COMP3161/9164

Concepts of Programming Languages

SAMPLE EXAM

Term 3, 2020

- Total Number of **Parts**:
- Total Number of **Marks**:
- All parts **are** of equal va
- Answer **all** questions.
- Excessively verbose answers may lose marks.
- Failure to make the declaration or making a false declaration results in a 100% mark penalty.
- Ensure you are the person listed on the declaration.
- All questions must be attempted **individually** withou one else.
- You must **save** your **before** e runs out.
- **Late submissions will not be accepted**
- You may save multiple times before the deadlin mission will be considered.

Declaration

☐ I, name, declare that these answers are **entirely my own**, and that I did not complete this exam with assistance from anyone else.

Part A (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:

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

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

1. (3 Marks) Derive the step-by-step evaluation of
   (and not false (and true not true))$.

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

2. (4 Marks) Are the rules that define Bool unambiguo ? If not, give an example of a term that allows for multiple de

They are unambi ple derivations. This is essentially becau

3. (6 Marks) The semantics rules listed above are small-step. Give an equivalent big-step semantics for this language. To write an inference rule with premises, you can write the rule $\frac{A \quad B}{C}$ as `A, B ⊢ C`.

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

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

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

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

4. (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.

Define a frame f as fo

$$\mid \quad (\texttt{and} \ \square \ e)$$

Define a stack s as a list of frames:

$$s \ ::= \ \circ \mid f \triangleright s$$

Then a machine s n $s \succ e$, or returning a value, written e

$$\frac{}{s \succ \texttt{not}\ e \ \mapsto}$$

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

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

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

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

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

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 $\circ \prec v$.

Part B (25 Marks)

1. (6 Marks) Give the most plicitly-typed MinHS expressions:

   i. InR (InR (3, 4))

   $\forall b\ a.\ (b \to \mathtt{Int} \times \mathtt{Int}) + (\mathtt{Int} \times \mathtt{Int}) + b$

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

   $\forall a\ b\ c.\ (a$

   iii. recfun $f\ x$ = case x of InL $g \to g$ False;

   $\forall a.\ (\mathtt{Bool} \to \mathtt{Bool}) + a \to \mathtt{Bool}$

2. (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: roll (InR (roll (InL 3))).

   Type inference becomes intractable as it is not clear which of the possible recursive types is intended from the term alone. For example, roll (InR (roll (InL 3))) could be given the type **rec** $t.\ \mathtt{Int} + t$, which is perhaps what is intended, but it could also be given the type: $\forall a\ b.$ **rec** $t.\ a + ($**rec** $u.\ \mathtt{Int} + b)$, 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 **rec** $t.\ \mathtt{Int} + t$ is not actually an instance of this type. Therefore there is no most general type that can be inferred.

3. (6 Marks) What is the most general unifier of the following pairs of type terms? If there is no unifier, explain why.

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

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

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

   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 $\texttt{Float} \to \texttt{Int}$

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

4. (7 Marks) We wish to specify an abstract data type (cribing sets of integers. We includ n integer into a set, and to perform union a plementation using a linked list data typ he full recursive linked list type:

$$\textbf{rec}\ t.\ \texttt{1} + (\texttt{I}$$

```
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,

   i. creates an empty set,
   ii. inserts an element into the set, and then
   iii. returns the size of the resulting set.

$$(\exists S.\ S \times (\texttt{Int} \to S \to S) \times (S \to S \to S) \times (S \to \texttt{Int}))$$

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';
    )
```


Part C (25 Marks)

1. (15 Marks) Suppose we extend MinHS with an (i ule that states

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

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

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.

Consider the identity function:

recfun *id* :: (`Square` $\to$ `Square`) $x = x$

By the subtyping rule above, this function also has the type `Rect → 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*?

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.

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

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

2. (5 Marks) Suppose we have a Show type class with $show :: a \to \texttt{String}$, and we write a function:

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

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

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

3. (5 Marks) Consider the following type:

$$\textbf{type}\ Cont\ a = (a \to \mathsf{IO}\ ()) \to \mathsf{IO}\ ()$$

Is the *Cont* type constructor *covariant, contravariant* or *invariant*? Justify your answer with coercion functions if needed.

It is covariant. Consider two types A and B and a coercion function $coerce :: A \to B$. Then we can write:

$$\begin{aligned} coerceCont &:: Cont\ A \to Cont\ B \\ coerceCont\ &axx\ bx = axx\ (bx \circ coerce) \end{aligned}$$

Part D (25 Marks)

1. (8 Marks) Decompose the following properties into t *afety* and *liveness* property.

   i. The process p is the only process that will ente

   Safety: No n
   Liveness: p

   ii. The process q will eventually release the lock.

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

   iii. No STM transaction will be interrupted.

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

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

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

2. (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**
w			
$i := i + 1$			$j := j + 1$
od			**od**

i. Assume each line is atomic. What are the possi

Depending on the interleaving, the number ution between -10 and $

ii. Now only assume atomicity for load and st for each line. What are the possible output values in this case?

There is a race condition when the processes proceed in lock-step. That is: First P reads m, call the value it read m_P. Then Q reads m, call the value it read m_Q. Observe that $m_P = m_Q$. Then, P writes $m_P + 1$ to m and Q writes $m_Q - 1$ to m. \end{enumerate} 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?

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 ℓ;		**take** ℓ;
$m := m + 1$;		$m := m - 1$;
release ℓ;		**release** ℓ;
$i := i + 1$		$j := j + 1$
od		**od**

3. (5 Marks) STM is a popular means of concurrency co languages.
   Describe how STM addresses the following desirable attributes:
   i. Mutual Exclusion
   ii. Deadlock-Free
   iii. Starvation-Fr

STM addresses mutual exclusion optimisticall tions in parallel by default, and only restarting them if a transact changes in another transaction. This means that in low-contentio 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.

END OF SAMPLE EXAM

(don't forget to save!)