

# Endterm Test Program verification with types and logic (IMC060)

17 June 2022

- Additional materials (laptops, tablets, phones, calculators, or books) are **not allowed**.
- The use of your own notes is **not allowed**.
- This test consists of 7 questions.
- You can obtain 100 points in total. Your final exam grade is determined by:

$$\text{final exam grade} \triangleq \min \left( 10, 1 + 9 \cdot \frac{\text{obtained points}}{100 - 23} \right)$$

- **(Changes after the exam took place:)** Since the exam turned out to be too long, Exercise 4 has been turned into a bonus.
- The division of points among the questions is:

Question:	1	2	3	4	5	6	7	Total
Points:	9	13	12	23	15	13	15	100

- Read the text and the questions carefully.
- Write proofs, terms and types in this test according to the conventions introduced during the course. Make sure to be very precise.
- Make sure to motivate your answer to every question.

**Good luck on your test!**

**Question 1 (9 points)**

For a type  $A$  we have the following notions of equality in Coq:

- $\text{eq} : A \rightarrow A \rightarrow \text{Prop}$
- $\text{eqb} : A \rightarrow A \rightarrow \text{bool}$
- $\text{dec} : \text{forall } x1 \ x2 : A, \{ \text{eq } x1 \ x2 \} + \{ \sim \text{eq } x1 \ x2 \}.$

This question is about these three notions of equality.

- [4 points] Explain the difference between  $\text{eq}$  and  $\text{eqb}$ . You should give the formal equivalence between  $\text{eq}$  and  $\text{eqb}$  as a Lemma in Coq syntax.
- [2 points] Explain the type  $\text{forall } x1 \ x2 : A, \{ \text{eq } x1 \ x2 \} + \{ \sim \text{eq } x1 \ x2 \}$  of  $\text{dec}$ .
- [3 points] Define  $\text{eqb}$  in terms of  $\text{dec}$ . Explain how to prove the Lemma of Question 1.a for your definition of  $\text{eqb}$ . You do not need to give a Coq proof script, but you should give a clear description of how the proof is carried out.

**Question 2 (13 points)**

Consider the syntax and small-step semantics of a small language with an “either” construct.

```

Inductive val :=
| VBool : bool -> val
| VNat : nat -> val.

Inductive expr :=
| EVal : val -> expr
| EEither : expr -> expr -> expr
| EIf : expr -> expr -> expr -> expr.

Inductive head_step : expr -> expr -> Prop :=
| Either_head_step_l e1 e2 :
  head_step (EEither e1 e2) e1
| Either_head_step_r e1 e2 :
  head_step (EEither e1 e2) e2
| If_head_step_true e2 e3 :
  head_step (EIf (EVal (VBool true)) e2 e3) e2
| If_head_step_false e2 e3 :
  head_step (EIf (EVal (VBool false)) e2 e3) e3.

Inductive step : expr -> expr -> Prop :=
| do_head_step e e' :
  head_step e e' -> step e e'
| If_step e1 e1' e2 e3 :
  step e1 e1' -> step (EIf e1 e2 e3) (EIf e1' e2 e3).

Inductive steps : expr -> expr -> Prop :=
| steps_refl e :
  steps e e
| steps_step e1 e2 e3 :
  step e1 e2 -> steps e2 e3 -> steps e1 e3.

```

- [3 points] Explain the intuitive semantics of the “either” construct. Your answer should involve an example program. You should give the possible return values of your example program and explain how the operational semantics assigns these return values. You may write your example program in pseudo syntax, *i.e.*, you do not have to use Coq syntax.

- (b) [5 points] Define an interpreter for this language as:

```
Fixpoint interp (e : expr) : list (option val)
```

The `list` type is used to account for non-determinism. The interpreter should return all possible return values as elements of the list. The list should be considered as a set, *i.e.*, the order and duplicates are irrelevant. The `option` type is used to account for expressions that get stuck. You should give your answer in Coq syntax. You are allowed to use standard functions on lists, such as `app : list A -> list A -> list A` and:

```
Fixpoint flat_map {A B} (f : A -> list B) (l : list A) : list B :=
  match l with
  | [] => []
  | x :: l => app (f x) (flat_map f l)
  end.
```

- (c) [5 points] State the correctness of the interpreter w.r.t. the small-step operational semantics. Your correctness lemma(s) should account for both the `Some` and `None` cases. You are allowed to use standard predicates on lists, such as `In : A -> list A -> Prop`. You should give your answer in Coq syntax.

### Question 3 (12 points)

This question is about programming in Rust.

- (a) [4 points] Explain the difference between Rust types that are `Clone` and `Copy`. Your answer should make the following clear:
- Is every type that is `Clone` also `Copy`. If yes, explain why. If not, give a counterexample.
  - Is every type that is `Copy` also `Clone`. If yes, explain why. If not, give a counterexample.
- (b) [3 points] Consider the following function:

```
fn swap1(v : &mut Vec<i32>, i : usize, j : usize) {
  let x = v[i];
  v[i] = v[j];
  v[j] = x
}
```

(Here, `usize` is a large integer type that can represent all array indices.)

Does the Rust type system accept the function `swap1`? If yes, explain why. If not, explain exactly where the Rust type checker will complain.

- (c) [5 points] Implement the function:

```
fn swap2<T : Clone>(v : &mut Vec<T>, i : usize, j : usize)
```

You should explain why the Rust type checker accepts your implementation.

### Question 4 (23 points)

We consider the semantics of a simply-typed language with `spawn` and `join` similar to those operations in Rust. The formal syntax of our language is as follows:

$$e \in \text{Expr} ::= x \mid n \mid \text{tid} \mid \lambda x. e \mid e_1 e_2 \mid \text{spawn } e \mid \text{join } e$$

$$A \in \text{Type} ::= \text{nat} \mid A_1 \rightarrow A_2 \mid \text{joinhandle } A$$

We let variables  $x \in \text{string}$ , numerals  $n \in \mathbb{N}$ , and thread IDs  $\text{tid} \in \mathbb{N}$ .

Aside from the usual typing rules of the simply-typed lambda calculus with natural numbers, the type system has the following typing rules for **spawn** and **join**:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \text{spawn } e : \text{joinhandle } A} \quad \frac{\Gamma \vdash e : \text{joinhandle } A}{\Gamma \vdash \text{join } e : A}$$

The intuitive semantics is that **spawn**  $e$  creates new thread that runs expression  $e$ , and returns the thread's join handle (represented as a thread ID). The construct **join**  $e$  will wait for the spawned thread  $e$  to terminate. Since the type system is unrestricted (*i.e.*, not substructural), one can use a join handler zero or multiple times, possibly in different threads. The idea is that each **join** will spin until the associated thread has terminated, and then obtains the return value of that thread.

We let  $\text{Val} \subseteq \text{Expr}$  denote the subset of expressions that are values, *i.e.*, numerals  $n$ , thread IDs  $\text{tid}$ , and functions  $\lambda x. e$ . Configurations  $\sigma$  are represented as lists of threads  $e_1 \dots e_n$ . The main thread is the first element in the configuration, followed by all other threads in the order they were spawned. Threads that have terminated (*i.e.*, have reduced to a value) are kept in the configuration. The small-step reduction  $\sigma \Rightarrow_i \sigma'$  says that thread  $i$  in configuration  $\sigma$  can step to configuration  $\sigma'$ :

$$\begin{array}{c} \frac{\sigma(i) = k((\lambda x. e) v) \quad \text{ctx } k}{\sigma \Rightarrow_i \sigma[i := k(\text{subst } x v e)]} \quad \frac{\sigma(i) = k(\text{spawn } e) \quad \text{tid} = \text{length } \sigma \quad \text{ctx } k}{\sigma \Rightarrow_i (\sigma[i := k \text{tid}] ++ [e])} \\[10pt] \frac{\sigma(i) = k(\text{join } \text{tid}) \quad \sigma(\text{tid}) = e \quad e \notin \text{Val} \quad \text{ctx } k}{\sigma \Rightarrow_i \sigma} \\[10pt] \frac{\sigma(i) = k(\text{join } \text{tid}) \quad \sigma(\text{tid}) = v \quad v \in \text{Val} \quad \text{ctx } k}{\sigma \Rightarrow_i \sigma[i := k v]} \end{array}$$

We represent evaluation contexts  $k$  as functions from expressions to expressions. The judgment  $\text{ctx } k$  says that  $k$  is a valid evaluation context. We use the notation  $\sigma(i)$  to look up the  $i$ th element of a list (*i.e.*, the assertion  $\sigma(i) = e$  means that the index  $i$  is within bounds of the list  $\sigma$ , and the value  $e$  is stored at position  $i$  in the list  $\sigma$ ), and the notation  $\sigma[i := e]$  to overwrite the  $i$ th element of the list  $\sigma$  with value  $e$ .

The step relation  $\sigma \Rightarrow \sigma'$  non-deterministically lets a thread take a step:

$$\frac{\sigma \Rightarrow_i \sigma' \quad i < \text{length } \sigma}{\sigma \Rightarrow \sigma'}$$

Our goal is to prove type safety, that is: If  $\emptyset \vdash e : A$ , then  $\text{safe } [e]$ . Safety is defined as follows:

$$\text{safe } \sigma \triangleq \forall \sigma', (\sigma \Rightarrow^* \sigma') \rightarrow \forall i < \text{length } \sigma'. (\sigma'(i) \in \text{Val}) \vee (\exists \sigma''. \sigma' \Rightarrow_i \sigma'')$$

Here,  $\Rightarrow^*$  is the reflexive-transitive closure of  $\Rightarrow$ .

- (a) [4 points] To prove type safety, we need to define a run-time typing judgment  $\vdash_{\text{cfg}} \sigma : A$  for configurations. We then prove the following properties of the run-time typing judgment:

- Initialization:  $\emptyset \vdash e : A$  implies  $\vdash_{\text{cfg}} [e] : A$

- Preservation:  $\vdash_{\text{cfg}} \sigma : A$  and  $\sigma \Rightarrow \sigma'$  implies  $\vdash_{\text{cfg}} \sigma' : A$
- Progress:  $\vdash_{\text{cfg}} \sigma : A$  implies  $\forall i < \text{length } \sigma. (\sigma(i) \in \text{Val}) \vee (\exists \sigma'. \sigma \Rightarrow_i \sigma')$

Explain how these three properties imply type safety.

- (b) [7 points] To define the run-time typing judgment  $\vdash_{\text{cfg}} \sigma : A$  for configurations, we first define a run-time typing judgment  $\Sigma \mid \Gamma \vdash e : A$  for expressions. Here,  $\Sigma$  is a *thread typing* represented as a list containing the types of all threads in the configuration. Give all inference rules for the run-time typing judgment  $\Sigma \mid \Gamma \vdash e : A$  required to prove type safety.
- (c) [6 points] Give a definition of the run-time typing judgment  $\vdash_{\text{cfg}} \sigma : A$  for configurations and briefly indicate why the three properties in Question 4.a hold. You do not need to give a formal proof of these properties.
- (d) [3 points] Our definition of type safety does not rule out configurations that are deadlocked *i.e.*, configurations consisting of threads that spin indefinitely long via `join` to wait on each other. Give an example of a configuration  $\sigma$  that is deadlocked, but for which `safe`  $\sigma$  holds. Explain your answer.
- (e) [3 points] Does the type system rule out deadlocks? If yes, give an intuition why. If not, give an example of a program that is well-typed, but that deadlocks.

### Question 5 (15 points)

For each proposition of separation logic below, state precisely the set of heaps it describes. You should explain your answers. (Recall that  $\top$  is `True`,  $\perp$  is `False`, and  $\neg P$  is  $P \rightarrow \perp$ .)

- (a) [3 points]  $l \mapsto n * \neg(l \mapsto n)$
- (b) [3 points]  $l \mapsto n \wedge \neg(l \mapsto n)$
- (c) [3 points]  $(l \mapsto n * \top) \wedge (k \mapsto m * \top)$
- (d) [3 points]  $l \mapsto n \wedge k \mapsto (-n)$
- (e) [3 points]  $l \mapsto n * ((k \mapsto m) \rightarrow \text{emp})$

### Question 6 (13 points)

In this exercise we consider a linear type system with a typing judgment  $\Gamma_1 \vdash e : A \dashv \Gamma_2$  with two contexts. Here,  $\Gamma_1$  is the *pre-typing context* and  $\Gamma_2$  is the *post-typing context*. The idea is that the post-typing context  $\Gamma_2$  contains the variables from  $\Gamma_1$  that are not used by the expression  $e$ . Examples of typing rules are:

$$\frac{n \in \mathbb{N}}{\Gamma \vdash n : \text{nat} \dashv \Gamma} \quad \frac{}{\Gamma, x : A \vdash x : A \dashv \Gamma} \quad \frac{\Gamma_1 \vdash e_1 : A \multimap B \dashv \Gamma_2 \quad \Gamma_2 \vdash e_2 : A \dashv \Gamma_3}{\Gamma_1 \vdash e_1 e_2 : B \dashv \Gamma_3}$$

- (a) [3 points] State a theorem that formalizes the expected equivalence between the standard linear typing judgment  $\Gamma \vdash e : A$  with a single context (from week 11), and the judgment  $\Gamma_1 \vdash e : A \dashv \Gamma_2$  with two contexts.
- (b) [3 points] Give the typing rule for `let`  $x = e_1$  `in`  $e_2$  using the typing judgment with two contexts.
- (c) [4 points] Assume that you have to implement a type checker for a linear type system, *i.e.*, a function/algorithm that given an expression computes its type. Explain why the

typing judgment  $\Gamma \vdash e : A$  with one context is not directly suitable for implementing a type checking function/algorithm, and how the typing judgment  $\Gamma_1 \vdash e : A \dashv \Gamma_2$  with two contexts helps. Clearly describe the type signature of the type checking functions that you consider. (Note: You do not have to worry about other challenging aspects of type checking that also appear when considering ordinary/unrestricted programming languages, like the inference of function types of  $\lambda$ -expressions/types of  $\lambda$ -bound variables.)

- (d) [3 points] Give the semantic interpretation of the judgment  $\Gamma_1 \vdash e : A \dashv \Gamma_2$  in separation logic. You are allowed to use the parallel substitution  $\text{subst\_map } \vec{v} e$ , and the semantic context typing  $\text{ctx\_typed } \Gamma \vec{v}$ , where  $\vec{v}$  is a finite map from variables names to values.

### Question 7 (15 points)

This exercise is about the verification of the following concurrent program using Iris:

```
foo n  $\triangleq$  let r = alloc n in
  let lk = newlock () in
    (
      acquire lk;
      r := !r - 10;
      if (!r) < 0 then 1 + () else release lk
    ) || (
      acquire lk;
      r := !r - 20;
      release lk
    );
  ()
```

For this exercise, you need to use the following rules of Iris for locks  $lk$  with assertion  $\text{isLock } lk \ R$  and ghost variables  $\gamma$  with assertions  $\gamma \hookrightarrow_\bullet n$  and  $\gamma \hookrightarrow_\circ n$ :

$\{ R \} \text{newlock } () \{ lk. \text{isLock } lk \ R \}$	(Ht-new-lock)
$\{ \text{isLock } lk \ R \} \text{acquire } lk \{ R \}$	(Ht-acquire)
$\{ \text{isLock } lk \ R * R \} \text{release } lk \{ \text{True} \}$	(Ht-release)
$\text{isLock } lk \ R \multimap \text{isLock } lk \ R * \text{isLock } lk \ R$	(Lock-dup)
$\text{True} \multimap \models (\exists \gamma. \gamma \hookrightarrow_\bullet n * \gamma \hookrightarrow_\circ n)$	(Ghost-alloc)
$\gamma \hookrightarrow_\bullet n * \gamma \hookrightarrow_\circ m \multimap n = m$	(Ghost-agree)
$\gamma \hookrightarrow_\bullet n * \gamma \hookrightarrow_\circ m \multimap \models (\gamma \hookrightarrow_\bullet n' * \gamma \hookrightarrow_\circ n')$	(Ghost-update)

- (a) [3 points] What are the possible behaviors of `foo n`, where  $n \in \mathbb{Z}$ ? Give a Hoare triple for `foo n` that exactly describes the safe behaviors. Give an intuitive explanation for why this Hoare triple holds.
- (b) [3 points] To verify that the function `foo n` satisfies the Hoare triple, we need to come up with an invariant  $R$  for the lock  $lk$  that guards the value of location  $r$ . Use ghost variables to give a lock invariant  $R$  with which you can prove your specification. Explain what initial values you use for the ghost variables.
- (c) [9 points] Give a proof outline for your Hoare triple for the function `foo n`.