

# Exam Program verification with types and logic (IMC060)

14 June 2024

- 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:	21	13	12	14	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 (21 points)**

In this exercise you will implement a type checker for a pure programming language, whose syntax and type system are defined in Coq as follows:

```

Inductive ty := TBool | TNat | TFun (A B : ty).

Inductive expr :=
  | ENat (n : nat)
  | EVar (x : string)
  | ELam (x : string) (A : ty) (e : expr)
  | EApp (e1 e2 : expr).

Inductive subty : ty -> ty -> Prop :=
  | subty_refl A : subty A A
  | subty_bool_nat : subty TBool TNat
  | subty_fun A1 A2 B1 B2 :
    subty B1 A1 ->
    subty A2 B2 ->
    subty (TFun A1 A2) (TFun B1 B2).

Inductive typed : stringmap ty -> expr -> ty -> Prop :=
  | Bool_false_typed Gamma :
    typed Gamma (ENat 0) TBool
  | Bool_true_typed Gamma :
    typed Gamma (ENat 1) TBool
  | Nat_typed Gamma n :
    typed Gamma (ENat n) TNat
  | Var_typed Gamma x A :
    StringMap.lookup Gamma x = Some A ->
    typed Gamma (EVar x) A
  | Lam_typed Gamma x e A B :
    typed (StringMap.insert x A Gamma) e B ->
    typed Gamma (ELam x A e) (TFun A B)
  | App_typed Gamma e1 e2 A B :
    typed Gamma e1 (TFun A B) ->
    typed Gamma e2 A ->
    typed Gamma (EApp e1 e2) B
  | Subsumption Gamma e A B :
    subty A B ->
    typed Gamma e A ->
    typed Gamma e B.

```

The key feature of this language is that it has a boolean and natural number type (TBool and TNat), but only natural number literals (ENat n). The literals 0 and 1 can be used as booleans and natural numbers. The subtyping relation (subty) and subsumption rule (Subsumption) can be used to coerce expressions of boolean type into natural number type.

(a) [2 points] Explain why the subtyping relation for functions is **not** defined as:

```

  | subty_fun A1 A2 B1 B2 :
    subty A1 B1 -> (* WRONG *)
    subty A2 B2 ->
    subty (TFun A1 A2) (TFun B1 B2).

```

**Solution:** The type A is the input of the function, so the direction of the subtyping is reversed compared to the output B.

- (b) [5 points] Our initial attempt to define a bool version of the subtyping relation is:

```
Fixpoint bsubty (A1 A2 : ty) : bool :=
  match A1, A2 with
  | TBool, TBool => true
  | TNat, TNat => true
  | TBool, TNat => true
  | TFun A1 A2, TFun B1 B2 => bsubty B1 A1 && bsubty A2 B2
  | _, _ => false
end.
```

Explain why Coq does not accept this definition, and give an alternative definition that will be accepted by Coq. You are recommended to introduce an auxiliary Fixpoint.

**Solution:** The call to `bsubty B1 A1` is not structurally recursive. A valid alternative definition is as follows:

```
Fixpoint bsubty_aux (ff : bool) (A1 A2 : ty) : bool :=
  match A1, A2 with
  | TBool, TBool => true
  | TNat, TNat => true
  | TBool, TNat => negb ff
  | TNat, TBool => ff
  | TFun A1 A2, TFun B1 B2 => bsubty_aux (negb ff) A1 B1 && bsubty_aux ff A2 B2
  | _, _ => false
end.
Definition bsubty : ty -> ty -> bool := bsubty_aux false.
```

- (c) [7 points] Give an implementation of the type checker:

```
Fixpoint type_check (Gamma : stringmap ty) (e : expr) : option ty :=
  (* FILL OUT *)
```

Your type checker should satisfy the following correctness lemma:

```
Lemma type_check_correct Gamma e A :
  typed Gamma e A <-> exists B, type_check Gamma e = Some B /\ subty B A.
```

You are recommended to use the usual operations on finite maps and the `bsubty` function (even if you did not complete Question 1.b).

**Solution:**

```
Fixpoint type_check (Gamma : stringmap ty) (e : expr) : option ty :=
  match e with
  | ENat (0 | 1) => Some TBool
  | ENat _ => Some TNat
  | EVar x => StringMap.lookup Gamma x
  | ELam x A e =>
```

```

    match type_check (StringMap.insert x A Gamma) e with
    | Some B => Some (TFun A B)
    | None => None
    end
  | EApp e1 e2 =>
    match type_check Gamma e1, type_check Gamma e2 with
    | Some (TFun A1 A2), Some B =>
      if bsubty B A1 then Some A2 else None
    | _, _ => None
    end
  end.
end.

```

- (d) [2 points] Explain whether it is possible to define a type checker that satisfies:

**Lemma** `type_check_correct_strong` `Gamma e A :`  
`typed Gamma e A <-> type_check Gamma e = Some A.`

**Solution:** This is impossible. Types are not unique, the expression `ENat 0` has type `TBool` and `TNat`. The theorem thus gives us `type_check [] (ENat 0) = Some TBool` and `type_check [] (ENat 0) = Some TNat`, hence `TBool = TNat`, which is contradictory.

- (e) [5 points] We extend our language with an if-then-else construct:

```

Inductive expr :=
  (* same as before *)
  | EIf (e1 e2 e3 : expr).

Inductive typed : stringmap ty -> expr -> ty -> Prop :=
  (* same as before *)
  | If_typed Gamma e1 e2 e3 B :
    typed Gamma e1 TBool ->
    typed Gamma e2 B ->
    typed Gamma e3 B ->
    typed Gamma (EIf e1 e2 e3) B.

```

Give the additional Coq code that needs to be added to the `type_check` function (you do not need to copy the entire prior code). Your new `type_check` function needs to satisfy the lemma `type_check_correct` from Question 1.c.

You are allowed to introduce an auxiliary function on `ty`. You are not required to give the implementation of that function, but you should describe clearly its behavior in English.

**Solution:**

```

Fixpoint type_check (Gamma : stringmap ty) (e : expr) : option ty :=
  match e with
  (* same as before *)
  | EIf e1 e2 e3 =>
    match type_check Gamma e1, type_check Gamma e2, type_check Gamma e3 with
    | Some TBool, Some A1, Some A2 => join A1 A2
    | _, _, _ => None
    end
  end

```

```
    end  
end.
```

The function `join A1 A2` computes the least supertype of `A1` and `A2`, and returns `None` if no such supertype exists.

**Question 2 (13 points)**

Consider a small programming language, which, similar to Rust, has a `panic` expression that safely terminates execution of the program:

$$\begin{aligned}
 v &::= n \mid b & (n \in \mathbb{N}, b \in \mathbb{B}) \\
 \odot &::= + \mid * \mid = \\
 e &::= x \mid v \mid \text{let } x = e_1 \text{ in } e_2 \mid e_1 \odot e_2 \mid \text{panic}
 \end{aligned}$$

The semantics of the language is given using a small-step operational semantics. Its head reduction relation  $\Rightarrow_h$  and whole-program reduction relation  $\Rightarrow$  are inductively defined as:

$$\frac{}{(\text{let } x = v \text{ in } e) \Rightarrow_h \text{subst } x \ v \ e} \quad \frac{\text{eval\_bin.op } \odot \ v_1 \ v_2 = w}{(v_1 \odot v_2) \Rightarrow_h w} \quad \frac{e_1 \Rightarrow_h e_2 \quad \text{ctx } k}{k \ e_1 \Rightarrow k \ e_2}$$

Note that `panic` does not have a reduction rule!

Evaluation contexts are defined as follows:

```

Inductive ctx : (expr -> expr) -> Prop :=
| Let_ctx y e2 : ctx (fun x => ELet y x e2)
| Op_r_ctx op e1 : ctx (EOp op e1)
| Op_l_ctx op v2 : ctx (fun x => EOp op x (EVal v2))
| Id_ctx : ctx (fun x => x)
| Compose_ctx k1 k2 : ctx k1 -> ctx k2 -> ctx (fun x => k1 (k2 x)).

```

- (a) [5 points] Define a predicate `safe : expr → Prop` that describes whether an expression is safe. Explain why your definition correctly captures safety.

Examples of **safe** expressions:

- `panic`
- `2 + panic`
- `(5 + true) + panic`, due to right-to-left evaluation `panic` is executed before `5 + true`

Examples of **unsafe** expressions:

- `5 + true`
- `panic + (5 + true)`, due to right-to-left evaluation `5 + true` is executed before `panic`

**Solution:**

```

Definition safe (e : expr) :=
  forall e',
    steps e e' ->
      (exists v, e' = EVal v) /\
      (exists k, ctx k /\ e' = k EPanic) /\
      (exists e'', step e' e'').

```

- (b) [3 points] Give the (Coq) type for an interpreter for this language. The interpreter should given a expression be able to tell whether it safely results in a value, panics, or is unsafe. You do not have to give the implementation of the interpreter.

**Solution:**

```
Inductive res := Panic | Wrong | Res (v : val).  
Fixpoint interp (e : expr) : res. := (* .. *).
```

(c) [5 points] Give the correctness lemma for your interpreter.

**Solution:** A possible solution is:

```
Lemma interp_correct e :  
  (forall v, interp e = Res v <-> steps e (Eval v)) /\  
  (interp e = Panic <-> exists k, ctx k /\ steps e (k EPanic)).
```

Note that this lemma uniquely specifies `interp` and we therefore do not need to add a `Wrong` case. There are many other solutions that uniquely specify `interp`.

**Question 3 (12 points)**

Consider a Rust library that provides the type `Foo<T>` with the following methods:

```
Foo::new(val : T) -> Foo<T>
Foo::replace(&self, val : T) -> T
```

The `new` method wraps a value into a `Foo`. The `replace` method allows one to replace the value of a `Foo` via a shared reference: it “moves out” the old value, and “moves in” the new value. A sample program that uses this library is as follows:

```
fn main() {
    let x = Foo::new(10);
    let r1 = &x;
    let r2 = &x;
    println!("{}", r2.replace(12)); // Prints 10
    println!("{}", r1.replace(14)); // Prints 12
}
```

- (a) [3 points] Consider the following additional method for `Foo<T>` that retrieves the value:

```
Foo::get(&self) -> T
```

Explain whether safety of Rust is preserved by adding the `get` method. If not, your explanation should include a well-typed Rust program that uses the `get` method, but has a memory error (*e.g.*, use-after-free).

**Solution:** It is only safe if `T` is `Copy`. Particularly it is not safe if `T` is a vector:

```
let x = Foo::new(vec![10,11]);
let v1 = (&x).get();
let v2 = (&x).get();
v1.push(12); // Reallocates the vector
println!("{}", v2.get(1)); // Use-after-free
```

- (b) [3 points] Consider the following additional method for `Foo<T>` that overwrites the value:

```
Foo::set(&self, value : T) -> ()
```

Explain whether safety of Rust is preserved by adding the `set` method. If not, your explanation should include a well-typed Rust program that uses the `get` method, but has a memory error (*e.g.*, use-after-free).

**Solution:** This function is safe. It is the same as calling `replace` and dropping its return value.

- (c) [3 points] Explain whether `Foo<T>` could be `Send`. (Your answer is allowed to depend on whether `T` is `Send` or `Sync`.)

**Solution:** The type `Foo<T>` is `Send` provided that `T` is `Send`. In case one has unique ownership of a `Foo<T>`, there cannot be any shared references to it. Hence it can be safely transferred to another thread without causing data races.



- (d) [3 points] Explain whether `Foo<T>` could be `Sync`. (Your answer is allowed to depend on whether `T` is `Send` or `Sync`.)

**Solution:** The type `Foo<T>` cannot be `Sync`. If it was `Sync`, one could share `&Foo<T>` between threads, and have racy calls to `replace`. Unless `replace` uses atomic operations or performs some form of synchronization, such races would be unsafe.

**Question 4 (14 points)**

We consider substructural typing for a programming language with references. Consider the following semantic interpretation for the typing judgment:

$$\Gamma \vdash e : A \triangleq \forall \vec{v}. \text{ctx\_typed } \Gamma \vec{v} \vdash \text{WP} (\text{subst\_map } \vec{v} e) [v. A v * \text{True}]$$

$$\text{ctx\_typed } \Gamma \vec{v} \triangleq \begin{cases} \text{emp} & \text{if } \Gamma = [] \\ \exists v. \text{lookup } \vec{v} x = \text{Some } v * A v * \text{ctx\_typed } \Gamma' \vec{v} & \text{if } \Gamma = (x, A) :: \Gamma' \end{cases}$$

Here,  $\vec{v}$  ranges over finite maps from variables names to values,  $\text{subst\_map } \vec{v} e$  is the parallel substitution, and  $\text{ctx\_typed } \Gamma \vec{v}$  is the semantic context typing. Contexts  $\Gamma$  are lists that associate types to variable names (*i.e.*, they can have multiple bindings for the same variable).

- (a) [4 points] Explain whether the “weakening rule” holds in a type system with the above semantic interpretation. Recall, the weakening rule is as follows:

$$\frac{\Gamma \vdash e : B \quad x \notin \text{dom } \Gamma}{x : A, \Gamma \vdash e : B}$$

**Solution:** The weakening rule holds in this system. We can turn  $\text{ctx\_typed } \Gamma (x, A) :: \vec{v}$  into  $\text{True} * \text{ctx\_typed } \Gamma \vec{v}$ , and then frame  $\text{True}$  into the  $\text{True}$  in the postcondition.

- (b) [4 points] Explain whether the “contraction rule” holds in a type system with the above semantic interpretation? Recall, the contraction rule is as follows:

$$\frac{x : A, x : A, \Gamma \vdash e : B \quad x \notin \text{dom } \Gamma}{x : A, \Gamma \vdash e : B}$$

**Solution:** The contraction rule does not hold in this system. The variable  $x : A$  in the context provides unique ownership (of, for example, a reference) and therefore in general not be duplicated.

- (c) [6 points] State the “type safety” theorem for linear languages from week 12, and explain which parts of this theorem **do** and **do not** hold for the above semantic interpretation.

**Solution:** The type safety theorem says: If  $\emptyset \vdash e : A$  for a copy-type  $A$ , then

1.  $e$  does not have memory errors, *i.e.*, use-after-free and double-free, and
2.  $e$  does not have memory leaks.

Using the weakening rule, we can ‘leak’ references, so property (2) does not hold.

**Question 5 (15 points)**

For each question you should explain your answer!

- (a) [3 points] State precisely the set of heaps described by  $\exists m. l_1 \mapsto 1 \wedge l_2 \mapsto m$ .

**Solution:** This assertion describes the heap consisting of a single location  $l_1$  (which equals  $l_2$ ) containing 1. The conjunction ensures that  $l_1$  and  $l_2$  are the same, and the existentially quantified value  $m$  equals 1.

- (b) [3 points] State precisely the set of heaps described by  $l \mapsto 5 * (l \mapsto 10 \multimap \text{True})$ .

**Solution:** This assertion describes the heaps that contain at least location  $l$  with value 5. That is because  $l \mapsto 10 \multimap \text{True}$  is logically equivalent to  $\text{True}$ , so the whole assertion describes heaps that can be split into a part with location  $l$  and an arbitrary part.

- (c) [3 points] State precisely the set of heaps described by  $l \mapsto 5 * (l \mapsto 10 \multimap \text{False})$ .

**Solution:** This assertion describes the empty set of heaps. The separating conjunction ensures the heap can be split into two disjoint parts: one with exactly location  $l$  and another part without location  $l$ . No heap without location  $l$  satisfies  $l \mapsto 10 \multimap \text{False}$  because any such heap can be extended with location  $l$ , but none satisfies  $\text{False}$ .

- (d) [3 points] Give a separation logic assertion that describes the heaps where all locations have a value that is unequal to 10.

**Solution:**  $\forall l. l \mapsto 10 * \text{True} \rightarrow \text{False}$ . This assertion says that if the heap contains *at least* a location  $l$  with value 10, we have a contradiction.

- (e) [3 points] Give a program  $e$  that satisfies:

$$[\exists k. l \mapsto k * k \mapsto l] e [w. w = ()]$$

(Recall that the separation logic assertion  $x = y$  is only satisfied when the heap is empty.)

**Solution:** Let  $e \triangleq \text{free}(\text{free } l); ()$ . There is a reference to a reference, so we need to free both. We assume that  $\text{free}$  returns the value of the reference that is freed.

**Question 6 (15 points)**

Consider a Coq inductive type for binary trees and a function that computes the depth:

```
Inductive tree :=
| leaf : nat -> tree
| node : tree -> tree -> tree.

Fixpoint depth_coq (t : tree) : nat :=
  match t with
  | leaf _ => 0
  | node t1 t2 => S (Nat.max (depth_coq t1) (depth_coq t2))
  end.
```

And an imperative version of the depth function:

```
Definition depth :=
  recclosure: "rec" "l" "k" "d" =>
    match: !"l" with
    | InjL "n" =>
      if: !"k" <: "d" then
        "k" <- "d";; VUnit
      else VUnit
    | InjR "node" =>
      let: "ll" := EFst "node" in
      let: "lr" := ESnd "node" in
      "rec" "ll" "k" ("d" +: VNat 1);;
      "rec" "lr" "k" ("d" +: VNat 1)
    end.
```

- (a) [4 points] Define a representation predicate `is_tree l t` in separation logic that states that at location  $l$  there is a mutable tree that matches up with the inductive Coq tree  $t$ . The type of `is_tree` should be  $\text{loc} \rightarrow \text{tree} \rightarrow \text{sepProp}$ . You can use either math or Coq syntax. You are allowed to ignore/leave implicit coercions such as `VNat` and `VRef`.

**Solution:**

```
Fixpoint is_tree (l : loc) (t : tree) : sepProp :=
  match t with
  | leaf v => l ~> VInjL (VNat v)
  | node t1 tr =>
    Ex ll lr,
    l ~> VInjR (VPair (VRef ll) (VRef lr)) **
    is_tree ll t1 **
    is_tree lr tr
  end.
```

- (b) [4 points] Our goal is to prove the following Hoare triple:

$$[\text{is\_tree } l \ t \ * \ k \mapsto 0] \text{depth } l \ k \ 0 [w. w = () \ * \ \text{is\_tree } l \ t \ * \ k \mapsto (\text{depth\_coq } t)]$$

To prove this Hoare triple by induction you need a strengthened version that provides a sufficient induction hypothesis. Give the strengthened version of the Hoare triple.

**Solution:**

$$[\text{is\_tree } l \ t * k \mapsto m] \text{depth } l \ k \ d \left[ \begin{array}{l} w. w = () * \text{is\_tree } l \ t * \\ k \mapsto m \max(d + \text{depth\_coq } t) \end{array} \right]$$

(c) [7 points] Give a proof outline for your strengthened version of the Hoare triple.

**Solution:**

```

depth l k d  $\triangleq$  [is_tree l t * k  $\mapsto$  m]
  match !l with
  | inl n       $\Rightarrow$  [l  $\mapsto$  inl n * k  $\mapsto$  m]  t = leaf n
    if !k < d then
      [l  $\mapsto$  inl n * k  $\mapsto$  m * m < d]
      k  $\leftarrow$  d; ()
      [l  $\mapsto$  inl n * k  $\mapsto$  d]
      [l  $\mapsto$  inl n * k  $\mapsto$  m max d]
    else
      [l  $\mapsto$  inl n * k  $\mapsto$  m * m  $\geq$  d]
      ()
      [l  $\mapsto$  inl n * k  $\mapsto$  m max d]
      [l  $\mapsto$  inl n * k  $\mapsto$  m max d]
      [l  $\mapsto$  inl n * k  $\mapsto$  m max(d + depth_coq t)]
  | inr node  $\Rightarrow$  [l  $\mapsto$  inr (l_l, l_r) * is_tree l_l t_l * is_tree l_r t_r * k  $\mapsto$  m]
    t = node t_l t_r, node = (l_l, l_r)
    let ll = fst node in ll = l_l
    let lr = snd node in lr = l_r
    depth ll k (d + 1);
    [w. w = () * l  $\mapsto$  inr (l_l, l_r) * is_tree l_l t_l * is_tree l_r t_r *
    [k  $\mapsto$  m max(d + 1 + depth_coq t_l)
    depth lr k (d + 1)
    [w. w = () * l  $\mapsto$  inr (l_l, l_r) * is_tree l_l t_l * is_tree l_r t_r *
    [k  $\mapsto$  m max(d + 1 + depth_coq t_l) max(d + 1 + depth_coq t_r)
    [w. w = () * l  $\mapsto$  inr (l_l, l_r) * is_tree l_l t_l * is_tree l_r t_r *
    [k  $\mapsto$  m max(d + depth_coq t)
    [w. w = () * is_tree l t * k  $\mapsto$  m max(d + depth_coq t)
    end
    [w. w = () * is_tree l t * k  $\mapsto$  m max(d + depth_coq t)]

```