Isabelle coursework exercises

John Wickerson

Autumn term 2025

There are three tasks, worth a total of 100 marks. The tasks 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 (or parts of tasks) that are labelled (⋆) are expected to be reasonably straightforward; those 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; those 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*)`.

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

[0]Revision history: first version, 23 September 2025.

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! Please **don't** share your answers to these tasks outside of your own pair, but please **do** help each other with general Isabelle tips and tricks – after all, when one student helps another student, *both* learn!

Task 1 (⋆) Recall the datatype we use to represent simple circuits.

```
datatype "circuit" =
  NOT "circuit"
| AND "circuit" "circuit"
| OR "circuit" "circuit"
| TRUE
| FALSE
| INPUT "int"
```

This task is about assessing the computational complexity of some optimisations on these circuits.

Recall the `opt_NOT` function, which optimises a given circuit by removing pairs of consecutive `NOT` gates.

```
fun opt_NOT where
  "opt_NOT (NOT (NOT c)) = opt_NOT c"
| "opt_NOT (NOT c) = NOT (opt_NOT c)"
| "opt_NOT (AND c1 c2) = AND (opt_NOT c1) (opt_NOT c2)"
| "opt_NOT (OR c1 c2) = OR (opt_NOT c1) (opt_NOT c2)"
| "opt_NOT TRUE = TRUE"
| "opt_NOT FALSE = FALSE"
| "opt_NOT (INPUT i) = INPUT i"
```

Consider the following `cost_opt_NOT` function.

```
fun cost_opt_NOT :: "circuit => nat" where
  "cost_opt_NOT (NOT (NOT c)) = 1 + cost_opt_NOT c"
| "cost_opt_NOT (NOT c) = 1 + cost_opt_NOT c"
| "cost_opt_NOT (AND c1 c2) =
    1 + cost_opt_NOT c1 + cost_opt_NOT c2"
| "cost_opt_NOT (OR c1 c2) =
    1 + cost_opt_NOT c1 + cost_opt_NOT c2"
| "cost_opt_NOT TRUE = 1"
| "cost_opt_NOT FALSE = 1"
| "cost_opt_NOT (INPUT _) = 1"
```

Its purpose is to estimate the computational 'cost' of running `opt_NOT` on a given circuit. Roughly speaking, it follows the same recursive structure as `opt_NOT`, and assigns a cost of 1 unit for each line of code.

Prove using Isabelle that the cost of `opt_NOT` on a given circuit is $O(n)$, where n is the number of gates in the circuit. More precisely, prove that there exist constants a and b such that the cost of `opt_NOT` on a circuit with n gates never exceeds $an + b$. [10 marks]

```
theorem opt_NOT_linear:
  "∃ a b ::nat. cost_opt_NOT c ≤ a * area c + b"
```

The definition of `area` is as follows.

```
fun area :: "circuit => nat" where
  "area (NOT c) = 1 + area c"
| "area (AND c1 c2) = 1 + area c1 + area c2"
| "area (OR c1 c2) = 1 + area c1 + area c2"
| "area _ = 0"
```

Now consider the `factorise` function, which was introduced in the 2021 coursework. This optimisation exploits identities like

$$(a \vee b) \wedge (a \vee c) = a \vee (b \wedge c)$$

in order to remove some gates.

```
fun factorise :: "circuit => circuit" where
  "factorise (NOT c) = NOT (factorise c)"
| "factorise (AND (OR c1 c2) (OR c3 c4)) = (
    let c1' = factorise c1; c2' = factorise c2;
        c3' = factorise c3; c4' = factorise c4 in
    if c1' = c3' then OR c1' (AND c2' c4')
    else if c1' = c4' then OR c1' (AND c2' c3')
    else if c2' = c3' then OR c2' (AND c1' c4')
    else if c2' = c4' then OR c2' (AND c1' c3')
    else AND (OR c1' c2') (OR c3' c4'))"
| "factorise (AND c1 c2) =
     AND (factorise c1) (factorise c2)"
| "factorise (OR c1 c2) =
     OR (factorise c1) (factorise c2)"
| "factorise TRUE = TRUE"
| "factorise FALSE = FALSE"
| "factorise (INPUT i) = INPUT i"
```

The following `cost_factorise` function estimates the cost of running `factorise`. It follows the same recursive structure as `factorise`, and roughly assigns a cost of 1 unit per line of code.

```
fun cost_factorise :: "circuit => nat" where
  "cost_factorise (NOT c) = 1 + cost_factorise c"
| "cost_factorise (AND (OR c1 c2) (OR c3 c4)) =
     4 + cost_factorise c1 + cost_factorise c2 +
     cost_factorise c3 + cost_factorise c4"
| "cost_factorise (AND c1 c2) =
     1 + cost_factorise c1 + cost_factorise c2"
| "cost_factorise (OR c1 c2) =
     1 + cost_factorise c1 + cost_factorise c2"
| "cost_factorise TRUE = 1"
| "cost_factorise FALSE = 1"
| "cost_factorise (INPUT i) = 1"
```

Prove using Isabelle that the cost of `opt_factorise` on a given circuit is $O(n)$, where n is the number of gates in the circuit. More precisely, prove that there exist constants a and b such that the cost of `opt_factorise` on a circuit with n gates never exceeds $an + b$. [10 marks]

```
theorem factorise_linear:
  "∃ a b ::nat. cost_factorise c ≤ a * area c + b"
```

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

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

The list of digits is assumed to be in reverse order, least significant digit first. For instance, `sum10 [2,4]` produces the number `42`.

The definition of `sum10` works by successively peeling elements off the *head* of the list until the list is empty.

The `sum10` function could also be obtained by successively peeling elements off the *end* of the list. Write a lemma that captures this, and prove it using Isabelle. The lemma should be of the following form (you need to replace the '...' with something to make it work). [10 marks]

```
lemma sum10_snoc: "sum10 (ds @ [d]) = ... sum10 ds ..."
```

Now, let us say that a list of digits is a *palindrome* if reversing it yields the same list. For instance, `[8,5,8]` is a palindrome.

Prove using Isabelle that if `ds` is a palindrome of even length, then `sum10 ds` is divisible by 11. [15 marks]

Task 3 (⋆)–(⋆⋆⋆) This task is about SAT solving. SAT queries contain symbols, which we shall represent as natural numbers.

```
type_synonym symbol = "nat"
```

A *literal* in a SAT query is either a symbol or a negated symbol. We shall represent a literal as a pair, where the first element is the symbol, and the second element is a Boolean value, with `true` meaning that the symbol is not negated, and `false` meaning that the symbol is negated.

```
type_synonym literal = "symbol * bool"
```

A *clause* in a SAT query is a disjunction of literals. We shall represent clauses as lists of literals.

```
type_synonym clause = "literal list"
```

A SAT query is a conjunction of clauses. We shall represent queries as lists of clauses.

```
type_synonym query = "clause list"
```

For instance, the query

$$(x_0 \vee x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2)$$

would be represented by the term

```
[[(0,True), (1,True), (2,True), (3,False)]
          [(1,False), (2,True)]].
```

A SAT query may be *satisfied* by a *valuation*. A valuation is a function that assigns each symbol a Boolean value. For instance, the function that maps every symbol to `False` is one valuation that satisfies the query above. If a query can be satisfied by at least one valuation, we say it is *satisfiable*; otherwise, we say it is *unsatisfiable*.

Now, any SAT query can be translated into a 3SAT query. A 3SAT query is a special kind of SAT query where each clause contains no more than three literals. (3SAT queries play an important role in complexity theory.) For instance, the query above can be translated into the following 3SAT query:

$$(x_4 \vee x_0 \vee x_1) \wedge (\neg x_4 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2)$$

The 3SAT query is similar to the original SAT query, but it uses an additional symbol, x_4. The new query is not *equivalent* to the original one, because there are valuations that satisfy the original query but do not satisfy the new one (one example: the 'map every symbol to `False`' valuation mentioned above). However, we can say that the two queries are *equisatisfiable*, which means that if one is satisfiable then so is the other (you just might need two different valuations).

Here is a function that takes a SAT clause and translates (or 'reduces') it to an equisatisfiable 3SAT query.

```
fun reduce_clause ::
  "symbol => clause => (symbol * query)"
where
  "reduce_clause x (l1 # l2 # l3 # l4 # c) =
  (let (x',cs) =
    reduce_clause (x+1) ((x, False) # l3 # l4 # c) in
  (x', [[(x, True), l1, l2]] @ cs))"
| "reduce_clause x c = (x, [c])"
```

Here is a function that takes a SAT query and reduces it to an equisatisfiable 3SAT query.

```
fun reduce :: "symbol => query => query"
where
  "reduce _ [] = []"
| "reduce x (c # q) =
  (let (x',cs) =
    reduce_clause x c in cs @ reduce x' q)"
```

The `reduce` function takes an additional parameter called `x` – this parameter tells the function which is the next symbol to use whenever a fresh symbol is needed (such as x_4 in the example above). Whenever the `reduce` function is called, it is expected that this parameter will be set to a value that is higher than all the symbols that already appear in the query.

1. ($\star$). Prove using Isabelle that the `reduce` function really does produce queries in 3SAT form – that is, no clause in `reduce x q` contains more than three literals. [10 marks]
2. ($\star\star$). Prove using Isabelle that the `reduce` function never *decreases* the number of clauses – that is, the number of clauses in `reduce x q` is never less than the number of clauses in `q`. [10 marks]
3. ($\star\star$). Prove that if `reduce x q` is satisfiable, then `q` is also satisfiable. [15 marks]

4. ($\star\star\star$). Prove that if `q` is satisfiable, and every symbol in `q` is below `x`, then `reduce x q` is also satisfiable. [20 marks]