

# Exam Program verification with types and logic (IMC060)

16 June 2023

- Additional materials (laptops, tablets, phones, calculators, or books) are **not allowed**.
- The use of your own notes is **not allowed**.
- This test consists of 6 questions.
- You can obtain 90 points in total. Your final exam grade is determined by:

$$\text{final exam grade} \triangleq 1 + 9 \cdot \frac{\text{obtained points}}{90}$$

- The division of points among the questions is:

Question:	1	2	3	4	5	6	Total
Points:	15	15	15	15	15	15	90

- 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 (15 points)**

Consider the functions `fold_left` and `fold_right` defined in pseudo-Coq as follows:

```

Fixpoint fold_left f z xs :=
  match xs with
  | [] => z
  | x :: xs => fold_left f (f z x) xs
  end.

Fixpoint fold_right f z xs :=
  match xs with
  | [] => z
  | x :: xs => f x (fold_right f z xs)
  end.

```

- (a) [2 points] Write down the most general types of `fold_left` and `fold_right`.

**Solution:**

```

fold_left : forall {A B}, (B -> A -> B) -> B -> list A -> B
fold_right : forall {A B}, (A -> B -> B) -> B -> list A -> B

```

- (b) [3 points] Using `fold_right`, define the predicate `In x xs` that expresses whether an element `x` is in the list `xs`. The type signature should be `In : forall A, A -> list A -> Prop`.

**Solution:** (Answers do not need to be point-free.)

```

Definition In : forall {A}, A -> list A -> Prop := fun A x =>
  fold_right (compose or (eq x)) False.

```

- (c) [3 points] Using `fold_left`, implement `reverse : forall A, list A -> list A`. Your solution should have linear asymptotic time complexity.

**Solution:** (Answers do not need to be point-free.)

```

Definition reverse : forall {A}, list A -> list A := fun A =>
  fold_left (flip cons) [].

```

- (d) [3 points] Assume we are in the special case where  $f : A \rightarrow A \rightarrow A$ . In this case, we can state a lemma that relates `fold_left` to `fold_right` as follows:

```

Lemma fold_left_right : forall {A} (f : A -> A -> A) z xs,
  (* INSERT HERE *) ->
  fold_left f z xs = fold_right f z xs

```

What precondition(s) should be used?

**Solution:**

```

Lemma fold_left_right : forall {A} (f : A -> A -> A) z xs,
  (forall x y, f x y = f y x) ->
  (forall x1 x2 x3, f x1 (f x2 x3) = f (f x1 x2) x3) ->
  fold_left f z xs = fold_right f z xs.

```

- (e) [4 points] Assume we try to prove the lemma `fold_left_right` by starting with:

```

intros A f z xs Hpre.
induction xs as [|x xs IH].

```

In order to complete this proof, you need to state an auxiliary lemma about `fold_left` or `fold_right`. Give the statement of this lemma and explain why it is useful. (You do **not** need to give the proof of the auxiliary lemma.)

**Solution:**

```

Lemma fold_left_f : forall {A} (f : A -> A -> A) z xs x,
  (forall x y, f x y = f y x) ->
  (forall x1 x2 x3, f x1 (f x2 x3) = f (f x1 x2) x3) ->
  fold_left f (f z x) xs = f x (fold_left f z xs).

```

## Question 2 (15 points)

Assume that we have a pure, statically-typed language with a small-step operational semantics. Formally, assume that we have a set of expressions `Expr`, a set of values `Val`  $\subseteq$  `Expr`, a set of types `Type`, a small-step operational semantics  $e \rightsquigarrow e'$ , and a typing judgment  $\vdash e : A$  for closed expressions  $e$ .

This language enjoys *type safety* if for all expressions  $e \in \text{Expr}$  and types  $A \in \text{Type}$  such that  $\vdash e : A$ , we have:

$$\forall e' \in \text{Expr}. (e \rightsquigarrow^* e') \rightarrow (e' \in \text{Val}) \vee (\exists e''. e' \rightsquigarrow e'')$$

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

- (a) [3 points] Type safety is often proved via the method of progress and preservation. Give the statements of progress and preservation and explain why together they imply type safety.

**Solution:**

- Preservation: if  $e \vdash A$  : and  $e \rightsquigarrow e'$  then  $e' \vdash A$  :.
- Progress: if  $e \vdash A$  : then  $(e \in \text{Val}) \vee (\exists e'. e \rightsquigarrow e')$ .

Do induction on  $\rightsquigarrow^*$ .

- (b) [6 points] Bob proposes the following property, which combines aspects of progress and preservation into a single statement:

$$\text{bob} : \forall (e \in \text{Expr})(A \in \text{Type}). (\vdash e : A) \rightarrow (e \in \text{Val}) \vee (\exists e'. (e \rightsquigarrow e') \wedge (\vdash e' : A))$$

Does **bob** imply type safety? If yes, give a proof. If no, give a counterexample.

Your proof should be precise and make clear where and how you use **bob**. The proof should be written in English, not Coq. Your counterexample should involve a definition of a set of expressions **Expr**, a set of values  $\text{Val} \subseteq \text{Expr}$ , a set of types **Type**, a small-step operational semantics  $e \rightsquigarrow e'$ , and a typing judgment  $\vdash e : A$ . You should clearly explain that **bob** holds for your counterexample, but type safety does not.

**Solution:** Let  $\text{Expr} \triangleq \{i, f_1, f_2\}$  and  $\text{Val} \triangleq \{f_1\}$  and  $\text{Type} \triangleq \{()\}$ .

$$\overline{i \rightsquigarrow f_1} \qquad \overline{i \rightsquigarrow f_2} \qquad \overline{\vdash i : ()} \qquad \overline{\vdash f_1 : ()}$$

This language does not satisfy type safety. From  $i$  we can step to  $f_2$ , which is final, but not a value. This language satisfies **bob**. Every well-typed expression is either a value or can step to another well-typed expression.

- (c) [6 points] Does progress and preservation imply **bob**? If yes, give a proof. If no, give a counterexample.

Your proof should be precise and make clear where and how you use progress and preservation. The proof should be written in English, not Coq. Your counterexample should involve a definition of a set of expressions **Expr**, a set of values  $\text{Val} \subseteq \text{Expr}$ , a set of types **Type**, a small-step operational semantics  $e \rightsquigarrow e'$ , and a typing judgment  $\vdash e : A$ . You should clearly explain that progress and preservation hold for your counterexample, but **bob** does not.

**Solution:** Progress and preservation imply **bob**. To prove **bob**, consider an expression  $e$  and type  $A$  with  $\vdash e : A$ . By progress we know that  $e \in \text{Val}$  or  $e \rightsquigarrow e'$  for some  $e'$ . If  $e \in \text{Val}$ , we are done. If  $e \rightsquigarrow e'$  we know  $\vdash e' : A$  by preservation. In that case, we are done too.

### Question 3 (15 points)

This question is about programming in Rust.

- (a) [5 points] Consider the following Rust program:

```
1  let mut m = vec![1,2,3,4];
2  let a = &m[0];
3  m.push(5);
4  println!("{}", *a);
```

Does the Rust type checker accept this program? Is this program safe? What is the behavior when running the program?

**Solution:** The Rust type checker does not accept this program. The lifetime of **a** ends at line 3 because **m** is used. Type checking fails at line 4 with a lifetime error.

This program is not safe. The **push** on line 3 might have to reallocate the buffer of the vector **m**. In that case, **a** becomes a dangling pointer, and the program might read old memory or crash with a segmentation fault.

(b) [5 points] Consider the following Rust program:

```

1  let m = Mutex::new(vec![1,2,3,4]);
2  let guard1 = m.lock().unwrap();
3  let a = &guard1[0];
4  let mut guard2 = m.lock().unwrap();
5  guard2.push(5);
6  println!("{}", *a);

```

Does the Rust type checker accept this program? Is this program safe? What is the behavior when running the program?

**Solution:** The Rust type checker accepts this program, and therefore this program is safe.

The program will cause a deadlock. There are two attempts to acquire the lock resulting in lock guards `guard1` and `guard2` (line 2 and line 4). Both locks will only be released at the end of the function. Hence the second `m.lock()` will never terminate/deadlock.

(c) [5 points] Consider the following two programs. Program one:

```

1  let mut x : Option<Vec<i32>> = Some(vec![1,2,3]);
2  let y : Vec<i32> = vec![1,2,3];
3  let p: &Vec<i32> = match &x {
4      None => { let z = y; &z },
5      Some(v) => v
6  };
7  println!("p[0] = {}", p[0]);

```

Program two:

```

1  let mut x : Option<Vec<i32>> = Some(vec![1,2,3]);
2  let y : Vec<i32> = vec![1,2,3];
3  let p: &Vec<i32> = match &x {
4      None => { let z = &y; z },
5      Some(v) => v
6  };
7  println!("p[0] = {}", p[0]);

```

The Rust type checker accepts one of these programs but not the other. Which one, and why? Is the program that the Rust type checker rejects safe? Explain why or why not.

**Solution:** The Rust type checker accepts the second program, but not the first.

In the first, we move `y` into `z`, and assign the address `&z` of `z` to `p` (line 4). Since the scope of `z` ends in line 4, `&z` becomes dangling reference. However, since `x` is `Some`, line 4 (the `None` branch) is never reached, so this would not cause actual unsafety.

In the second, `z` becomes a reference to `y`, which we in turn assign to `p` (line 4).

#### Question 4 (15 points)

We start with the lambda calculus with references, *i.e.*, with allocation (`alloc e`), load (`!l`), store (`l := e`), and free (`free l`). We extend it with a new nondeterministic choice operator  $e_1$  **either**  $e_2$ , which will execute either  $e_1$  or  $e_2$  entirely. For example  $(10 + 1)$  **either**  $2$  will result in either  $11$

or 2. The expressions  $e_1$  and  $e_2$  may contain side-effects, *e.g.*,  $(x := !x + 1)$  **either**  $(x := !x * 2)$  will either increment or double the value of location  $x$ .

We give this language a standard small-step operational semantics  $(e, h) \rightsquigarrow (e', h)$ , where  $e, e'$  are expressions and  $h, h'$  are heaps. The operational semantics of  $e_1$  **either**  $e_2$  is as follows:

$$\frac{}{(e_1 \text{ either } e_2, h) \rightsquigarrow (e_1, h)} \qquad \frac{}{(e_1 \text{ either } e_2, h) \rightsquigarrow (e_2, h)}$$

- (a) [5 points] Alice and Bob want to use separation logic to verify **safety** of programs in this language. They propose the following definitions of the weakest precondition connective:

$$\begin{aligned} \text{WP}_{\text{alice}} e \{ \Phi \} &\triangleq \lambda h_1. \forall e', h'. \forall h_f. \text{dom } h_1 \cap \text{dom } h_f = \emptyset \rightarrow \\ &\quad (e, h_1 \cup h_f) \rightsquigarrow^* (e', h') \rightarrow \\ &\quad \exists h_2. \text{dom } h_2 \cap \text{dom } h_f = \emptyset \wedge \\ &\quad \quad h' = h_2 \cup h_f \wedge \\ &\quad \quad ((e' \in \text{Val} \wedge \Phi \ e' \ h_2) \vee \text{can\_step}(e', h')) \\ \text{WP}_{\text{bob}} e \{ \Phi \} &\triangleq \lambda h_1. \forall h_f. \text{dom } h_1 \cap \text{dom } h_f = \emptyset \rightarrow \\ &\quad \exists v, h_2. \text{dom } h_2 \cap \text{dom } h_f = \emptyset \wedge \\ &\quad \quad (e, h_1 \cup h_f) \rightsquigarrow^* (v, h_2 \cup h_f) \wedge \\ &\quad \quad \Phi \ v \ h_2 \end{aligned}$$

Quantifiers in red were accidentally omitted in original version.

Here,  $\rightsquigarrow^*$  is reflexive-transitive closure of the relation  $\rightsquigarrow$ , and  $\text{can\_step}(e', h')$  is defined as  $\exists e'', h''. (e', h') \rightsquigarrow (e'', h'')$ .

Considering the goal is to verify **safety**, should we use Alice's or Bob's version? Explain your answer. For the version that we should **not** use, you should give a program  $e$  that is unsafe, but can be verified using the weakest precondition (that is,  $\text{WP } e \{ \Phi \}$  is valid for some  $\Phi$ ).

**Solution:** We should use Alice's version. Alice's version says that the program cannot go wrong. Every execution either leads to a value, or one can keep on stepping. Bob's version says that there exists an execution to a final state that satisfies the postcondition. However, with non-determinism this is not sufficient. No execution should go wrong. For example, we can prove  $\text{WP}_{\text{bob}} (10 \text{ either } (1 + ())) \{ v. v = 10 \}$  using Bob's version, while that program is not safe if non-deterministically the second choice is used.

- (b) [3 points] Give the Hoare-triple inference rule for  $e_1$  **either**  $e_2$ , of the following form:

$$\frac{\text{FILL OUT} \quad \text{FILL OUT}}{\{ \text{FILL OUT} \} e_1 \text{ either } e_2 \{ \text{FILL OUT} \}}$$

Recall that Hoare triples are defined as  $\{ P \} e \{ \Phi \} \triangleq P \vdash \text{WP } e \{ \Phi \}$ , where the postcondition has type  $\Phi : \text{Val} \rightarrow \text{sepProp}$ .

**Solution:**

$$\frac{\{P\}_{e_1} \{ \Phi \} \quad \{P\}_{e_e} \{ \Phi \}}{\{P\}_{e_1} \text{either } e_2 \{ \Phi \}}$$

- (c) [3 points] Give the weakest precondition rule for  $e_1 \text{ either } e_2$ , of the following form:

$$\text{FILL OUT} \vdash \text{WP } (e_1 \text{ either } e_2) \{ \Phi \}$$

**Solution:**

$$\text{WP } e_1 \{ \Phi \} \wedge \text{WP } e_2 \{ \Phi \} \vdash \text{WP } (e_1 \text{ either } e_2) \{ \Phi \}$$

- (d) [4 points] Julie proposes the following linear typing rule for the choice operator:

$$\frac{\Gamma_1 \vdash e_1 : A \quad \Gamma_2 \vdash e_2 : A \quad \Gamma_1 \cap \Gamma_2 = \emptyset}{\Gamma_1 \cup \Gamma_2 \vdash e_1 \text{ either } e_2 : A}$$

Explain if this typing rule is sound or not. If it is not, you should provide a counterexample. This involves a program that can be type-checked, but that performs an unsafe operation (such as use-after-free or double-free) or leaks memory.

**Solution:** This typing rule is unsound. Consider:

```
let l = alloc 10 in ((free l) either 10)
```

Depending on the non-deterministic choice, either  $l$  will be freed, or  $l$  is leaked.

### Question 5 (15 points)

For each proposition of separation logic below, state precisely the set of heaps it describes. You should explain your answers.

We represent locations  $l \in \text{Loc} \triangleq \mathbb{N}$  and values  $v \in \text{Val} \triangleq \mathbb{N}$  as natural numbers. Recall that **True** is the separation logic proposition representing the set of all possible heaps.

- (a) [3 points]  $(\exists l. l \mapsto l) * \text{True}$

**Solution:** Heaps that have at least one location that points to itself satisfy.

The proposition states that the heap should be split up into two parts. One that satisfies  $(\exists l. l \mapsto l)$ , and one part that is arbitrary. The proposition  $(\exists l. l \mapsto l)$  describes the heaps that have exactly a location that points to itself. Hence the whole version with  $* \text{True}$  describes the heaps that have at least one location that points to itself.

(b) [3 points]  $\forall l. l \mapsto 10$

**Solution:** No heap.

This proposition describes heaps  $h$  that are equal to  $\{l \mapsto 10\}$  for every location  $l$ . Since there are at least two different locations, say 0 and 1, we get  $h = \{0 \mapsto 10\} = \{1 \mapsto 10\}$ , which is contradictory. Hence no heap satisfies this proposition.

(c) [3 points]  $(7 \mapsto 10) \rightarrow (7 \mapsto 10)$

**Solution:** All heaps.

The proposition  $P \rightarrow P$  is logically equivalent to **True**. Hence all heaps satisfy this proposition.

(d) [3 points]  $(7 \mapsto 10) \multimap (7 \mapsto 10)$

**Solution:** The empty heap and heaps that contain location 7.

The proposition  $(7 \mapsto 10) \multimap (7 \mapsto 10)$  holds for heaps  $h$  where  $\text{dom } h \cap \{7\} = \emptyset$  implies  $\{7 \mapsto 10\} = h \cup \{7 \mapsto 10\}$ . Only if  $h$  is empty, the conclusion is true. Only if  $h$  contains location 7, the premise is false.

(e) [3 points]  $(\exists l. l \mapsto 12) \wedge (\exists n. 7 \mapsto n)$

**Solution:** Only the singleton heap  $\{7 \mapsto 12\}$ .

The proposition states that the heap should satisfy both  $(\exists l. l \mapsto 12)$  and  $(\exists n. 7 \mapsto n)$ . The first conjunct says that the heap should have exactly one location with value 12. The second conjunct says that heap should have exactly the location 7 with an arbitrary value. Together this gives that the heap should be the singleton  $\{7 \mapsto 12\}$ .

### Question 6 (15 points)

This exercise is about the verification of the following program using separation logic:

```
search ll x b  $\triangleq$  match ll with
| inl ()       $\Rightarrow$  ()
| inr node  $\Rightarrow$  (if x = fst (!node) then b := true);
               search (snd (!node)) x b
end
```

The version written in Coq is as follows:

```
Definition search :=
  recclosure: "rec" "ll" "x" "b" =>
    match: "ll" with
      InjL "_" => VUnit
    | InjR "node" =>
      (if: "x" =: EFst !"node" then "b" <- VBool true else VUnit);;
      "rec" (ESnd !"node") "x" "b"
    end.
```



The function `search ll x b` searches for the value  $x$  in the imperative list  $ll$ , and modifies the boolean  $b$  to indicate whether  $x$  was found. The list is represented as `inl ()` (the empty list), or `inr l'`, where  $l'$  is a pointer to a tuple  $(x, ll')$  containing the head element  $x$  and tail list  $ll'$ .

- (a) [5 points] Define a predicate `is_list ll  $\vec{x}$`  that states that  $ll$  is a mutable list that contains the numbers in the mathematical list  $\vec{x}$ . The type of `is_list` should be `Val  $\rightarrow$  List  $\mathbb{N} \rightarrow$  sepProp`. You can use either math or Coq syntax. If you write Coq code, you are allowed to ignore coercions such as `VNat` and `VRef`.

**Solution:**

```
Fixpoint is_list (ll : val) (ns : list nat) : sepProp :=
  match ns with
  | [] => @[ ll = VInjL VUnit ]
  | n :: ns => Ex l ll',
    @[ ll = VInjR (VRef l) ] ** l  $\leadsto$  VPair (VNat n) ll' ** is_list ll' ns
  end.
```

- (b) [3 points] Consider the following Hoare triple for `search ll x b`:

$$\{ \text{is\_list } ll \vec{x} * b \mapsto \text{false} \} \text{search } ll \ x \ b \{ w. w = () * \text{is\_list } ll \vec{x} * b \mapsto (x \in \vec{x}) \}$$

Explain why induction on the list  $\vec{x}$  is not strong enough to prove this Hoare triple, and give a strengthened version of the Hoare triple that gives a strong enough induction hypothesis.

**Solution:** If the ‘if’ succeeds, the value of location  $b$  will no longer be `false`. The strengthened version is:

$$\{ \text{is\_list } ll \vec{x} * b \mapsto \beta \} \text{search } ll \ x \ b \{ w. w = () * \text{is\_list } ll \vec{x} * b \mapsto (\beta \vee x \in \vec{x}) \}$$

- (c) [7 points] Give a proof outline for the strengthened Hoare triple, which shows that the definition of `search ll x b` satisfies the Hoare triple.

**Solution:**

```

search ll x b  $\triangleq$  { is_list ll  $\vec{x} * b \mapsto \beta$  }
  match ll with
  | inl ()  $\Rightarrow$  { is_list (inl ())  $\vec{x} * b \mapsto \beta$  }  $\vec{x} = []$ 
    ()
    { w. w = () * is_list (inl ()) [] * b  $\mapsto \beta$  }
    { w. w = () * is_list (inl ()) [] * b  $\mapsto (\beta \vee x \in [])$  }
  | inr node  $\Rightarrow$  { is_list (inr node)  $\vec{x} * b \mapsto \beta$  }  $\vec{x} := x_1 \vec{x}_2$ 
    { node  $\mapsto (x_1, v') * \text{is\_list } v' \vec{x}_2 * b \mapsto \beta$  }
    (if x = fst (!node) then b := true);
    { node  $\mapsto (x_1, v') * \text{is\_list } v' \vec{x}_2 * b \mapsto (\beta \vee x = x_1)$  }
    search (snd (!node)) x b
    { w. w = () * node  $\mapsto (x_1, v') * \text{is\_list } v' \vec{x}_2 *$  }
    { w. b  $\mapsto ((\beta \vee x = x_1) \vee x \in \vec{x}_2)$  }
    { w. w = () * is_list (inr node)  $\vec{x} * b \mapsto (\beta \vee x \in (x_1 \vec{x}_2))$  }
  end
  { w. w = () * is_list ll  $\vec{x} * b \mapsto (\beta \vee x \in \vec{x})$  }

```