

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.

- (a) [4 points] Explain the difference between eq and eqb . You should give the formal equivalence between eq and eqb as a Lemma in Coq syntax.

Solution: The notion eq is a mathematical statement and the notion eqb is an algorithm. The formal correspondence is as follows:

```
Lemma eqb_correct : forall x1 x2,
  eq x1 x2 <-> eqb x1 x2 = true.
```

- (b) [2 points] Explain the type $\text{forall } x1 \ x2 : A, \{ \text{eq } x1 \ x2 \} + \{ \sim \text{eq } x1 \ x2 \}$ of dec .

Solution: The type $\{ \text{eq } x1 \ x2 \} + \{ \sim \text{eq } x1 \ x2 \}$ is a sumbool. A sumbool is similar to the Boolean type, but the constructors `left Heq` and `right Hneq` are annotated with a proof of equality $\text{Heq} : \text{eq } x1 \ x2$, respectively, inequality $\text{Hneq} : \sim \text{eq } x1 \ x2$.

- (c) [3 points] Define eqb in terms of dec . Explain how to prove the Lemma of Question 1.a for your definition of 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.

Solution:

```
Definition eqb : A -> A -> bool := fun x1 x2 =>
  if dec x1 x2 then true else false.
```

In order to prove the lemma `eqb_correct`, we unfold the definition of `eqb` and then perform a case analysis (*i.e.*, `destruct`) on `dec x1 x2`. This leads to the following two goals, which are both tautologies:

- $\text{eq } x1 \ x2 \leftrightarrow \text{true} = \text{true}$ with hypothesis $\text{Heq} : \text{eq } x1 \ x2$
- $\text{eq } x1 \ x2 \leftrightarrow \text{false} = \text{true}$ with hypothesis $\text{Hneq} : \sim \text{eq } x1 \ x2$

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.

```

- (a) [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.

Solution: The expression `EEither e1 e2` non-deterministically evaluates to `e1` or `e2`. A sample is `e := EEither (EVal (VBool true)) (EVal (VBool false))`, which tosses a coin. We have `steps e (EVal (VBool true))` and `steps e (EVal (VBool false))`.

- (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.

```

Solution:

```

Fixpoint interp (e : expr) : list (option val) :=
  match e with
  | EVal v => [Some v]
  | EEither e1 e2 => interp e1 ++ interp e2
  | EIf e e1 e2 => flat_map (fun v =>
    match v with
    | Some (VBool b) => interp (if b then e1 else e2)
    | _ => [None]
    end) (interp e)
  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.

Solution:

```

Definition unsafe e :=
  exists e', steps e e' /\ ~(exists v, e' = EVal v) /\ ~(exists e'', step e' e'').

Lemma interp_Some e v : In (Some v) (interp e) <-> steps e (EVal v).
Lemma interp_None e : In None (interp e) <-> unsafe e.

```

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.

Solution: A type is `Copy` if it can be duplicated through a shallow copy, and `Clone` if it can be duplicated through a deep copy. Every type that is `Copy` is also `Clone`. The inclusion does not hold in the other way: `Vec<i32>` is `Clone`, not `Copy`.

- (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.

Solution: The Rust type checker accepts this definition. Since `i32` is `Copy`, we do not have to give up ownership of the vector `v` when accessing elements.

(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.

Solution:

```
fn swap2<T : Clone>(v : &mut Vec<T>, i : usize, j : usize) {
    let x = v[i].clone();
    v[i] = v[j].clone();
    v[j] = x
}
```

When accessing a vector element `v[k]`, one has to give up ownership of the entire vector `v` for the lifetime of the element. Since we need to mutate the vector `v` during the lifetime of both `v[i]` and `v[j]`, we need to clone the elements.

Additional/alternative answer: One could use `std::mem::swap` from the Rust standard library (which itself is implemented using unsafe code), which is more efficient, and avoids the need for the type to be `Clone`.

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 tid , and functions $\lambda x. e$. Configurations σ 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 σ can step to configuration σ' :

$$\frac{\sigma(i) = k((\lambda x. e) v) \quad \text{ctx } k}{\sigma \Rightarrow_i \sigma[i := k(\text{subst } x v e)]} \quad \frac{\sigma(i) = k(\text{spawn } e) \quad tid = \text{length } \sigma \quad \text{ctx } k}{\sigma \Rightarrow_i (\sigma[i := k tid] ++ [e])}$$

$$\frac{\sigma(i) = k(\text{join } tid) \quad \sigma(tid) = e \quad e \notin \text{Val} \quad \text{ctx } k}{\sigma \Rightarrow_i \sigma}$$

$$\frac{\sigma(i) = k(\text{join } tid) \quad \sigma(tid) = v \quad v \in \text{Val} \quad \text{ctx } k}{\sigma \Rightarrow_i \sigma[i := k v]}$$

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 σ , and the value e is stored at position i in the list σ), and the notation $\sigma[i := e]$ to overwrite the i th element of the list σ 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.

Solution: Assume $\emptyset \vdash e : A$, we need to prove $\text{safe } [e]$. That means we should assume that $[e] \Rightarrow^* \sigma'$, and need to prove that all threads in σ' are a value or can step. Given $\emptyset \vdash e : A$, by (1) we have $\vdash_{\text{cfg}} [e] : A$. By induction on $[e] \Rightarrow^* \sigma'$, we obtain by (2) that $\vdash_{\text{cfg}} \sigma' : A$. By (3) this gives that all threads in σ' are a value or can step.

(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, Σ 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.

Solution:

$$\begin{array}{c}
\frac{\Gamma(x) = A}{\Sigma \mid \Gamma \vdash x : A} \quad \frac{n \in \mathbb{N}}{\Sigma \mid \Gamma \vdash n : \mathbf{nat}} \quad \frac{\Sigma(tid) = A}{\Sigma \mid \Gamma \vdash tid : \mathbf{joinhandle} A} \\
\\
\frac{\Sigma \mid \Gamma, x : A \vdash e : B}{\Sigma \mid \Gamma \vdash \lambda x. e : A \rightarrow B} \quad \frac{\Sigma \mid \Gamma \vdash e_1 : A \rightarrow B \quad \Sigma \mid \Gamma \vdash e_2 : A}{\Sigma \mid \Gamma \vdash e_1 e_2 : B} \\
\\
\frac{\Sigma \mid \Gamma \vdash e : A}{\Sigma \mid \Gamma \vdash \mathbf{spawn} e : \mathbf{joinhandle} A} \quad \frac{\Sigma \mid \Gamma \vdash e : \mathbf{joinhandle} A}{\Sigma \mid \Gamma \vdash \mathbf{join} e : A}
\end{array}$$

- (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.

Solution: A possible definition of $\vdash_{\text{cfg}} \sigma : A$ is as follows:

$$\begin{aligned}
\vdash_{\text{cfg}} \sigma : A \triangleq & \exists \Sigma. \text{length } \Sigma = \text{length } \sigma \wedge \\
& \Sigma(0) = A \wedge \\
& (\forall i < \text{length } \sigma. \Sigma \mid \Gamma \vdash \sigma(i) : \Sigma(i))
\end{aligned}$$

The three properties in Question 4.a hold because:

- Initialization. Let $\Sigma = [A]$. We prove that $\Gamma \vdash e : A$ implies $\Sigma \mid \Gamma \vdash e : A$.
- Preservation. We need to prove that after a step of reduction, we can find a new thread typing Σ , and all resulting threads are typed in that Σ . Such a Σ exists because the types of threads do not change, only new threads can be added.
- Progress: The definition of $\vdash_{\text{cfg}} \sigma : A$ tells us that each thread is run-time typed. By the definition of the run-time judgment for expressions, we obtain that each threads is either a value or can step.

- (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 σ that is deadlocked, but for which **safe** σ holds. Explain your answer.

Solution: Take **[join 1; join 0]**. The first thread will spin indefinitely long waiting for the second, and the second will spin indefinitely long waiting for the first. This configuration is safe, because spinning is considered to be a safe behavior.

- (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.

Solution: The type system rules out deadlocks.

In a deadlocked configuration, threads will be waiting for each other in a cycle, see the answer of Question 4.d for example. While such a configuration can be typed using the run-time typing judgment, it cannot be typed using the static typing judgment.

Deadlocked configurations can never be reached from a statically typed program, since a thread can only “join” another thread that has been spawned earlier.

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] $l \mapsto n * \neg(l \mapsto n)$

Solution: This proposition describes the set of heaps that contain at least the location l with value n . Due to the separating conjunction $*$, the heap should be split up into a part containing l with value n , and an arbitrary other part that does not contain l .

- (b) [3 points] $l \mapsto n \wedge \neg(l \mapsto n)$

Solution: This proposition is logically equivalent to \perp , it thus describes the empty set of heaps. Note that $P \wedge \neg P$ is always false, since no proposition can hold and not hold at the same time for the same heap.

- (c) [3 points] $(l \mapsto n * \top) \wedge (k \mapsto m * \top)$

Solution: This proposition describes the heaps at least the locations l and k with values n and m , respectively. Due to the conjunction \wedge , the locations l and k may be the same.

- (d) [3 points] $l \mapsto n \wedge k \mapsto (-n)$

Solution: The proposition describes the heap with location l , which is equal to k , with value 0. Due to the conjunction \wedge the locations l and k should be the same. Moreover, this location can only hold the values n and $-n$ at the same time if $n = 0$.

- (e) [3 points] $l \mapsto n * ((k \mapsto m) \rightarrow \text{emp})$

Solution: This proposition describes the set of heaps that contain at least location l with value n . The additional part of the heap should not contain exactly location k with value m (it might contain location k with another value than m). Due to the separating conjunction $*$, the heap should be split up into a part containing l with value n , and an other part satisfying $(k \mapsto m) \rightarrow \text{emp}$. The proposition $(k \mapsto m) \rightarrow \text{emp}$ describes all heaps except those with location k with value m .

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, Γ_1 is the *pre-typing context* and Γ_2 is the *post-typing context*. The idea is that the post-typing context Γ_2 contains the variables from Γ_1 that are not used by the expression e . Examples of typing rules are:

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

- (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.

Solution:

$$\Gamma_1 \vdash e : A \dashv \Gamma_2 \quad \text{iff} \quad \exists \Gamma. \Gamma_1 = \Gamma ++ \Gamma_2 \wedge \Gamma \vdash e : A$$

- (b) [3 points] Give the typing rule for `let $x = e_1$ in e_2` using the typing judgment with two contexts.

Solution:

$$\frac{\Gamma_1 \vdash e_1 : A \dashv \Gamma_2 \quad \Gamma_2, x : A \vdash e_2 : B \dashv \Gamma_3 \quad x \notin \text{dom } \Gamma_2}{\Gamma_1 \vdash \text{let } x = e_1 \text{ in } e_2 : B \dashv \Gamma_3}$$

- (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 λ -expressions/types of λ -bound variables.)

Solution: If you make a type checker inspired by the one-context judgment, you end up with a signature such as the following:

```
Fixpoint typecheck : ctx -> term -> expr -> option ty := ...
```

The challenge is type checking binary constructs such as function application and arithmetic operators. There you need to split the context Γ into parts Γ_1 and Γ_2 with $\Gamma = \Gamma_1 ++ \Gamma_2$, and then check the operands in Γ_1 and Γ_2 , respectively. Since there are exponentially many splittings of Γ , this quickly becomes infeasible.

If you make a type checker inspired by the two-context judgment, you end up with a signature such as the following:

```
Fixpoint typecheck : ctx -> term -> expr -> option (ctx * ty) := ...
```

Now you can type check the operands of binary constructs in sequence. You can type check the first operand first, and then use the post-typing context to type check the second operand.

- (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.

Solution:

$$\Gamma_1 \vdash e : A \dashv \Gamma_2 \triangleq \forall \vec{v}. \text{ctx_typed } \Gamma_1 \vec{v} \vdash \text{WP } \text{subst_map } \vec{v} e \{ v. A v * \text{ctx_typed } \Gamma_2 \vec{v} \}$$

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 γ 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) |
| $\text{True} \multimap \exists \gamma. \gamma \hookrightarrow_\bullet n * \gamma \hookrightarrow_\circ n$ | (Ghost-alloc) |
| $\gamma \hookrightarrow_\bullet n * \gamma \hookrightarrow_\circ m \multimap n = m$ | (Ghost-agree) |
| $\gamma \hookrightarrow_\bullet n * \gamma \hookrightarrow_\circ m \multimap \exists \gamma'. \gamma \hookrightarrow_\bullet n' * \gamma \hookrightarrow_\circ n'$ | (Ghost-update) |

- (a) [3 points] What are the possible behaviors of $\text{foo } n$, where $n \in \mathbb{Z}$? Give a Hoare triple for $\text{foo } n$ that exactly describes the safe behaviors. Give an intuitive explanation for why this Hoare triple holds.

Solution: The program is safe if n is at least (or equal to) 30. The Hoare triple is:

$$\{ 30 \leq n \} \text{foo } n \{ w. w = () \}$$

This Hoare triple holds intuitively because there are two possible execution orders. Either the first thread acquires the lock, and then the second thread, or vice versa. If the first thread acquires the lock first, the program will crash if n is smaller than 10 because the sub-expression $1 + ()$ is executed. If the second thread acquires the lock first, it will decrement n with 20 first. Hence the program will crash if n is smaller than 30.

- (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.

Solution: (Many alternative solutions of the following two sub-exercises are possible.) We use two ghost variables γ_1 and γ_2 to keep track of the number of decrements by the first and second thread:

$$R \triangleq \exists k_1, k_2 : \mathbb{N}. r \mapsto ((n - 30) + k_1 + k_2) * \gamma_1 \hookrightarrow_\bullet k_1 * \gamma_2 \hookrightarrow_\bullet k_2$$

The ghost values describe the remaining decrements each thread is allowed to perform. The initial values of the ghost variables should be 10 and 20.

- (c) [9 points] Give a proof outline for your Hoare triple for the function `foo n`.

Solution:

```

{ 30 ≤ n }
let r = alloc n in
{ r ↦ n }
{ r ↦ n * γ1 ↦• 10 * γ1 ↦• 10 * γ2 ↦• 20 * γ2 ↦• 20 }
{ R * γ1 ↦• 10 * γ2 ↦• 20 }
let lk = newlock () in
{ isLock lk R * γ1 ↦• 10 * γ2 ↦• 20 }
(
  acquire lk;
  r := !r - 10;
  if (!r) < 0 then 1 + () else release lk
  ||
  acquire lk;
  r := !r - 20;
  release lk
);
()
{ w. w = () }
```

The proof outline for the first thread is:

```

{ isLock lk R * γ1 ↦• 10 }
acquire lk;
{ isLock lk R * γ1 ↦• 10 * r ↦ ((n - 30) + k1 + k2) * γ1 ↦• k1 * γ2 ↦• k2 }
{ isLock lk R * γ1 ↦• 10 * r ↦ ((n - 30) + 10 + k2) * γ1 ↦• 10 * γ2 ↦• k2 }
r := !r - 10;
{ isLock lk R * γ1 ↦• 10 * r ↦ ((n - 30) + 0 + k2) * γ1 ↦• 10 * γ2 ↦• k2 }
{ isLock lk R * γ1 ↦• 0 * r ↦ ((n - 30) + 0 + k2) * γ1 ↦• 0 * γ2 ↦• k2 }
if (!r) < 0 then 1 + ()
{ isLock lk R * γ1 ↦• 0 * r ↦ ((n - 30) + 0 + k2) * γ1 ↦• 0 * γ2 ↦• k2 }
{ isLock lk R * γ1 ↦• 0 * R }
else release lk
{ isLock lk R * γ1 ↦• 0 }
```

The “then” branch with $1 + ()$ is never executed because $0 \leq (n - 30) + 0 + k_2$. That inequality holds because $30 \leq n$ and $k_2 \in \mathbb{N}$.

The proof outline for the second thread is:

```

{ isLock lk R *  $\gamma_2 \hookrightarrow_o 20$  }
acquire lk;
{ isLock lk R *  $\gamma_2 \hookrightarrow_o 20 * r \mapsto ((n - 30) + k_1 + k_2) * \gamma_1 \hookrightarrow_\bullet k_1 * \gamma_2 \hookrightarrow_\bullet k_2$  }
{ isLock lk R *  $\gamma_2 \hookrightarrow_o 20 * r \mapsto ((n - 30) + k_1 + 20) * \gamma_1 \hookrightarrow_\bullet k_1 * \gamma_2 \hookrightarrow_\bullet 20$  }
r := !r - 20;
{ isLock lk R *  $\gamma_2 \hookrightarrow_o 20 * r \mapsto ((n - 30) + k_1 + 0) * \gamma_1 \hookrightarrow_\bullet k_1 * \gamma_2 \hookrightarrow_\bullet 20$  }
{ isLock lk R *  $\gamma_2 \hookrightarrow_o 0 * r \mapsto ((n - 30) + k_1 + 0) * \gamma_1 \hookrightarrow_\bullet k_1 * \gamma_2 \hookrightarrow_\bullet 0$  }
else release lk
{ isLock lk R *  $\gamma_2 \hookrightarrow_o 0$  }

```