuits that only contain NAND gates. That is, prove for any circuit c that `only_nands (intro_nand(`c`))` holds, where `only_nands` is defined below.

```
1 fun only_nands where
2   "only_nands (NAND c1 c2) =
3       (only_nands c1 /\ only_nands c2)"
4 | "only_nands (INPUT _) = True"
5 | "only_nands _ = False"
```

Note that, once again, there is a (deliberate) bug in the definition of `only_nands`, which you will need to fix before you can prove the theorem.

Task 2 ($\star$) Here is a function that converts a number into its list of digits.

3

```
1  fun digits10 :: "nat => nat list"
2  where
3    "digits10 n = (if n < 10 then [n] else
4                     (n mod 10) # digits10 (n div 10))"
```

To simplify the definition, the list of digits is produced in reverse order, least significant digit first. So, for instance `digits10 42` produces the list `[2, 4]`.

Prove using Isabelle that every digit in a list produced by `digits10` is always less than 10.

```
theorem "∀d ∈ set (digits10 n). d < 10"
```

Hint: you may find it helpful to rephrase the statement above as a helper lemma that names the list explicitly, e.g.

```
lemma "ds = digits10 n ==> ∀d ∈ set ds. d < 10"
```

because this allows you to perform induction on the list, `ds`.

Also prove using Isabelle that `digits10` never produces the empty list.

Task 3 (⋆⋆) Here is a function that does the opposite of `digits10`: it converts a list of digits back into a number.

```
1  fun sum10 :: "nat list => nat"
2  where
3    "sum10 [] = 0"
4  | "sum10 (d # ds) = d + 10 * sum10 ds"
```

Prove using Isabelle that whenever you apply `digits10` and then `sum10`, you always get back to the original number; that is, we have

```
sum10 (digits10 n) = n
```

for any number n.

Also use Isabelle to show that applying `sum10` and then `digits10` *doesn't* always get back to the original digit list; that is,

```
digits10 (sum10 ds) = ds
```

is not always true.

*Note: you may find some of the theorems proved in Task 2 helpful. If you were not able to complete Task 2, please use **sorry** to get Isabelle to accept the Task 2 theorems without proof, so that you can use them when attempting Task 3.*

Task 4 (⋆⋆) Prove in Isabelle that any 6-digit number of the form ABABAB is divisible by 37.

Note: you may find the theorem proved in Task 3 helpful. If you were not able to complete Task 3, please use **sorry** *to get Isabelle to accept the Task 3 theorem without proof, so that you can use it when attempting Task 4.*

Task 5 (⋆) This task is about verifying the following naive SAT solver.

```
1  definition naive_solve :: "query => valuation option"
2  where
3    "naive_solve q ==
4    let xs = symbol_list q in
5    let ρs = mk_valuation_list xs in
6    List.fold (until (evaluate q)) ρs None"
```

The code above works as follows, line by line:

1. The input to `naive_solve` is a `query`. A `query` is a list of `clause`s. Each `clause` is a list of `literal`s. Each `literal` is a pair comprising a `symbol` and a `bool`. A `symbol` is implemented simply as `string`. As an example: a query like $(a \lor b) \land (\neg b \lor c)$ can be represented using the `query` datatype as

```
[
  [
    (''a'', True),
    (''b'', True)
  ],
  [
    (''b'', False),
    (''c'', True)
  ]
]
```

The output of `naive_solve` is either `None` (query is unsatisfiable) or `Some` ρ (query is satisfied by `valuation` ρ). A `valuation` is a list of symbols and their truth-values (i.e., a `(symbol * bool) list`). As an example: a valuation like $[a \mapsto true, b \mapsto true, c \mapsto false]$ can be represented using the `valuation` datatype as

```
[(''a'', True), (''b'', True), (''c'', False)]
```

This valuation satisfies the query above.

4. The solver extracts the list of symbols that appear in the query `q`. If a symbol appears more than once in the query, it will only appear once in this list.

5. The solver then calculates *all possible* valuations over these symbols. Every symbol can be assign true or false, so if there are N different symbols in the query, there will be 2^N possible valuations. (This is a remarkably inefficient SAT solver!)

6. The solver iterates through these valuations until it finds one that satisfies the query. If it finds no such valuation, it returns `None`.

Prove using Isabelle that if the solver returns `Some` ρ, then ρ really does satisfy the given query; that is:

```
theorem
  assumes "naive_solve q = Some ρ"
  shows "evaluate q ρ"
```

Also prove using Isabelle that if the solver returns `None`, then none of the valuations it tried satisfied the given query; that is:

```
theorem
  assumes "naive_solve q = None"
  shows "∀ρ ∈ set (mk_valuation_list (symbol_list q)).
           ¬ (evaluate q ρ)"
```

Task 6 (⋆⋆⋆) This task is about verifying a SAT solver that is slightly less naive than the one in Task 5, but still rather simple. Its code is as follows:

```
1  function simp_solve :: "query => valuation option"
2  where
3    "simp_solve q = (
4      case q of
5        [] => Some []
6      | [] # _ => None
7      | ((x,_) # _) # _ => (
8        case simp_solve (update_query x True q) of
9          Some ρ => Some ((x, True) # ρ)
10        | None => (
11          case simp_solve (update_query x False q) of
12            Some ρ => Some ((x, False) # ρ)
13          | None => None)))"
```

It works as follows. Given a query, it does a three-way case split. If the query has no clauses (line 5) then it is trivially satisfiable (with the empty valuation). If the first clause in the query is empty (line 6), then the query is unsatisfiable. Otherwise (line 7), it considers the first symbol that appears in the query, and makes two recursive solving attempts: one with that

symbol evaluated to true (line 8), and one with it evaluated to false (line 11). If neither recursive attempt succeeds (line 13), the query is deemed unsatisfiable.

- Prove using Isabelle that the `simp_solve` function always terminates.
- Prove using Isabelle that if the solver returns `Some` ρ, then ρ really does satisfy the given query.
- Prove using Isabelle that if the solver returns `None`, then no well-formed valuation satisfies the query. (A valuation is 'well-formed' as long as it does not assign a truth-value for the same symbol more than once.)

Hint. Here is a helpful lemma that you may wish to prove.

```
lemma evaluate_update_query:
  assumes "x ∉ domain (ρ)"
  shows "evaluate (update_query x b q) ρ
         = evaluate q ((x, b) # ρ)"
```

It says that if symbol x *is not assigned a truth-value by valuation* ρ*, then updating a query under the valuation* x=>b *is the same as updating* ρ *itself and leaving the query unchanged.*