COMP4161 T3/2024
Advanced Topics in Software Verification

Assignment 2

This assignment is released on October 11th, and is due on November 1st 17:59:59. We will accept Isabelle theory (.thy) files only. You are allowed to make late submissions up to five days (120 hours) after the deadline, but at a cost: -5 marks per day.

The assignment is take-home. This does NOT mean you can work in groups. Each submission is personal. For more information, see the plagiarism policy: https://student.unsw.edu.au/plagiarism

Submit using `give` on a CSE machine:

```
give cs4161 a2 files ...
```

For example:

```
give cs4161 a2 a2.thy
```

For this assignment, all proof methods and proof automation available in the standard Isabelle distribution is allowed. This includes, but is not limited to, `simp`, `auto`, `blast`, `force`, and `fastforce`.

However, if you're going for full marks, you shouldn't use "proof" methods that bypass the inference kernel, such as `sorry`. We *may* award partial marks for plausible proof sketches where some subgoals or lemmas are sorried.

If you use `sledgehammer`, it's important to understand that the proofs suggested by `sledgehammer` are just suggestions, and aren't guaranteed to work. Make sure that the proof suggested by `sledgehammer` actually terminates on your machine (assuming an average spec machine). If not, you can try to reconstruct the proof yourself based on the output, or apply a few manual steps to make the subgoal smaller before using `sledgehammer`.

Note: this document contains explanations of the problems and your assignment tasks. The full set of definitions can be found in the associated Isabelle theory files (this document may not contain all the definitions). You are not allowed to modify the various definitions and the lemma names and their statements provided in the Isabelle theory files, unless you are instructed to do so. You are however allowed to add appropriate attributes to those lemmas to facilitate automation as you need. You are also allowed to prove your own additional lemmas and use them in your solutions to the assignment questions.

Hint: make sure that you read through the hints provided at the end of this document.

1 Block Representation of Binary Numbers (12 marks)

In this assignment, we consider one representation of binary numbers where the bits in a number is segmented into alternating sequences (or *blocks*) of 1s or 0s. For instance, "1110011" consists of "three 1s, two 0s, and two 1s". Formally, we define such blocks as follows:

datatype *block* =
  Zeros nat | *Ones nat*

A list of such *block* represents a binary number if it satifies the following well-formedness considerations:

- number 0 should be represented only by an empty list;

- each block represents non-empty sequence of 0s or 1s;
- blocks of 0s and 1s should alternate in a list.

The first point means that lists such as [*Zeros 3*] or [*Ones 0*] are not well-formed. The second point means that $0 < i$ should hold for any block *Zeros i* and *Ones i*. The third point says that lists that have consecutive blocks of 1s or 0s, eg. [*Ones i*, *Ones j*, ...], are ill-formed. We name the type of our binary number representation as *bnat*. The inductive predicate *wf* gives the well-formedness conditions as discussed above:

```
type-synonym bnat = block list
inductive wf :: bnat ⇒ bool where
   wf-nil: wf []
 | wf-ones: i > 0 ⟹ wf [Ones i]
 | wf-zeros: i > 0 ⟹ wf blks ⟹ is1hd blks ⟹ wf (Zeros i # blks)
 | wf-ones2: i > 0 ⟹ wf blks ⟹ is0hd blks ⟹ wf (Ones i # blks)
```

where each of the *primrec* functions *is1hd* and *is0hd* tests if the head of the given list is a 1-block or 0-block, respectively.

We then define a function *to-nat* that computes the value as natural number (converts a *bnat* number to a natural number). Note that we interpret the head of a list as the least significant (block of) digits of the number.

```
primrec to-nat :: bnat ⇒ nat where
  to-nat [] = 0
| to-nat (b # blks) =
    (case b of
      Zeros i ⇒ 2^i * (to-nat blks)
    |  Ones i ⇒ 2^i - 1 + 2^i * (to-nat blks))
```

(a) Prove simplification rules for *is0hd* and list append (@) (1 marks):

$$is0hd\ (x\ @\ y) = (if\ x = []\ then\ is0hd\ y\ else\ is0hd\ x)$$

(b) State and prove similar simplification rules for *is1hd* and list append (@). (2 marks)

(c) Prove that *is1hd n* and *is0hd n* cannot be the same for any *n*. (2 marks)

(d) Prove that, for any well-formed *n*, the value 0 is represented by [] and nothing else. (4 marks)

(e) Prove that, for any well-formed *n*, if *is1hd n* holds, the value of *n* must be positive. (3 marks)

2 Successor and Predecessor (40 marks)

Next, we define a successor function (*succ*) and a predecessor function (*pred*) for *bnat*. In order to do so, we need to define functions *add0s* and *add1s* that add a block of 0s or 1s, respectively, at the head of a *bnat* number in a *correct* way, so that, when applied to a well-formed *bnat* number, those functions will return a well-formed *bnat* number.

```
primrec add-block0 :: nat ⇒ block ⇒ block list where
  add-block0 i (Zeros j) = [Zeros (i+j)]
| add-block0 i (Ones j) = (if i = 0 then [Ones j]
                           else [Zeros i,Ones j])

primrec add0s :: nat ⇒ bnat ⇒ bnat where
  add0s i [] = []
| add0s i (b # blks) = (add-block0 i b) @ blks
```

(a) Prove simplification rules for applying *add0s* twice in a row. (2 marks)

(b) Prove that *add0s* and *add1s* return well-formed *bnat* numbers. (4 marks)

(c) Prove the correctness of *add0s*, i.e., the *bnat* numbers it return have correct values as natural numbers. (4 marks)

(d) Prove the correctness of *add1s*, similarly as above. (5 marks)

Now we can define *succ* and *pred* as follows:

```
primrec succ :: bnat ⇒ bnat where
  succ [] = [Ones 1]
| succ (b # blks) =
    (case b of
        Zeros i ⇒ add1s 1 (add0s (i−1) blks)
      | Ones i ⇒ Zeros i # succ blks)

primrec pred :: bnat ⇒ bnat where
  pred [] = []
| pred (b # blks) =
    (case b of
        Ones i ⇒ add0s 1 (add1s (i−1) blks)
      | Zeros i ⇒ Ones i # pred blks)
```

(e) Prove that *succ* will turn a *is0hd* list into a *is1hd* list and vice versa. (4 marks)

(f) Prove that *succ* preserves the well-formedness conditions. (3 marks)

(g) Prove the correctness of *succ*. (4 marks)

(h) Prove that *pred* preserves the well-formedness conditions. (4 marks)

(i) Prove the correctness of *pred*. (6 marks)

(j) Prove *pred* (*succ n*) = *n* for a well-formed *n*. (4 marks)

3 Conversion from Natural Numbers (30 marks)

We now consider converting natural numbers into our block binary representation. The function *from-nat* does this conversion using the function *succ* that we defined in the previous section.

Throughout this assignment it may be helpful to state your own intermediate lemmas. This is especially the case for parts (c), (d) and (e) of this question.

```
primrec from-nat :: nat ⇒ bnat where
  from-nat 0 = []
| from-nat (Suc n) = succ (from-nat n)
```

(a) Prove that *from-nat* generates a well-formed *bnat* number. (2 marks)

(b) Prove that *to-nat* is the inverse of *from-nat*. (2 marks)

(c) Prove that adding a 0-block (*add0s i* (*from-nat n*)) gives the representation of the number $2^i * n$. (8 marks)

(d) Prove that a single element 1-block [*Ones i*] represents the natural number $2^i - 1$ assuming $i > 0$. (8 marks)

(e) Prove that *from-nat* is the inverse of *to-nat* for well-formed *bnat* numbers. (10 marks)

4 Addition and Multiplication (18 marks)

Here we consider addition and multiplication of our block binary representation. The *simple-add* below defines such addition that uses an auxiliary worker function with an extra variable.

```
primrec simple-add-worker :: nat ⇒ bnat ⇒ bnat ⇒ bnat
  where
    simple-add-worker 0 i j = []
  | simple-add-worker (Suc fuel) i j =
    (if i = [] then j
      else simple-add-worker fuel (pred i) (succ j))

definition simple-add :: bnat ⇒ bnat ⇒ bnat
  where simple-add i j = simple-add-worker (to-nat i + 2) i j
```

(a) Prove that *simple-add-worker* return a well-formed *bnat* number if its last argument is well-formed. (3 marks)

(b) Prove the correctness of *simple-add-worker*. (3 marks)

(c) Prove the correctness of *simple-add*. (2 marks)

Now, we can define a multiplication function, using a similar auxiliary function that uses *simple-add*.

(d) Complete the definition of *simple-mul-worker* using *simple-add*. (3 marks)

(e) Complete the definition of *simple-mul*. (2 marks)

(f) Prove the correctness of *simple-mul-worker*. (3 marks)

(g) Prove the correctness of *simple-mul*. (2 marks)

5 Hints

- Many proofs will require induction of one kind or the other. Other than inducting on datatypes directly, you may find it useful to do induction on inductively defined relations *wf*. The induction rules for these are automatically generated by Isabelle.

  You can apply these induction rules as elimination rules, e.g. `apply (erule wf.induct)`, but a more convenient and flexible alternative is

  ```
  apply (induct rule: wf.induct)
  ```

  which allows you to specify which variables should not be all-eliminated using e.g.

  ```
  apply (induct arbitrary: x y rule: wf.induct)
  ```

- Not everything needs an induction.
- The equivalent of `spec` for the meta-logic universal quantifier, if you need it, is called `meta_spec`.
- For some exercises, you will likely need additional lemmas to make the proof go through. Part of the assignment is figuring out which lemmas are needed.
- Make use of the `find_theorems` command to find library theorems. You are allowed to use all theorems proved in the Isabelle distribution.