COMP4161 T3/2020
Advanced Topics in Software Verification

Assignment 2

This assignment starts on Friday 16th October 2020 and is due on Friday 30th October 2020 6pm. We will accept Isabelle theory (.thy) files.

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
```

Any proof method or proof automation not explicitly disallowed is allowed. This includes, but is not limited to, `simp`, `auto`, `blast`, `force`, and `fastforce`.

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 file.

Hint: there are hints at the end of this document.

1 Expressions and Types (28 marks)

For the first question, we define a type system for a simple language of arithmetic and boolean expressions. The expressions are built from integer and boolean literals, variables, as well as three binary operators: addition (*Add*), conjunction (*And*) and less-than or equal (*Le*).

datatype *exp* =
  Bool bool
| *Integer int*
| *Var string*
| *Add exp exp*
| *And exp exp*
| *Le exp exp*

A practical programming language would have more operators. This expression language however provides a reasonable core for an assignment and

demonstrates proof concepts while keeping the required proofs relatively short.

Evaluating an expression, as represented by the function *eval*, either produces *None* which represents a failed evaluation, or it produces a *value*:

datatype *val* =
  Boolv bool | *Intv int*

Note that the evaluation function fails whenever it encounters a free variable. This is because the semantics we are building towards is substitution-based, so in practice, any variable that occurs in an expression will have been substituted for its value before the expression is evaluated. The function *subst-exp* substitutes a variable for a value in an expression and the function *vars* returns the set of variables that occur inside an expression.

Prove the following:

(a) $x \notin \mathit{vars}\ e \Longrightarrow \mathit{subst\text{-}exp}\ e\ x\ d = e$ (1 mark)

(b) $x \notin \mathit{vars}\ (\mathit{subst\text{-}exp}\ e\ x\ v)$ (2 marks)

(c) $x \neq y \Longrightarrow \mathit{subst\text{-}exp}\ (\mathit{subst\text{-}exp}\ e\ x\ v)\ y\ v' = \mathit{subst\text{-}exp}\ (\mathit{subst\text{-}exp}\ e\ y\ v')\ x\ v$ (2 marks)

(d) $\mathit{subst\text{-}exp}\ (\mathit{subst\text{-}exp}\ e\ x\ v)\ x\ v' = \mathit{subst\text{-}exp}\ e\ x\ v$ (2 marks)

Our type system for expressions needs only two types: booleans and integers.

datatype *vtype* = *BoolT* | *IntT*

The type of an expression is defined using the inductive relation *type-exp*. The intuition is that if *type-exp* Γ *exp ty* holds, then *exp* has type *ty* in the typing environment Γ. Typing environments record the types of variables. They are partial functions from variable names to types *string* $\rightharpoonup$ *vtype*, which in Isabelle/HOL is encoded as functions from *string* to *vtype option*.

The point of the type system is to guarantee that every well-typed (ground) expression evaluates to a value of the right type.

Prove the following properties about the type system:

(e) $[\![\mathit{type\text{-}exp}\ \Gamma\ e\ ty;\ \Gamma\ x = \mathit{Some}\ (\mathit{type\text{-}v}\ v)]\!] \Longrightarrow \mathit{type\text{-}exp}\ \Gamma\ (\mathit{subst\text{-}exp}\ e\ x\ v)\ ty$ (3 marks)

(f) $[\![\mathit{type\text{-}exp}\ \Gamma\ e\ ty;\ x \notin \mathit{vars}\ e]\!] \Longrightarrow \mathit{type\text{-}exp}\ (\Gamma(x := t))\ e\ ty$ (2 marks)

(g) $\mathit{type\text{-}exp}\ (\Gamma(x \mapsto \mathit{type\text{-}v}\ v))\ e\ ty \Longrightarrow \mathit{type\text{-}exp}\ \Gamma\ (\mathit{subst\text{-}exp}\ e\ x\ v)\ ty$ (6 marks)

(h) $\mathit{type\text{-}exp}\ \mathit{Map.empty}\ e\ ty \Longrightarrow \exists\, v.\ \mathit{eval}\ e = \mathit{Some}\ v \wedge \mathit{type\text{-}v}\ v = ty$ (6 marks)

(i) Is it the case that every expression that evaluates successfully has a type? I.e., is the following true?

$eval\ e = Some\ v \Longrightarrow type\text{-}exp\ Map.empty\ e\ (type\text{-}v\ v)$

If you answer yes, explain why (informally). If you answer no, give a counterexample. (4 marks)

2 Processes and Session Types (29 marks)

For this question, we will develop a simple process description language. Processes can compute using the expression language from the previous question, and they can send and receive synchronous messages to each other:

datatype *process* =
  Done
  | *Send exp process*
  | *Receive string process*
  | *IfThen exp process process*
  | *ExtChoice process process*

The *Done* process does nothing, and represents successful termination. The *Send e p* process sends the value of the expression e, and then continues as p; if e doesn't evaluate to a value, the process is stuck. The *Receive v p* process waits to receive a value and then continue as p. Any occurence of the placeholder variable v in p will be replaced by the received value before execution continues. As you might expect, *IfThen* branches on the value of a boolean expression, but that's not all: once a branch has been chosen, we assume that the process sends a signal to inform other processes which branch was taken. The *ExtChoice* (external choice) process waits to receive such a branch-selection signal, so that it can synchronise its choice of branch with another process executing an *IfThen*.

We use pairs $(p,\ q)$ to represent two processes p and q executing in parallel. The inductive transition relation *semantics* formalises the above intuition about how processes should behave. Informally, $semantics\ (p,\ q)\ (r,\ s)$ means that p and q can synchronise to perform an action together, and that after this action, the processes can continue as r and s.

The semantics is defined through four rules *com-l*, *com-r*, *choice-l*, and *choice-r*. The *com-l* rule states that given a pair of processes, if the left process sends, the right process receives, and the sent expression evaluates successfully to a value, then a sender and a receiver may synchronise, causing the sent value to be substituted into the receiving process:

$$\frac{eval\ exp = Some\ d \qquad q' = subst\text{-}proc\ q\ x\ d}{semantics\ (Send\ exp\ p,\ Receive\ x\ q)\ (p,\ q')}$$

The *choice-l* rule allows a process executing an *IfThen* statement to synchronise its choice of branch with another process executing an *ExtChoice*. This is only possible if the branch-selecting evaluates to a boolean value.

$$\frac{\begin{array}{c} eval\ exp = Some\ (Boolv\ b) \\ p' = (\mathit{if}\ b\ \mathit{then}\ p\ \mathit{else}\ q) \qquad q' = (\mathit{if}\ b\ \mathit{then}\ r\ \mathit{else}\ s) \end{array}}{semantics\ (IfThen\ exp\ p\ q,\ ExtChoice\ r\ s)\ (p',\ q')}$$

The *com-r* and *choice-r* rules are the symmetric versions of the *com-l* and *choice-l* rules presented above, with the roles of the two processes reversed.

(a) Consider a pair of processes: one that sends the integer 5 and then successfully terminates, and another that receives a message and then successfully terminates. Formulate and prove a lemma stating that the *semantics* relates this pair of processes to a pair of successfully terminated processes. (2 marks)

(b) Consider another pair of processes: one waits for an external choice, and then terminates regardless of branch choice. Another chooses a branch based on the value of a boolean expression, then terminates regardless of branch choice. Formulate and prove a lemma stating that the *semantics* relates this pair of processes to a pair of successfully terminated processes. (2 marks)

Now, we will define a type system that can guarantee the absence of deadlocks in a parallel composition of two processes. For the purposes of this assignment, a deadlock occurs when at least one process wants to perform an action, but no transition is possible according to *semantics*.

A *session type* describes the communication behaviour of a process, but abstracts from the concrete values being communicated:

```
datatype stype =
  DoneT
  | SendT vtype stype
  | ReceiveT vtype stype
  | IntChoiceT stype stype
  | ExtChoiceT stype stype
```

For example, the type *SendT BoolT DoneT* is the type of a process that will send a boolean value and then terminate.

The typing judgement for processes (*type-proc*) builds on the type system for expressions. Given a typing environment Γ, a process p and a type t, it determines if p is well-typed under Γ and t. The function *varsp* returns the set of variables that occur in a process.

To test our understanding, we will first prove some sanity checks on our type system so far:

(c) Formulate and prove a lemma listing a type and typing environment under which the process that sends the integer 5 then directly terminates is well-typed through the *type-proc* typing relation. (2 marks)

(d) $\mathit{type\text{-}proc}\ (\Gamma(x \mapsto \mathit{type\text{-}v}\ v))\ e\ ty \Longrightarrow \mathit{type\text{-}proc}\ \Gamma\ (\mathit{subst\text{-}proc}\ e\ x\ v)\ ty$ (5 marks)

(e) $[\![\mathit{type\text{-}proc}\ \Gamma\ p\ ty;\ x \notin \mathit{varsp}\ p]\!] \Longrightarrow \mathit{type\text{-}proc}\ (\Gamma(x := t))\ p\ ty$ (3 marks)

The *dual* of a session types is obtained by swapping all sends for receives, and swapping all internal choices for external choices. The key idea is that the dual is your ideal communication partner: when interacting with its dual, a well-typed process can always make progress, that is, it can always perform a step using the *semantics* relation to a pair of well-typed processes.

Prove the following:

(f) $\mathit{dual}\ (\mathit{dual}\ t) = t$ (1 mark)

(g) $[\![\mathit{type\text{-}proc}\ \mathit{Map.empty}\ p\ ty;\ \mathit{type\text{-}proc}\ \mathit{Map.empty}\ q\ (\mathit{dual}\ ty);\ p \neq \mathit{Done}]\!] \Longrightarrow \exists\, r\ s.\ \mathit{semantics}\ (p,\ q)\ (r,\ s)$ (6 marks)

(h) $[\![\mathit{semantics}\ (p,\ q)\ (r,\ s);\ \mathit{type\text{-}proc}\ \mathit{Map.empty}\ p\ ty;\ \mathit{type\text{-}proc}\ \mathit{Map.empty}\ q\ (\mathit{dual}\ ty)]\!] \Longrightarrow \exists\, ty'.\ ty' \in \mathit{reduce}\ ty \wedge \mathit{type\text{-}proc}\ \mathit{Map.empty}\ r\ ty' \wedge \mathit{type\text{-}proc}\ \mathit{Map.empty}\ s\ (\mathit{dual}\ ty')$ (8 marks)

3 Compiling Internal Choice (43 marks)

The *IfThen* primitive used in the previous section had an unusual extra feature: it also communicated the choice of branch to another process. This is a convenience to make processes easier to type-check, but it doesn't add any expressiveness to the language.

In this section, we will prove this by compiling our process description language into a new language where *IfThen* only has the usual local behaviour, i.e. selects a branch and makes no effort to communicate this fact to the environment. The new process description language will reuse the same syntax developed previously, but will replace the *semantics* relation with a new relation $\mathit{semantics}'$ that treats *IfThen* differently. The key rule in the new language is the following:

$$\frac{\mathit{eval}\ \mathit{exp} = \mathit{Some}\ (\mathit{Boolv}\ b)}{\mathit{semantics}'\ (\mathit{IfThen}\ \mathit{exp}\ p\ q,\ r)\ (\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q,\ r)}$$

The function *compile* introduces a protocol that implements the implicit synchronisation of *IfThen* with explicit communication:

$$compile\ (IfThen\ s\ p\ q) = IfThen\ s\ (Send\ (Bool\ True)\ (compile\ p))\ (Send\ (Bool\ False)\ (compile\ q))$$

and

$$compile\ (ExtChoice\ p\ q) = Receive\ ''x''\ (IfThen\ (Var\ ''x'')\ (compile\ p)\ (compile\ q))$$

We will prove that this compilation scheme preserves the behaviour of the source language in a precise sense: for every reduction using *semantics* (i.e., every two pairs of terms (p, q) and (r, s) that are related by *semantics*), the compiled processes can perform a sequence of reductions using *semantics'* that reach the compiled version of the processes that the source-language reduces to (i.e. (*compile p*, *compile q*) and (*compile r*, *compile s*) are related by the reflexive and transitive closure of *semantics'*).
Prove the following:

(a) $[\![semantics\ (p, q)\ (r, s);\ ''x'' \notin varsp\ p;\ ''x'' \notin varsp\ q]\!] \Longrightarrow semantics'^{**}\ (compile\ p, compile\ q)\ (compile\ r, compile\ s)$ (31 marks)

(b) Explain why the side-conditions $''x'' \notin varsp\ p$ on the aforementioned theorem are necessary. (2 marks)

(c) What changes to the *compile* function would be necessary to make the theorem hold without the side condition? Define an alternative *compile'*. (6 marks)

(d) Explain why it solves the problem. You do not have to prove it correct. (4 marks)

4 Hints

- You are allowed—encouraged, in fact—to use solutions to questions as lemmas in the answers to other questions.
- 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 such as *type-exp* and *type-proc*. The induction rules for these relations are automatically generated by Isabelle.

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

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

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

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

- The assumptions of theorems stated in the `assumes-show` format are accessible via the fact named `assms`. For example, you can do `simp add: assms` or `rule assms(1)`. The assumptions can be added directly to the goal state by beginning your proof with the command `insert assms` or by typing `using assms`; that is usually what you want to do before starting an induction.
- The equivalent of `spec` for the meta-logic universal quantifier is called `meta_spec`.
- Some inductive proofs will require you to strengthen the induction hypothesis in order to close the proof. Think about which variables should be `arbitrary`.
- For the later 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.
- This assignment uses a number of functions and relations from the Isabelle distribution that have not been used in the lectures, for example:
  - `Map.empty`, a partial function with empty domain.
  - `Fun.fun_upd`, written $f(x := y)$. For partial functions, $f(x \mapsto y)$ abbreviates *fun-upd* $f\ x\ (Some\ y)$
  - `rtranclp`, written R^{**}, the reflexive and transitive closure of a relation.

  To learn more about these, you can ctrl-click on the constant names in Isabelle/jEdit to see their definitions, or you can search for theorems about them using e.g. `find_theorems rtranclp`.