Branded types

Final exam, MPRI 2–4

Wednesday 8th March, 2023 — Duration: 2h45

Answers are judged by their correctness, but also by their clarity, conciseness, and accuracy. You don't have to justify answers unless explicitly required. Although the questions are in English, it is permitted to answer in either French or English—and recommended to answer in French if this is your mother language. Questions are in logical order, and many questions do not depend on previous ones.

1 A library of arrays without bounds checks at accesses

We consider the following module interface, written in OCaml:

```
module BrandedArray = sig
  type ('a, 'g) t
  type 'g idx

  val idx_of_int: ('a, 'g) t -> int -> 'g idx option
  val get: ('a, 'g) t -> 'g idx -> 'a
  val set: ('a, 'g) t -> 'g idx -> 'a -> unit

  type ('a, 'ret) callback = { cb: 'g. ('a, 'g) t -> 'ret }
  val with_array: int -> 'a -> ('a, 'ret) callback -> 'ret
end
```

This interface contains two types: the type `('a, 'g) t` of *branded arrays* and the type `'g idx` of *branded indices*. Operationally, branded arrays are just arrays and indices are just integers; but the nature of these values is hidden behind abstract types. The functions `get` and `set` can be used to access an array at a given index. The function `idx_of_int` creates an index out of an integer, checking that the given integer is within the array bounds, and returns `None` if the check fails.

Both the type of arrays and of indices are *branded*: they are parameterized by a type `'g`. This type is a phantom type: an entity used syntactically as a type in the language, but which we never use as the type of a term. The role of this type parameter is to guarantee that an index is used to access only the array with which it has been created, thus guaranteeing that `get` and `set` are always given an index which is valid for the array.

Array creation is done through the function `with_array`, which initializes a new array of a given size with the given initial value, then calls a user-provided callback with that array, and finally returns the value returned by the user. The user-provided function is wrapped in the `callback` record type because it needs to be polymorphic over the phantom type `'g`: the function `with_array` is an example of rank-2 polymorphism, which, in OCaml, is only supported through polymorphic record types. The syntax `cb: 'g. ('a, 'g) t -> 'ret` indicates that `cb` is a polymorphic field, quantified over the type variable `'g`.

Question 1 *Write a implementation module for the signature `BrandedArray`.* Answer

□

Question 2 *What would happen if we added a function `make: int -> 'a -> ('a, 'g) t`, similarly to the function `Array.make` of the OCaml standard library?* Answer

□

Question 3 *Write a function* `BrandedArray.make` *that creates a new array, but returns it using a GADT (which you will define) instead of using rank-2 polymorphism. What is its signature?*
Answer
□

We now consider an implementation of `BrandedArray` where `get` and `set` do not perform bounds checks[1]. This implementation is still safe, thanks to the guarantee stated above.

Question 4 *Write a function that takes as parameters a branded array and two integers (of type* `int`*), and exchanges the values at these indices in the array. This function should only use the functions provided in the* `BrandedArray` *module interface, and do as few bounds checks as possible. Compare the number of bounds checks this function does with an analogous function written with the OCaml standard library.*
Answer
□

Question 5 *The module* `Array` *of the standard library of OCaml contains a function* `map: ('a -> 'b) -> 'a array -> 'b array`*. We consider adding to* `BrandedArray` *an analogous function* `map: ('a -> 'b) -> ('a, 'g) t -> ('b, 'g) t`*. Would it break safety? If yes, justify. Otherwise, give a counter-example.*
Answer
□

2 Soundness proof

The goal of this section is to prove the soundness of the approach based on branded types that was introduced in the previous section. Our formal setting will be System F extended with branded persistent arrays (see fig. 1). This extension introduces a dedicated construct to represent (persistent) arrays $[\bar{v}]$, which takes a sequence $\bar{v}$ of values, two syntactic constructs to operate on arrays ($e.(e)$ and $e[e \leftarrow e]$) and two primitive functions (`idx_of_int` and `with_array`).

The operational semantics admits the following properties:

- values never reduce;
- for all E, e_1 and e, if e_1 is not a value and $E[e_1] \longrightarrow e$, then there exists e_2 such that $e_1 \longrightarrow e_2$ and $e = E[e_2]$;
- the semantics is deterministic.

These properties are taken for granted: no need to prove them, but you should clearly mention them when they are necessary for a proof to hold.

It is recommended to read the operational semantics and typing rules before answering the next questions.

Question 6 *Compared to the module signature studied in the first part of the exam, identify the fundamental differences introduced by this idealized calculus. In what respect do they simplify the formalization?*
Answer
□

Question 7 *Formally state (without proving it) the subject reduction property for this calculus. Prove that it does* ***not*** *hold.*
Answer
□

As a consequence, it is not possible to use a *direct* syntactic proof to prove type soundness. Instead, we will pursue a semantic proof based on a unary logical relation, which gives a semantic interpretation to types.

The logical relation is defined in fig. 1. For a given interpretation of type variables ρ, the interpretation $[\![\tau]\!]^\rho$ of a type τ consists in two components:

- $[\![\tau]\!]^\rho$.**vals** is a set of values semantically well-behaved at type τ;

[1] This implementation can, for example, use unsafe array access functions `Array.unsafe_get` and `Array.unsafe_set` of the standard library. These accesses have undefined behavior if given invalid indices.

Syntax:

$$\begin{aligned}
\tau &::= \alpha \mid \tau \to \tau \mid \forall\alpha.\,\tau \mid \texttt{int} \mid (\tau,\tau)\ \texttt{array} \mid \tau\ \texttt{index} \\
e &::= x \mid n \mid e\,e \mid \lambda x.\,e \mid [\bar{v}] \mid e.(e) \mid e[e \leftarrow e] \mid \texttt{idx_of_int} \mid \texttt{with_array} \\
v &::= n \mid \lambda x.\,e \mid [\bar{v}] \mid \texttt{idx_of_int} \mid \texttt{idx_of_int}\,v \mid \texttt{with_array} \mid \texttt{with_array}\,v \mid \texttt{with_array}\,v\,v \\
E &::= \square \mid E\,e \mid v\,E \mid E.(e) \mid v.(E) \mid E[e \leftarrow e] \mid v[E \leftarrow e] \mid v[v \leftarrow E]
\end{aligned}$$

Operational semantics:

$$\frac{e_1 \longrightarrow e_2}{E[e_1] \longrightarrow E[e_2]} \qquad (\lambda x.\,e)\,v \longrightarrow e[x/v] \qquad \frac{0 \le i < |\bar{v}|}{\texttt{idx\_of\_int}\,[\bar{v}]\,i \longrightarrow i}$$

$$\frac{i < 0 \vee |\bar{v}| \le i}{\texttt{idx\_of\_int}\,[\bar{v}]\,i \longrightarrow \texttt{idx\_of\_int}\,[\bar{v}]\,i} \qquad \frac{\bar{v} = v_1 \dots v_1 \qquad |\bar{v}| = \max(n,0)}{\texttt{with\_array}\,n\,v_1\,v_2 \longrightarrow v_2\,[\bar{v}]}$$

$$\frac{0 \le i < n}{[u_0 \dots u_{n-1}].(i) \longrightarrow u_i} \qquad \frac{0 \le i < n}{[u_0 \dots u_i \dots u_{n-1}][i \leftarrow v] \longrightarrow [u_0 \dots v \dots u_{n-1}]}$$

Typing rules:

$$\textsc{Get}\ \frac{\Gamma \vdash e_1 : (\tau_1,\tau_2)\ \texttt{array} \qquad \Gamma \vdash e_2 : \tau_2\ \texttt{index}}{\Gamma \vdash e_1.(e_2) : \tau_1} \qquad \textsc{Int}\ \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \texttt{int}} \qquad \textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\,e_2 : \tau_2}$$

$$\textsc{Set}\ \frac{\Gamma \vdash e_1 : (\tau_1,\tau_2)\ \texttt{array} \qquad \Gamma \vdash e_2 : \tau_2\ \texttt{index} \qquad \Gamma \vdash e_3 : \tau_1}{\Gamma \vdash e_1[e_2 \leftarrow e_3] : (\tau_1,\tau_2)\ \texttt{array}} \qquad \textsc{AllIntro}\ \frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{FV}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\,\tau}$$

$$\textsc{IdxOfInt}\quad \Gamma \vdash \texttt{idx\_of\_int} : \forall\alpha\beta.\,(\alpha,\beta)\ \texttt{array} \to \texttt{int} \to \beta\ \texttt{index} \qquad \textsc{AllElim}\ \frac{\Gamma \vdash e : \forall\alpha.\,\tau_1}{\Gamma \vdash e : \tau_1[\alpha/\tau_2]} \qquad \textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\textsc{WithArray}\quad \Gamma \vdash \texttt{with\_array} : \forall\alpha\beta.\,\texttt{int} \to \beta \to (\forall\gamma.\,(\beta,\gamma)\ \texttt{array} \to \alpha) \to \alpha \qquad \textsc{Lam}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\,e : \tau_1 \to \tau_2}$$

Logical relation:

$$\begin{aligned}
[\![\alpha]\!]^\rho &\triangleq \rho(\alpha) \\
[\![\tau]\!]^\rho.\mathbf{len} &\triangleq 0 \qquad \text{if } \tau \text{ is not a type variable} \\
[\![\texttt{int}]\!]^\rho.\mathbf{vals} &\triangleq \mathbb{Z} \\
[\![\tau_1 \to \tau_2]\!]^\rho.\mathbf{vals} &\triangleq \{v \mid \forall v_1 \in [\![\tau_1]\!]^\rho.\mathbf{vals}.\ (v\,v_1) \in \mathcal{E}^\rho(\tau_2)\} \\
[\![\forall\alpha.\,\tau]\!]^\rho.\mathbf{vals} &\triangleq \{v \mid \forall A.\ v \in [\![\tau]\!]^{\rho[\alpha \leftarrow A]}.\mathbf{vals}\} \\
[\![(\tau_1,\tau_2)\ \texttt{array}]\!]^\rho.\mathbf{vals} &\triangleq \{[v_0 \dots v_{n-1}] \mid n = [\![\tau_2]\!]^\rho.\mathbf{len} \wedge \forall i \in [0,n-1].\ v_i \in [\![\tau_1]\!]^\rho.\mathbf{vals}\} \\
[\![\tau\ \texttt{index}]\!]^\rho.\mathbf{vals} &\triangleq [0, [\![\tau]\!]^\rho.\mathbf{len} - 1] \\
\mathcal{E}^\rho(\tau) &\triangleq \{e \mid \mathit{Safe}(e) \wedge \forall v.\ e \longrightarrow^* v \Longrightarrow v \in [\![\tau]\!]^\rho.\mathbf{vals}\} \\
\mathcal{G}^\rho(\Gamma) &\triangleq \{\gamma \mid \forall (x : \tau) \in \Gamma.\ \gamma(x) \in [\![\tau]\!]^\rho.\mathbf{vals}\} \\
\Gamma \vDash e : \tau &\triangleq \forall\rho.\ \forall\gamma \in \mathcal{G}^\rho(\Gamma).\ \gamma(e) \in \mathcal{E}^\rho(\tau)
\end{aligned}$$

Figure 1: Definitions of the calculus and of the logical relation

– $[\![\tau]\!]^\rho$.**len** is an integer: the length of an array branded by the type τ.

This definition of $[\![\tau]\!]^\rho$ has some particularities that are worth mentioning:

– Except for type variables, $[\![\tau]\!]^\rho$.**len** is always 0.
– The definition of $[\![\tau_1 \to \tau_2]\!]^\rho$.**vals** for arrow types depends on an auxiliary definition, $\mathcal{E}^\rho(\tau)$, of semantically well-behaved *expressions*: a semantically well-behaved expression is safe (i.e., it does not reduce to a stuck state in any number of steps), and when it reduces to a value, this value is semantically well-behaved.
– The definition of $[\![\forall\alpha.\,\tau]\!]^\rho$.**vals** involves a universal quantification over an arbitrary type interpretation A, meaning the set A.**vals** and the integer A.**len** are completely arbitrary.

Finally, we define the set of semantically well-behaved substitutions $\mathcal{G}^\rho(\Gamma)$ for a given typing environment Γ, and a semantic typing judgement $\Gamma \vDash e : \tau$ (note that $\gamma(e)$ stands for substituting in e any unbound variable x with $\gamma(x)$).

Question 8 *The logical relation is recursively defined: justify why its definition is well-formed.*
Answer
□

Question 9 *Prove that the logical relation is adequate: if e is a closed expression such that $\emptyset \vDash e : \tau$, then e never reduces to a stuck expression.* Answer
□

Question 10 *We assume that v is a value. Prove that $v \in \mathcal{E}^\rho(\tau)$ if, and only if $v \in [\![\tau]\!]^\rho$.**vals**.*
Answer
□

Question 11 *We assume that e is neither stuck nor a value, and that for all e' such that $e \longrightarrow e'$, we have $e' \in \mathcal{E}^\rho(\tau)$. Prove $e \in \mathcal{E}^\rho(\tau)$.* Answer
□

Question 12 *We assume $e \in \mathcal{E}^\rho(\tau_1)$, and that for any $v \in [\![\tau_1]\!]^\rho$.**vals** we have $E[v] \in \mathcal{E}^\rho(\tau_2)$. Prove $E[e] \in \mathcal{E}^\rho(\tau_2)$.* Answer
□

We are now interested in the *fundamental theorem of the logical relation*, which states that every rule of the type system stays valid when replacing the syntactic judgement $\Gamma \vdash e : \tau$ with the semantic counterpart $\Gamma \vDash e : \tau$.

Question 13 *Prove the fundamental theorem **restricted to the rule** Lam.*
Hint: you can use the lemmas proved in Question 10 and Question 11. Answer
□

Question 14 *Prove the fundamental theorem **restricted to the rule** Get.*
Hint: you can use the lemma proved in Question 12. □

Question 15 *Prove the fundamental theorem **restricted to the rule** WithArray.*
Hint: you can use the lemmas proved in Question 10 (three times) and then Question 11. □

We admit the fundamental theorem of the logical relation for the other rules.

Question 16 *Formally state the soundness property for this type system. Prove it.*
Answer
□

```rust
pub struct GhostToken<'id> { ... }
pub struct GhostCell<'id, T> { ... }

impl<'id> GhostToken<'id> {
    pub fn new<F, R>(f: F) -> R
        where F: for<'new_id> FnOnce(GhostToken<'new_id>) -> R { ... }
}

impl<'id, T> GhostCell<'id, T> {
    pub fn new(value: T) -> Self { ... }
    pub fn into_inner(self) -> T { ... }
    pub fn get_mut(&mut self) -> &mut T { ... }

    pub fn borrow<'a>(&'a self, token: &'a GhostToken<'id>) -> &'a T { ... }
    pub fn borrow_mut<'a>(&'a self, token: &'a mut GhostToken<'id>) -> &'a mut T { ... }
}
```

Figure 2: The GhostCell library in Rust

3 Branded types in Rust: `GhostCell`

In this section, we study another use of branded types in the `GhostCell` library, written in the Rust programming language.

The interface of the library is given in fig. 2. This is an alternative to the `RefCell` and `Cell` libraries: it allows creating and modifying data structures with potentially aliased pointers, while still preserving safety. As with `Cell`, values of type `T` are stored in a dedicated type `GhostCell<'id, T>`, which contains no other information (it has the same size). As with `RefCell`, the API allows obtaining values of types `&T` and `&mut T` from values of type `&GhostCell<'id, T>`, but instead of ensuring safety with an internal counter and dynamic checks, `GhostCell` uses *ghost tokens*. These are values of type `GhostToken<'id>`, which do not convey any runtime information (their size is 0), but do convey ownership of some resources.

Both ghost cells and ghost tokens are *branded* by a ghost lifetime `'id`. This lifetime is a *ghost lifetime*: it does not correspond to the lifetime of a borrow, but is in fact a *name* which uniquely identifies a token.

The function bodies do contain `unsafe` blocks, but no runtime check: operationally, every function associated with the `GhostCell` type directly returns its first parameter (after having changed its type).

Question 17 *How does this library allow using a shared borrow to mutate memory? What generic name qualifies libraries providing such a feature?* Answer

□

The function `GhostToken::new` can be used to create a new ghost token. Its prototype uses the syntax `where F: for<'new_id> FnOnce(GhostToken<'new_id>) -> R`, which means that `F` should implement `FnOnce(GhostToken<'new_id>) -> R` for every lifetime `'id`.

Question 18 *Explain the prototype of `GhostToken::new`: what is the purpose of the type variables `F` and `R`? Explain the choice of the trait `FnOnce`. Why is the quantifier `for<'new_id>` necessary? Why doesn't the function `GhostToken::new` return a ghost token directly?*

Answer

□

Question 19 *What general invariant of the Rust type system governs the set of active borrows to a given location at a given instant in time? Give an informal argument as to why the borrows returned by `GhostCell::borrow` and `GhostCell::borrow_mut` preserve this invariant.*

Answer

□

Question 20 *Assuming that the Rust compiler perform reasonable optimizations, what is the performance overhead of ghost cells? How do we qualify such an abstraction?* Answer

□

Question 21 *We assume that there is a declared instance of* `Send` *and* `Sync` *for* `GhostToken<'id>`*. Are there cases where it is safe to declare* `Send` *and* `Sync` *instances for* `GhostCell<'id, T>`*? If yes, at which condition on* `T`*? No need to justify.* Answer

□

Solutions

Question 1

```
module BrandedArray: BrandedArray = struct
  type ('a, 'g) t = 'a array
  type 'g idx = int

  let idx_of_int a i =
    if 0 <= i && i < Array.length a then Some(i) else None

  let get a i = a.(i)
  let set a i x = a.(i) <- x

  type ('a, 'ret) callback = { cb: 'g. ('a, 'g) t -> 'ret }
  let with_array len x cb =
    cb.cb (Array.make len x)
end
```

Question 2

In order to make sure that the branding mechanism works as intended, we need to guarantee that distinct arrays have non-unifiable phantom type variables `'g`. Thus, from the client side, everything should happen as if each array creation created a fresh type to instantiate the type variable `'g`.

If such a function would be added, then the module would no longer have the stated guarantee, since we could have two arrays with different lengths but with the same brand `'g`. Thus, an array index could be created with the longer array and used on the shorter array: the bounds checking would not be effective.

Question 3

```
type 'a wrapped = W: ('a, 'g) t -> 'a wrapped
let make len x = W (Array.make len x)
(* val make: int -> 'a -> 'a wrapped *)
```

Question 4

```
let swap i j a =
  let i = Option.get (BrandedArray.idx_of_int a i) in
  let j = Option.get (BrandedArray.idx_of_int a j) in
  let x = BrandedArray.get a i in
  let y = BrandedArray.get a j in
  BrandedArray.set a j x;
  BrandedArray.set a i y
```

This functions does only 2 bounds checks, while a function written with the OCaml standard library would do 4 bounds checks, once at each array access.

Question 5

Adding this function is still safe, because even with this extension, all the arrays branded with the same type `'g` have the same length. Therefore, indices checked with a given array can be used with another array with the same brand `'g`, since it will have the same size. Brands now identify array lengths, not arrays themselves.

Question 6

- The idealized calculus handles persistent arrays while the signature uses mutable arrays (simplifications: no global heap, the `set` function do not have a `unit` return type).
- The `idx_of_int` function diverges when given an invalid argument instead of returning an `option` type (simplification: no option type).

Question 7

Subject reduction would state that all well-typed expressions stay well-typed after reduction.

There are several reasons why it does not hold: for example, no typing rule gives a type to a value of the form $[\bar{v}]$. Since these values are produced by the evaluation of some well-typed expressions, subject reduction cannot hold.

Similarly, there is no typing rule to give the type τ `index` to an integer.

Question 8

The mutually recursive definitions are those of $[\![\tau]\!]^\rho$ and $\mathcal{E}^\rho(\tau)$. They are defined by structural recursion over the type τ: $\mathcal{E}^\rho(\tau)$ depends directly on $[\![\tau]\!]^\rho$, but $[\![\tau]\!]^\rho$ depends on $\mathcal{E}^\rho(\tau)$ only for structurally smaller types.

Question 9

Let γ be an arbitrary substitution from program variables to values, and ρ an arbitrary substitution from type variables to types. Clearly, $\gamma \in \mathcal{G}^\rho(\emptyset)$, and, because e is closed, $\gamma(e) = e$. So, by definition of $\emptyset \vDash e : \tau$, we have $e \in \mathcal{E}^\rho(\tau)$. This implies $\mathit{Safe}(e)$.

Question 10

A value does not reduce and is always safe. So $[\![\tau]\!]^\rho.\mathbf{vals} \subseteq \mathcal{E}^\rho(\tau)$. Moreover, $v \longrightarrow^* v$, so $v \in \mathcal{E}^\rho(\tau) \Rightarrow v \in [\![\tau]\!]^\rho.\mathbf{vals}$.

Question 11

The expression e is safe since it is not stuck, and all of its direct reducts are safe[2]. Moreover, if $e \longrightarrow^* v$, then there exists e' such that $e \longrightarrow e'$ and $e' \longrightarrow^* v$. We have $e' \in \mathcal{E}^\rho(\tau)$, so $v \in [\![\tau]\!]^\rho.\mathbf{vals}$. So we indeed have $e \in \mathcal{E}^\rho(\tau)$.

Question 12

Let's start by proving the safety of $E[e]$, and imagine that $E[e]$ reduces to a stuck state after some steps. Because of the technical property stated in the exam statement, the reduction of $E[e]$ starts by only reducing e: other reduction steps may only happen when evaluating $E[v]$, after e is reduced to a value v. So either the stuck state appears during the evaluation of e, or it appears during the evaluation of $E[v]$. The first case is impossible, because e is safe since $e \in \mathcal{E}^\rho(\tau_1)$. In the second case, we have $v \in [\![\tau_1]\!]^\rho.\mathbf{vals}$, because $e \in \mathcal{E}^\rho(\tau_1)$. So $E[v] \in \mathcal{E}^\rho(\tau_2)$, which implies that it is safe.

Now, we assume that $E[e] \longrightarrow^* v'$, and we prove that $v' \in [\![\tau_2]\!]^\rho.\mathbf{vals}$: the reduction $E[e] \longrightarrow^* v'$ can again be decomposed into two stages: there exists v such that $E[e] \longrightarrow^* E[v]$ and $E[v] \longrightarrow^* v'$. Because $e \in \mathcal{E}^\rho(\tau_1)$, we have $v \in [\![\tau_1]\!]^\rho.\mathbf{vals}$, and hence $E[v] \in \mathcal{E}^\rho(\tau_2)$, which implies $v' \in [\![\tau_2]\!]^\rho.\mathbf{vals}$.

[2]There is only one reduct because of determinism. But the argument here still holds in the presence of non-determinism.

Question 13

We give here the complete proof for all the rules. Of course, only three of them are required by the question.

- Rule INT: we need to prove that $\mathbb{Z} \subseteq \mathcal{E}^\rho(\texttt{int})$. Since integers are values, this amounts (Question 10) to proving that $\mathbb{Z} \subseteq [\![\texttt{int}]\!]^\rho.\mathbf{vals}$, which is true by definition.
- Rule VAR: after unfolding the definitions, this amounts to prove that if $\gamma(x) \in [\![\tau]\!]^\rho.\mathbf{vals}$, then $\gamma(x) \in \mathcal{E}^\rho(\tau)$, which is true by Question 10.
- Rule APP: let $e'_1 \triangleq \gamma(e_1)$ and $e'_2 \triangleq \gamma(e_2)$. We have $e'_1 \in \mathcal{E}^\rho(\tau_1 \to \tau_2)$ and $e'_2 \in \mathcal{E}^\rho(\tau_1)$, and we need to prove $e'_1\, e'_2 \in \mathcal{E}^\rho(\tau_2)$.
  We use Question 12 with $E = \Box\, e'_2$. We are left to prove that for any $v_1 \in [\![\tau_1 \to \tau_2]\!]^\rho.\mathbf{vals}$, we have $v_1\, e'_2 \in \mathcal{E}^\rho(\tau_2)$.
  We use Question 12 a second time with $E = v_1\, \Box$. We are left to prove that for any $v_2 \in [\![\tau_1]\!]^\rho.\mathbf{vals}$, we have $v_1\, v_2 \in \mathcal{E}^\rho(\tau_2)$.
  This is trivial by definition of $[\![\tau_1 \to \tau_2]\!]^\rho.\mathbf{vals}$.
- Rule LAM: we wish to prove that $\gamma(\lambda x.\, e) \in \mathcal{E}^\rho(\tau_1 \to \tau_2)$. We use Question 10: it is enough to prove that $\gamma(\lambda x.\, e) \in [\![\tau_1 \to \tau_2]\!]^\rho.\mathbf{vals}$. I.e., we need to prove that for any $v \in [\![\tau_1]\!]^\rho.\mathbf{vals}$, we have $\gamma(\lambda x.\, e)\, v \in \mathcal{E}^\rho(\tau_2)$.
  Let e' such that $\gamma(\lambda x.\, e) = \lambda x.\, e'$. We use Question 11: we need to prove that $e'[x/v] \in \mathcal{E}^\rho(\tau_2)$.
  But we remark that $e'[x/v] = \gamma[x \leftarrow v](e)$. By using the premise $\Gamma, x : \tau_1 \vDash e : \tau_2$, this amounts to proving that $\gamma[x \leftarrow v] \in \mathcal{G}^\rho(\Gamma, x : \tau_1)$.
  This last property follows immediately from $\gamma \in \mathcal{G}^\rho(\Gamma)$ and $v \in [\![\tau_1]\!]^\rho.\mathbf{vals}$.
- Rule ALLELIM: by monotonicity of the definitions of the logical relation, this amounts to proving that $[\![\forall\alpha.\, \tau_1]\!]^\rho.\mathbf{vals} \subseteq [\![\tau_1[\alpha/\tau_2]]\!]^\rho.\mathbf{vals}$. By a trivial induction over the type structure, we can prove that $[\![\tau_1[\alpha/\tau_2]]\!]^\rho = [\![\tau_1]\!]^{\rho[\alpha \leftarrow [\![\tau_2]\!]^\rho]}$. By definition, we have $[\![\forall\alpha.\, \tau_1]\!]^\rho.\mathbf{vals} \subseteq [\![\tau_1]\!]^{\rho[\alpha \leftarrow [\![\tau_2]\!]^\rho]}$, which let us conclude.
- Rule ALLINTRO: let ρ and $\gamma \in \mathcal{G}^\rho(\Gamma)$. The safety of $\gamma(e)$ follows directly from the premise $\Gamma \vDash e : \tau$, applied with the same ρ and γ.
  Let v such that $\gamma(e) \longrightarrow^* v$. We want to prove that $v \in [\![\forall\alpha.\, \tau]\!]^\rho.\mathbf{vals}$. That is, for any type interpretation A, we need to prove $v \in [\![\tau]\!]^{\rho[\alpha \leftarrow A]}.\mathbf{vals}$.
  Because $\alpha \notin \mathcal{FV}(\Gamma)$, we have $\mathcal{G}^{\rho[\alpha \leftarrow A]}(\Gamma) = \mathcal{G}^\rho(\Gamma)$. So we can use the premise $\Gamma \vDash e : \tau$ on $\rho[\alpha \leftarrow A]$ and γ, and indeed deduce $v \in [\![\tau]\!]^{\rho[\alpha \leftarrow A]}.\mathbf{vals}$.
- Rule GET: let $e'_1 \triangleq \gamma(e_1)$ and $e'_2 \triangleq \gamma(e_2)$. We have $e'_1 \in \mathcal{E}^\rho((\tau_1, \tau_2)\ \texttt{array})$ and $e'_2 \in \mathcal{E}^\rho(\tau_2\ \texttt{index})$, and we need to prove $e'_1.(e'_2) \in \mathcal{E}^\rho(\tau_1)$.
  As in the proof for APP, we use Question 12 twice: we have two values $v_1 \in [\![(\tau_1, \tau_2)\ \texttt{array}]\!]^\rho.\mathbf{vals}$ and $v_2 \in [\![\tau_2\ \texttt{index}]\!]^\rho.\mathbf{vals}$, and we wish to prove $v_1.(v_2) \in \mathcal{E}^\rho(\tau_1)$. The value v_2 is an integer j, with $0 \leq j < [\![\tau_2]\!]^\rho.\mathbf{len}$, and v_1 is of the form $[w_0 \dots w_{[\![\tau_2]\!]^\rho.\mathbf{len}-1}]$, with $w_j \in [\![\tau_1]\!]^\rho.\mathbf{vals}$.
  The reduction $v_1.(v_2) \longrightarrow w_j$ holds and is the only available. This let us conclude $v_1.(v_2) \in \mathcal{E}^\rho(\tau_1)$, because $w_j \in [\![\tau_1]\!]^\rho.\mathbf{vals}$.
- Rule SET: we proceed as for rule GET. By applying Question 12 three times, we are left to prove that if $v_1 \in [\![(\tau_1, \tau_2)\ \texttt{array}]\!]^\rho.\mathbf{vals}$, $v_2 \in [\![\tau_2\ \texttt{index}]\!]^\rho.\mathbf{vals}$ and $v_3 \in [\![\tau_1]\!]^\rho.\mathbf{vals}$, then $v_1[v_2 \leftarrow v_3] \in \mathcal{E}^\rho((\tau_1, \tau_2)\ \texttt{array})$. The value v_2 is an integer j, with $0 \leq j < [\![\tau_2]\!]^\rho.\mathbf{len}$, and v_1 is of the form $[w_0 \dots w_{[\![\tau_2]\!]^\rho.\mathbf{len}-1}]$.
  The reduction $v_1[v_2 \leftarrow v_3] \longrightarrow [w_0 \dots v_3 \dots w_{[\![\tau_2]\!]^\rho.\mathbf{len}-1}]$ holds and is the only available. But $[w_0 \dots v_3 \dots w_{[\![\tau_2]\!]^\rho.\mathbf{len}-1}] \in [\![(\tau_1, \tau_2)\ \texttt{array}]\!]^\rho.\mathbf{vals}$, because $v_3 \in [\![\tau_1]\!]^\rho.\mathbf{vals}$. This let us conclude $v_1[v_2 \leftarrow v_3] \in \mathcal{E}^\rho((\tau_1, \tau_2)\ \texttt{array})$.
- Rule IDXOFINT: after unfolding the definitions, and applying Question 10 twice, this amounts to proving that if:
  - A and B are two type interpretations,
  - $\rho' = \rho[\alpha \leftarrow A, \beta \leftarrow B]$,
  - $[\bar{v}] \in [\![(\alpha, \beta)\ \texttt{array}]\!]^{\rho'}.\mathbf{vals}$ such that $|\bar{v}| = B.\mathbf{len}$, and
  - $n \in \mathbb{Z}$,

  then $\texttt{idx\_of\_int}\, v_1\, v_2 \in \mathcal{E}^{\rho'}(\beta\ \texttt{index})$.

We distinguish two cases for proving this fact:
  - either $n < 0 \vee |\bar{v}| \leq n$, in which case $\texttt{idx\_of\_int}\, v_1\, v_2$ diverges, and is therefore indeed an element of $\mathcal{E}^{\rho'}(\beta\ \texttt{index})$;
  - or $0 \leq n < |\bar{v}|$, in which case $\texttt{idx\_of\_int}\, v_1\, v_2 \longrightarrow n$ is valid and is the only possible reduction, with $n \in [\![\beta\ \texttt{index}]\!]^{\rho'}.\mathbf{vals}$ (because $0 \leq n < [\![\beta]\!]^{\rho'}.\mathbf{len} = \rho'(\beta) = B.\mathbf{len}$).

  So, in both cases we have $\texttt{idx\_of\_int}\, v_1\, v_2 \in \mathcal{E}^{\rho'}(\beta\ \texttt{index})$.
- Rule WithArray: after unfolding the definitions, and applying Question 10 three times, we are left to prove that if:
  - A and B are two type interpretations,
  - $\rho' = \rho[\alpha \leftarrow A, \beta \leftarrow B]$,
  - $n \in \mathbb{Z}$,
  - $v_1 \in B.\mathbf{vals}$,
  - $v_2 \in [\![\forall\gamma.\, (\beta, \gamma)\ \texttt{array} \to \alpha]\!]^{\rho'}.\mathbf{vals}$, and
  - $\bar{v}$ is the sequence of values containing v_1, repeated $\max(n, 0)$ times,

  then $\texttt{with\_array}\, n\, v_1\, v_2 \in \mathcal{E}^{\rho'}(\alpha)$.

  We use Question 11: the reduction $\texttt{with\_array}\, n\, v_1\, v_2 \longrightarrow v_2\, [\bar{v}]$ is valid and is the only available. So we are left to prove that $v_2\, [\bar{v}] \in \mathcal{E}^{\rho'}(\alpha)$.

  Let C the type interpretation such that $C.\mathbf{vals} = \emptyset$ and $C.\mathbf{len} = \max(n, 0)$, and let $\rho'' = \rho[\gamma \leftarrow C]$.

  We have:
  - $v_2 \in [\![(\beta, \gamma)\ \texttt{array} \to \alpha]\!]^{\rho''}.\mathbf{vals}$, and,
  - $[\bar{v}] \in [\![(\beta, \gamma)\ \texttt{array}]\!]^{\rho''}.\mathbf{vals}$ because $v_1 \in B.\mathbf{vals} = [\![\beta]\!]^{\rho''}.\mathbf{vals}$ and $|\bar{v}| = \max(n, 0) = C.\mathbf{len} = [\![\gamma]\!]^{\rho''}.\mathbf{len}$.

  So, by definition, $v_2\, [\bar{v}] \in \mathcal{E}^{\rho''}(\alpha) = A = \mathcal{E}^{\rho'}(\alpha)$.

Question 16

The soundness of the type system states that no closed well-typed expression reduces to a stuck expression.

Let's prove it: we assume $\Gamma \vdash e : \tau$. By a trivial induction using the fundamental theorem, we can prove $\Gamma \vDash e : \tau$. Thanks to adequacy, we conclude that indeed e cannot reduce to a stuck state.

Question 17

The function `borrow_mut` uses a shared borrow to a ghost cell to return a unique borrow to the content of the ghost cell, which can be used to mutate memory. So this library allows using a potentially aliased pointer to mutate memory.

Thus, this library provides *interior mutability*.

Question 18

Instead of returning a token as a return value, the function `GhostToken::new` takes a closure which it will call with the token as parameter. Thus, it is written in the same style as `with_array` above.

When the closure terminates, it returns a value of type `R`: the result of the computation that used a token. This result will itself be returned by `GhostToken::new`.

As always in Rust, there is no "type of closures": closures have types, which implement a closure trait: `F` is such a type.

This closure implements the trait `FnOnce(GhostToken<'new_id>) -> R`, for any lifetime `'new_id`: the choice of `FnOnce` means that the closure will be called *at most once*, and therefore can consume its captured variables.

The quantification over `'new_id` is required so that we can guarantee that the closure is called with a fresh lifetime each time, and therefore that tokens can be distinguished statically by their brands. This is analogous to the quantification over `'g` in the type of `with_array` above.

This complex prototype is necessary to guarantee that every token bears a different "branding" lifetime: if the token were returned directly, then the branding lifetime would be quantified by the caller, and this does not guarantee uniqueness.

Question 19

The general invariant, shortly stated as "mutation XOR aliasing" states that there are never two active borrows to the same memory location, at least one of them being mutable.

The branding lifetime `'id` uniquely identifies ghost tokens, and a given ghost cell can only by associated to one branding lifetime. Hence, if a unique borrow `&mut T` is created with `GhostCell::borrow_mut`, then we must have consumed a unique borrow of the corresponding token, with the same lifetime. So, as long as this unique borrow `&mut T` is active, the token is uniquely borrowed, and there cannot be another active unique or shared borrow to the token. Thus we cannot call `GhostCell::borrow_mut` or `GhostCell::borrow` to create a conflicting borrow.

Similarly, as long as a shared borrow `&T` created with `GhostCell::borrow` is active, there cannot be a unique borrow of the token, and we cannot call `GhostCell::borrow_mut`.

In a sense, when a `GhostCell` is shared, the ownership of the underlying value is carried by the token: any access to the value requires an access to the token of the same kind.

Question 20

We can reasonably assume that all the function calls will be inlined, since the function bodies only perform trivial actions. This includes the `GhostToken::new` function, which only calls a closure whose body is known. Moreover, ghost tokens occupy no space in memory. So there is no performance overhead associated with using this library: this is a *Zero-cost Abstraction*.

Question 21

(We give here a justification which is not required by the statement.)

Sending a `GhostCell` to another thread is the same as transferring the full ownership of its contents. So it is safe to assume `GhostCell<'id, T>: Send` if, and only if `T: Send` (and potentially unsafe otherwise).

On the other hand, tokens can always be transferred and shared between threads, thanks to the assumed instances. So, if we can have shared borrows to a `GhostCell` in two different threads, then:

- We can have shared borrows to the token in the two threads (because the token is `Sync`), and use `borrow` in the two threads to get shared borrows `&T` to the same inner value. Hence, we can share `T`.
- We can consecutively call `borrow_mut` in each thread, by transferring the token in between (this is allowed because the token is `Send`). The unique borrows obtained from these calls can be used to access the cell for writing and reading in two different threads. This means that we can use the `GhostCell` to transfer the ownership of a value of type `T`.

Hence, to preserve safety, as soon as `GhostCell<'id, T>: Sync`, we need `T` to be `Sync` and `Send`. So it is safe to assume `GhostCell<'id, T>: Sync` if, and only if `T: Send + Sync` (and potentially unsafe otherwise).