The University of New South Wales

COMP3161/9161
Concepts of Programming Languages

Sample Final Exam

Semester 2, 2018

Time Allowed: 2 Hours

Total Number of Questions: 4

Answer **all** questions.

The questions **are** of equal value.

You are permitted **two** hand-written, single-sided A4 sheets of notes.

Only write your answers on the provided booklets.

Answers must be written in ink, with the exception of diagrams.

Drawing instruments or rules may be used.

Excessively verbose answers may lose marks.

There is a 3% penalty if your name and student number are not filled in correctly.

Question 1 (25 marks)

Consider the following inductive definition of a language of restricted boolean expressions:

$$\frac{}{\texttt{true}\ \textbf{Bool}} \qquad \frac{}{\texttt{false}\ \textbf{Bool}} \qquad \frac{b\ \textbf{Bool}}{\texttt{not}\ b\ \textbf{Bool}} \qquad \frac{b_1\ \textbf{Bool} \quad b_2\ \textbf{Bool}}{(\texttt{and}\ b_1\ b_2)\ \textbf{Bool}}$$

It has the following small-step semantics:

$$\frac{b \mapsto b'}{\texttt{not}\ b \mapsto \texttt{not}\ b'}1 \qquad \frac{}{\texttt{not true} \mapsto \texttt{false}}2 \qquad \frac{}{\texttt{not false} \mapsto \texttt{true}}3$$

$$\frac{b_1 \mapsto b_1'}{(\texttt{and}\ b_1\ b_2) \mapsto (\texttt{and}\ b_1'\ b_2)}4 \qquad \frac{}{(\texttt{and true}\ b_2) \mapsto b_2}5 \qquad \frac{}{(\texttt{and false}\ b_2) \mapsto \texttt{false}}6$$

(a) (3 marks) Derive the step-by-step evaluation of (`and not false (and true not true)`).

Solution:

$$\begin{array}{lll}
 & (\texttt{and not false (and true not true)}) & (4,3) \\
\mapsto & (\texttt{and true (and true not true)}) & (5) \\
\mapsto & (\texttt{and true not true}) & (5) \\
\mapsto & \texttt{not true} & (2) \\
\mapsto & \texttt{false} &
\end{array}$$

(b) (4 marks) Are the rules that define **Bool** unambiguous? If so, briefly explain why. If not, give an example of a term that allows for multiple derivations.

Solution: They are **unambiguous** as there is no term that allows for multiple derivations. This is essentially because the `and` expressions are parenthesised.

(c) (6 marks) The semantics rules listed above are small-step. Give an equivalent big-step semantics for this language.

Solution: Let E be the language **Bool** and V be the set of truth values $\{\top, \bot\}$.

$$\frac{}{\texttt{true} \Downarrow \top} \qquad \frac{}{\texttt{false} \Downarrow \bot}$$

$$\frac{b \Downarrow \bot}{\texttt{not}\ b \Downarrow \top} \qquad \frac{b \Downarrow \top}{\texttt{not}\ b \Downarrow \bot}$$

$$\frac{b_1 \Downarrow \top \quad b_2 \Downarrow v_2}{(\texttt{and}\ b_1\ b_2) \Downarrow v_2} \qquad \frac{b_1 \Downarrow \bot}{(\texttt{and}\ b_1\ b_2) \Downarrow \bot}$$

(d) (12 marks) Give a new version of the small-step semantics where *all rules are axioms.* It may be helpful to expand the state with supporting data structures, similarly to the C-Machine. Give an inductive definition for any notation you define.

Solution: Define a frame f as follows, where e denotes an expression:

$$\begin{array}{rcl}
f & ::= & \texttt{not}\ \square \\
 & | & (\texttt{and}\ \square\ e)
\end{array}$$

Define a stack s as a list of frames:

$$\begin{array}{rcl}
s & ::= & \circ \\
 & | & f \triangleright s
\end{array}$$

Then a machine state is either evaluating an expression, written $s \succ e$, or returning a value, written $e \prec v$.

Now our semantics are:

$$\frac{}{s \succ \texttt{not}\ e \quad \mapsto \quad \texttt{not}\ \Box \triangleright s \succ e}$$

$$\frac{}{\texttt{not}\ \Box \triangleright s \prec \top \quad \mapsto \quad s \prec \bot}$$

$$\frac{}{\texttt{not}\ \Box \triangleright s \prec \bot \quad \mapsto \quad s \prec \top}$$

$$\frac{}{s \succ (\texttt{and}\ e_1\ e_2) \quad \mapsto \quad (\texttt{and}\ \Box\ e_2) \triangleright s \succ e_1}$$

$$\frac{}{(\texttt{and}\ \Box\ e_2) \triangleright s \prec \top \quad \mapsto \quad s \succ e_2}$$

$$\frac{}{(\texttt{and}\ \Box\ e_2) \triangleright s \prec \bot \quad \mapsto \quad s \prec \bot}$$

Initial states are evaluating any expression on the empty stack $\circ \succ e$, and final states are those returning any value to an empty stack $\circ \prec v$.

Question 2 (25 marks)

(a) (6 marks) Give the most general type of the following implicitly-typed MinHS expressions.

i. $\mathsf{InR}\ (\mathsf{InR}\ (3, 4))$

Solution: $\forall b\ a.\ (b + (a + (\texttt{Int} \times \texttt{Int})))$

ii. $\mathsf{recfun}\ f\ x = \mathsf{fst}\ (\mathsf{snd}\ x)$

Solution: $\forall a\ b\ c.\ (a \times (b \times c)) \rightarrow b$

iii. $\mathsf{recfun}\ f\ x = \mathsf{case}\ x\ \mathsf{of}\ \mathsf{InL}\ g \rightarrow g\ \texttt{False}; \mathsf{InR}\ x \rightarrow \texttt{False}$

Solution: $\forall a.\ (\texttt{Bool} \rightarrow \texttt{Bool}) + a \rightarrow \texttt{Bool}$

(b) (6 marks) Explain, with the help of an example, why most-general-type inference is not possible for the simple **rec**-based recursive types we added to MinHS in lectures.
Hint: Consider the type(s) of the term: $\mathsf{roll}\ (\mathsf{InR}\ (\mathsf{roll}\ (\mathsf{InL}\ 3)))$.

Solution: Type inference becomes intractable as it is not clear which of the possible recursive types is intended from the term alone. For example, $\mathsf{roll}\ (\mathsf{InR}\ (\mathsf{roll}\ (\mathsf{InL}\ 3)))$ could be given the type $\textbf{rec}\ t.\ \texttt{Int} + t$, which is perhaps what is intended, but it could also be given the type: $\forall a\ b.\ \textbf{rec}\ t.\ a + (\textbf{rec}\ u.\ \texttt{Int} + b)$, which it could be argued is more general. However, when the type variables a and b are bound, the variable t is not in scope, so the other type $\textbf{rec}\ t.\ \texttt{Int} + t$ is not actually an instance of this type. Therefore there is no most general type that can be inferred.

(c) (6 marks) What is the most general unifier of the following pairs of type terms? If there is no unifier, explain why.

i. $(a \times b) \rightarrow (b \times a)$ and $(\texttt{Int} \times c) \rightarrow (c \times c)$

Solution: $[a := \texttt{Int}, b := \texttt{Int}, c := \texttt{Int}]$

ii. $a \to (a + a)$ and $(b + b) \to b$

Solution: If we naively proceed without an occurs check, we get the infinite substitution $[a := b \times b, b := a \times a]$. There is no unifier as the occurs check fails.

iii. $\texttt{Int} \to \texttt{Int}$ and $\texttt{Float} \to \texttt{Int}$.

Solution: There is no unifier as `Int` cannot be unified with `Float`.

(d) (7 marks) We wish to specify an abstract data type (ADT) in MinHS for describing sets of integers. We include a function to create an empty set, to insert an integer into a set, and to perform union and intersection on sets. We will provide one implementation using a linked list data type `List`. Assume that `List` is a shorthand for the full recursive linked list type:

$$\textbf{rec}\ t.\ \mathbf{1} + (\texttt{Int} \times t))$$

```
Pack List (
  recfun $empty$ :: List = roll (InL ())),
  recfun $insert$ :: (Int → List → List) = ··· ,
  recfun $union$ :: (List → List → List) = ··· ,
  recfun $size$ :: (List → Int) = ···
)
```

What is the type of the above Pack expression? Write a small MinHS program that, given a set ADT,

1. creates an empty set,
2. inserts an element into the set, and then
3. returns the size of the resulting set.

Solution: $(\exists S.\ S \times (\texttt{Int} \to S \to S) \times (S \to S \to S) \times (S \to \texttt{Int}))$
Assuming we have pattern matching in MinHS:

```
recfun $f$ $setM$ =
  Open $setM$
    ($S$. ($empty, insert, union, size$).
      let
        $s = empty$;
        $s' = insert\ 42\ s$;
      in $size\ s'$;
    )
```

Question 3 (25 marks)

(a) (15 marks) Suppose we extend MinHS with an (incorrect) subtyping rule that states

$$\frac{A \leq A' \qquad B \leq B'}{(A \to B) \leq (A' \to B')}$$

i. What is the *variance* of the right hand side of the function arrow $\to$?

Solution: It is *covariant* on the right hand side, as the direction of subtyping is not changed when types are placed on the right hand size of the function arrow.

ii. Assuming `Square` $\leq$ `Rect`, and the variable r : `Rect`, show that the above subtyping rule is incorrect by giving a MinHS expression of type `Square` that evaluates to r.

Solution: Consider the identity function:

$$\textbf{recfun}\ \mathit{id}\ :: (\texttt{Square} \to \texttt{Square})\ x = x$$

By the subtyping rule above, this function also has the type `Rect` $\to$ `Square`. Therefore, *id r* is a well-typed expression of type `Square` but it reduces to r, which is of type `Rect`.

iii. How does the existence of such an expression violate *progress* or *preservation*?

Solution: As *id r* is of type `Square` but it reduces to r of type `Rect`, this violates preservation as preservation requires r to have type `Square`.

iv. Provide a different rule for subtyping of functions that corrects this problem.

Solution: The left-hand side of the rule should be contravariant, as follows:

$$\frac{A' \leq A \qquad B \leq B'}{(A \to B) \leq (A' \to B')}$$

(b) (5 marks) Suppose we have a Show type class with one method, *show* :: $a \to$ `String`, and we write a function:

$$\mathit{exclaim} :: \mathsf{Show}\ a \Rightarrow a \to \texttt{String}$$
$$\mathit{exclaim}\ x = \mathit{show}\ x \mathbin{+\!\!+} \texttt{"!"}$$

Provide an equivalent version of the *exclaim* function that relies on parametric polymorphism and parameter-passing instead of ad-hoc polymorphism.

Solution:

$$\mathit{exclaim}' :: (a \to \texttt{String}) \to a \to \texttt{String}$$
$$\mathit{exclaim}'\ \mathit{show}\ x = \mathit{show}\ x \mathbin{+\!\!+} \texttt{"!"}$$

(c) (5 marks) Suppose we had defined a monadic interface for managing a bank account:

withdraw :: `Amount` $\to$ `Bank Cash` *deposit* :: `Amount` $\to$ `Bank Receipt`

i. What purpose does the monadic Bank type serve?

Solution: There is some hidden state (such as the bank balance), that is encapsulated by the `Bank` monad. The monad ensures that the state is treated linearly, that is, that once cash is withdrawn from an account, we cannot "time travel" and withdraw more cash from the same, pre-withdrawal account.

ii. How would that same purpose be accomplished using *linear types* to model the bank account instead?

Solution: If we explicitly model the `Account` as a linear type, we can pass it in as an explicit state parameter to *withdraw* and *deposit*:

$$\begin{aligned} \mathit{withdraw} &:: \texttt{Amount} \to \texttt{Account} \to \texttt{Account} \otimes \texttt{Cash} \\ \mathit{deposit} &:: \texttt{Amount} \to \texttt{Account} \to \texttt{Account} \otimes \texttt{Receipt} \end{aligned}$$

Question 4 (25 marks)

(a) (8 marks) Decompose the following properties into the intersection of a *safety* and *liveness* property.

i. The process p is the only process that will enter its critical section.

Solution:
Safety: No non-p process will enter its critical section.
Liveness: p will enter its critical section.

ii. The process q will eventually release the lock.

Solution:
Safety: All behaviours.
Liveness: q will eventually release the lock.

iii. No STM transaction will be interrupted.

Solution:
Safety: No STM transaction will be interrupted.
Liveness: All behaviours.

iv. The program will throw an exception, not return a value.

Solution:
Safety: The program will not return a value.
Liveness: The program will throw an exception.

(b) (12 marks) Consider the following concurrent program, consisting of processes **P** and **Q**, which manipulate a shared variable m that starts initialised to zero:

```
Process P              ||    Process Q

while $i < 10$ do            while $j < 10$ do
   $m := m + 1$;                $m := m - 1$;
   $i := i + 1$                 $j := j + 1$
od                           od
```

i. Assume each line is atomic. What are the possible final values of m?

Solution: Depending on the interleaving, the number will range during execution between -10 and 10, however the final value is always 0.

ii. Now only assume atomicity for load and store instructions, not for each line. What are the possible output values in this case?

Solution: There is a race condition when the processes proceed in lock-step. That is:

1. P reads m, call the value it read m_P.
2. Q reads m, call the value it read m_Q. Observe that $m_P = m_Q$.
3. P writes $m_P + 1$ to m.
4. Q writes $m_Q - 1$ to m.

This sequence of events effectively nullifies P incrementing m once. Depending on interleaving, this could happen arbitrarily often to either process, resulting in a final value of m ranging between -10 and 10.

iii. How would you ensure *mutual exclusion* such that the final value of m is guaranteed to be 0, regardless of the atomicity model used?

Solution: We would place the increment/decrement in a *critical section*, using a well known *critical section solution* such as STM or a lock. Using a lock ℓ:

```
Process P             ||   Process Q

while $i < 10$ do          while $j < 10$ do
  take $\ell$;                 take $\ell$;
  $m := m + 1$;                $m := m - 1$;
  release $\ell$;              release $\ell$;
  $i := i + 1$                 $j := j + 1$
od                         od
```

(c) (5 marks) STM is a popular means of concurrency control in programming languages. Describe how STM addresses the main desiderata for concurrency control:

i. Mutual Exclusion
ii. Deadlock-Freedom
iii. Starvation-Freedom

Solution: STM addresses mutual exclusion optimistically, by running transactions in parallel by default, and only restarting them if a transaction is interrupted by changes in another transaction. This means that in low-contention cases, STM can be more efficient.

Deadlock freedom is addressed by STM by reducing the reliance on locks. The only locking mechanism used is in the commit phase where the locking is handled by the language run-time and is guaranteed not to block indefinitely.

Starvation freedom is not guaranteed by STM at all, however in practice it is rare to see indefinite starvation.