Answers to the three parts are mostly independent of one another, although answers to parts 2 and 3 still require close reading of part 1. Questions are not ordered by their difficulty or the length of expected answers.

Answers will also be judged by their clarity and concision.

In the course, we studied a call-by-value version of System F with references, exceptions, and recursion. In exercises below, we enhance typechecking of System F by tracing different forms of effects, independently of one another, but using closely related techniques. We define two new languages \( \mathcal{F}_{\text{ref}} \) and \( \mathcal{F}_{\text{exn}} \) that have the same dynamic semantics as System F, and only differ from it by their types and typing rules.

In contrast with the course, languages are presented and studied in their implicitly-typed versions, so as to avoid explicit effect annotations inside source terms.

1 A larger class of non-expansive expressions / Étendre la classe des expressions non-expansives

Question 1

In System F with references, we restricted type generalization to expressions in value form. Why did we make this restriction?

Question 2

In their simplest form, value forms, written \( u \), are restricted to only variables and values.

Why can we safely treat variables as values?
We call $F_{\text{ref}}$ the following variant of System $F$ that traces side effects in the store.

For that purpose we introduce a new class of type expressions $\varphi$ call effects. An effect may be an effect variable $\varepsilon$, the constant $\circ$ (read: pure) which guarantees the absence of side-effect during evaluation or the constant $\bullet$ (read: impure) which allows a possible side-effect during evaluation. We write $\gamma$ to range other either regular type variables $\alpha$ or effect variables $\varepsilon$. We write $\sigma$ to range other type expressions, which may be a regular type $\tau$ or an effect $\varphi$.

Notice that the arrow type constructor becomes ternary: it carries an additional effect that describes the latent effect that may occur when a function of that type is applied.

As in the course, type variables must be explicitly bound in $\Gamma$ and $\Gamma$ cannot bind twice the same variable. We write $\Gamma \vdash \sigma$ to mean that $\sigma$ is well-formed in $\Gamma$, which also implies that $\Gamma$ is well-formed. We omit well-formedness typing rules which are as in the course. Typing rules of $F_{\text{ref}}$ are as follows:

$$
\begin{align*}
\text{VAR} & \\
x : \tau \in \Gamma & \quad \Gamma \vdash \varphi \\
\hline
\Gamma \vdash x : \tau & \quad \Gamma \vdash \varphi \\
\text{CONST} & \\
c : \tau \in \Delta & \quad \Gamma \vdash \varphi \\
\hline
\Gamma \vdash c : \tau & \quad \Gamma \vdash \varphi \\
\text{LAM} & \\
\Gamma, x : \tau_1 \vdash a : \tau_2 & \quad \Gamma \vdash \varphi_0 \\
\hline
\Gamma \vdash \lambda x. a : \tau \rightarrow \langle \varphi \rangle \tau_2 & \quad \Gamma \vdash \varphi_0 \\
\end{align*}
$$

Finally, the typing environment for constants $\Delta$ assigns the following types to primitives:

$$
\begin{align*}
\text{ref} & : \forall \alpha. \alpha \rightarrow \langle \bullet \rangle \text{ref} \alpha \\
(\!) & : \forall \alpha. \text{ref} \alpha \rightarrow \langle \bullet \rangle \alpha \\
(\varepsilon) & : \forall \alpha. \varepsilon \rightarrow \langle \varepsilon \rangle \text{ref} \alpha \rightarrow \langle \bullet \rangle \alpha
\end{align*}
$$

(Notice that assignment will return the value assigned to the reference so as to avoid introducing a unit.) In program examples, we may also use let bindings let $x : \tau = a$ in $b$ as syntactic sugar for $(\lambda x. b) a$.

**Question 3**

Describe the grammar of values in $F_{\text{ref}}$.  

Nous appelons $F_{\text{ref}}$ la variante suivante du système $F$ qui trace les effets de bord dans la mémoire.

Pour cela, nous introduisons une nouvelle classe d’expressions $\varphi$ appelées effets. Un effet peut être une variable d’effet $\varepsilon$, la constante $\circ$ (lire : pure) qui garantit l’absence d’effets de bords pendant l’évaluation ou la constante $\bullet$ (lire : impure) qui permet un éventuel effet de bord pendant l’évaluation. Nous écrivons $\gamma$ pour désigner indifféremment une variable de type usuelle $\alpha$ ou une variable d’effet $\varepsilon$ et $\sigma$ pour désigner une expression de type, qui peut être un type usuel $\tau$ ou un effet $\varphi$.

Remarquer que le constructeur de type flèche devient ternaire, prenant un argument supplémentaire qui décrit l’effet latent qui peut apparaître quand une fonction de ce type est appliquée.

Comme dans les notes de cours, les variables de types doivent être explicitement liées dans $\Gamma$ et cela au plus une fois. On écrit $\Gamma \vdash \sigma$ pour dire que $\sigma$ est bien formé dans $\Gamma$, ce qui implique aussi que $\Gamma$ est bien formé. On omet les règles de bonne formation, qui sont comme dans le cours. Les règles de typage de $F_{\text{ref}}$ sont les suivantes:

$$
\begin{align*}
\text{GEN} & \\
\Gamma, \gamma \vdash a : \tau & \quad \Gamma \vdash \sigma \\
\hline
\Gamma, \gamma \vdash a : \forall \gamma. \tau & \quad \Gamma \vdash \sigma \\
\end{align*}
$$

$$
\begin{align*}
\text{APP} & \\
\Gamma \vdash a_1 : \tau_2 & \quad \langle \varphi \rangle \tau_1 \quad \Gamma \vdash a_2 : \tau_2 \\
\hline
\Gamma \vdash a_1 : \tau_2 \rightarrow \langle \varphi \rangle \tau_1 \quad \Gamma \vdash a_2 : \tau_2 \\
\end{align*}
$$

Enfin, l’environnement de typage des constantes $\Delta$ assigne les types suivants aux primitives:

$$
\begin{align*}
\text{ref} & : \forall \alpha. \alpha \rightarrow \langle \bullet \rangle \text{ref} \alpha \\
(\!) & : \forall \alpha. \text{ref} \alpha \rightarrow \langle \bullet \rangle \alpha \\
(\varepsilon) & : \forall \alpha. \varepsilon \rightarrow \langle \varepsilon \rangle \text{ref} \alpha \rightarrow \langle \bullet \rangle \alpha
\end{align*}
$$

(Noter que l’affectation retourne la valeur assignée à la référence pour éviter d’avoir à introduire une constante unit.) Dans les programmes, nous pourrons également utiliser la construction de liaison let $x : \tau = a$ in $b$ comme du sucire syntaxique pour $(\lambda x. b) a$.  

**Question 3**

Décrire la grammaire des valeurs dans $F_{\text{ref}}$.  

---

2
Question 4

Can you explain why the conclusion of rule Lam is \( \Gamma \vdash \lambda x. a : \tau \rightarrow (\varphi) \varphi' \mid \varphi_0 \) rather than \( \Gamma \vdash \lambda x. a : \tau \rightarrow (\varphi) \varphi' \mid \circ \)?

Pouvez-vous expliquer pourquoi la conclusion de la règle Lam est \( \Gamma \vdash \lambda x. a : \tau \rightarrow (\varphi) \varphi' \mid \varphi_0 \) plutôt que \( \Gamma \vdash \lambda x. a : \tau \rightarrow (\varphi) \varphi' \mid \circ \)?

Question 5

Let \( a \) be the term \( \lambda f. \lambda x. f \, x \). It admits simple types of the form:

\[
(\tau_1 \rightarrow (\varphi_0) \varphi_2) \rightarrow (\varphi_1) \tau_1 \rightarrow (\varphi_2) \tau_2
\]

Describe all the triplets \((\varphi_0, \varphi_1, \varphi_2)\) for which this is a type of \( a \).

Soit \( a \) le terme \( \lambda f. \lambda x. f \, x \). Il admet des types simples de la forme :

\[
(\tau_1 \rightarrow (\varphi_0) \varphi_2) \rightarrow (\varphi_1) \tau_1 \rightarrow (\varphi_2) \tau_2
\]

Décrire tous les triplets \((\varphi_0, \varphi_1, \varphi_2)\) pour lesquels ceci est bien un type de \( a \).

Question 6

The following program is known to be unsound.

\[
\begin{align*}
\text{let } x : \forall \alpha. \text{ref}(\alpha \rightarrow \alpha) = \text{ref}(\lambda z. z) & \\
(x) := \text{not} ; ! (x) 1
\end{align*}
\]

a) Explain how it is rejected in System F?

b) Is it still rejected in \( F_{\text{ref}} \) (justify)?

Le programme suivant est connu pour être incorrect.

\[
\begin{align*}
\text{let } x : \forall \alpha. \text{ref}(\alpha \rightarrow \alpha) = \text{ref}(\lambda z. z) & \\
(x) := \text{not} ; ! (x) 1
\end{align*}
\]

a) Expliquer comment il est rejeté dans le système F?

b) Est-il toujours rejeté dans \( F_{\text{ref}} \) (justifier) ?

Question 7

Give a term in \( F_{\text{ref}} \) that is not typable in \( F \).

Explain why it is well-typed in \( F_{\text{ref}} \) (just give the key steps of its typing derivation).

Can you intuitively explain why your example is safe in \( F_{\text{ref}} \)?

Donner un terme de \( F_{\text{ref}} \) qui n’est pas typable dans \( F \).

Expliquer pourquoi il est bien typé dans \( F_{\text{ref}} \) (donner seulement les étapes clés de sa dérivation de typage).

Pouvez-vous expliquer intuitivement pourquoi votre exemple est sûr dans \( F_{\text{ref}} \) ?

Question 8

Conversely, all closed programs typable in System F are also typable in \( F_{\text{ref}} \). Can you say why this is true, intuitively?

À l’inverse, tous les programmes clos du système F sont typables dans \( F_{\text{ref}} \). Pouvez-vous dire pourquoi c’est le cas, intuitivement ?

To define the dynamic semantics, we introduce locations, model the store as a map from locations to values, and reduce configurations which are pairs \( a / \mu \) composed of an expression and a store. Typing rules for locations and configurations are as follows:

\[
\begin{align*}
\text{LOC:} & \quad \ell : \tau \in \Gamma \quad \Gamma \vdash \varphi \\
\text{CONFIG:} & \quad \gamma \vdash a : \tau \mid \varphi \quad \gamma \vdash \mu : \Sigma \\
\text{STORE:} & \quad \forall \ell \in \text{dom}(\mu) \quad \gamma, \Sigma \vdash \mu(\ell) : \Sigma(\ell) \mid \circ \\
\end{align*}
\]

Evaluation contexts \( E \) are:

\[
E := [ ] \mid a \mid v [ ]
\]
Reduction rules including $\delta$-rules for store operations are:

$$
\text{Beta-v} \quad (\lambda x. a) \, v / \mu \longrightarrow [x \mapsto v]a / \mu \\
\text{Ref} \quad \ell \notin \text{dom}(\mu) \quad \mu / \text{ref} \, v \longrightarrow \mu[\ell \mapsto v] / \ell \\
\text{Set} \quad \mu / (\ell := v) \longrightarrow \mu[\ell \mapsto v] / v \\
\text{Get} \quad \mu / ! \ell \longrightarrow \mu / \mu(\ell)
$$

You may assume that basic lemmas (permutation, weakening, substitution lemmas) hold. Still, their use in proofs should be explicitly mentioned.

**Question 9**

Show that $\Gamma \vdash E[a] : \tau | \circ$ implies that there exists $\vec{\alpha}$ and $\tau'$ such that $\Gamma, \vec{\alpha} \vdash a : \tau' | \circ$.

You may admit that $\Gamma \vdash a : \tau | \circ$ is not derivable when $a$ is a full application of a constant (i.e. ref $a$, ! $a$, or $a_1 := a_2$).

**Question 10**

Show that evaluation of pure expressions do not modify the store, i.e.

$$
\vec{\gamma} \vdash a_1 / \mu_1 : \tau | \circ \land a_1 / \mu_1 \longrightarrow a_2 / \mu_2 \implies \mu_1 = \mu_2
$$

**Question 11**

Can you state a stronger property about the evaluation of pure expressions (without proving it)?

**Question 12**

Show that the following typing rule is admissible in $F_{\text{ref}}$ (i.e. adding it won’t change the set of derivable judgments):

$$
\text{Sub} \quad \Gamma \vdash a : \tau | \emptyset \\
\Gamma \vdash a : \tau | \varphi
$$

2 An optimized monadic translation / Une traduction monadique optimisée

In this section, we remove from $F_{\text{ref}}$ the effect variables $\varepsilon$ and thus also the ability to generalize over an effect variable, $\forall \varepsilon. \tau$ (but we retain the ability to generalize over a type variable, $\forall \alpha. \tau$). Therefore, effects $\varphi$ are either $\circ$ or $\bullet$.

Dans cette section, nous enlevons du langage $F_{\text{ref}}$ les variables d’effet $\varepsilon$ et donc également la possibilité de généraliser une variable d’effet, $\forall \varepsilon. \tau$. (Mais nous laissons la possibilité de généraliser une variable de type, $\forall \alpha. \tau$.) Par conséquent, un effet $\varphi$ est soit $\circ$ soit $\bullet$.  

4
In the course, we studied a generic monadic translation that takes imperative programs like those of language \text{F}_{\text{ref}} and turns them into pure functional programs, using the \texttt{bind} operator of the monad to encode the sequencing of computations. This translation is coarse: every subterm of the source program is turned into a monadic computation, even those subterms that are already pure (e.g. do not use the operations \texttt{ref}, \texttt{!} and := over references).

In this section, we shall take advantage of the effects tracked by our type-and-effect system to define a more clever monadic translation for \text{F}_{\text{ref}}:

- A term that has effect $\circ$ (pure) is translated into a pure System \text{F} term that does not use the \texttt{ret} and \texttt{bind} operations of the state monad.
- A term that has effect $\bullet$ (impure) is translated into a monadic computation within the state monad, using \texttt{ret} to inject values into computations and \texttt{bind} to sequence computations.

The target language for the monadic translation is System \text{F} extended with types \texttt{ref} $\tau$ for references and \texttt{mon} $\tau$ for monadic computations in the state monad:

$$
\tau ::= \alpha \mid \tau \to \tau \mid \texttt{ref} \tau \mid \texttt{mon} \tau \mid \forall \gamma. \tau
$$

The target language provides, as constants, the operations of the state monad:

- \texttt{ret} : $\forall \alpha. \alpha \to \texttt{mon} \alpha$
- \texttt{bind} : $\forall \alpha. \forall \beta. \texttt{mon} \alpha \to (\alpha \to \texttt{mon} \beta) \to \texttt{mon} \beta$
- \texttt{ref} : $\forall \alpha. \alpha \to \texttt{mon} \texttt{ref} \alpha$
- (\!) : $\forall \alpha. \texttt{ref} \alpha \to \texttt{mon} \alpha$
- (:=) : $\forall \alpha. \alpha \to \texttt{ref} \alpha \to \texttt{mon} \alpha$

The optimized monadic translation is directed by the types and effects assigned to source terms. We, therefore, present the translation as an extension of the typing judgment for \text{F}_{\text{ref}}, which becomes:

$$
\Gamma \vdash a : \tau \mid \varphi \Rightarrow b
$$

The new component $b$ is a term of System \text{F} plus state monad that is the monadic translation of the \text{F}_{\text{ref}} term $a$, viewed with type $\tau$ and effect $\varphi$.

Here are some of the rules defining this typing-and-translation judgment:

- Un terme qui a l’effet $\circ$ (pur) est transformé en un terme pur du Système \text{F} qui n’utilise pas les opérations \texttt{ret} et \texttt{bind} de la monade d’état.
- Un terme qui a l’effet $\bullet$ (impur) est transformé en un calcul monadique à l’intérieur de la monade d’état, utilisant \texttt{ret} pour injecter les valeurs dans les calculs, et \texttt{bind} pour enchaîner les calculs en séquence.

Le langage cible de la traduction monadique est le système \text{F} étendu avec les types \texttt{ref} $\tau$ pour les références et \texttt{mon} $\tau$ pour les calculs monadiques dans la monade d’état :

$$
\tau ::= \alpha \mid \tau \to \tau \mid \texttt{ref} \tau \mid \texttt{mon} \tau \mid \forall \gamma. \tau
$$

Le langage cible fournit les constantes suivantes, qui sont les opérations de la monade d’état :

- \texttt{ret} : $\forall \alpha. \alpha \to \texttt{mon} \alpha$
- \texttt{bind} : $\forall \alpha. \forall \beta. \texttt{mon} \alpha \to (\alpha \to \texttt{mon} \beta) \to \texttt{mon} \beta$
- \texttt{ref} : $\forall \alpha. \alpha \to \texttt{mon} \texttt{ref} \alpha$
- (\!) : $\forall \alpha. \texttt{ref} \alpha \to \texttt{mon} \alpha$
- (:=) : $\forall \alpha. \alpha \to \texttt{ref} \alpha \to \texttt{mon} \alpha$

La traduction monadique optimisée est dirigée par les types et les effets attribués aux termes sources. Par conséquent, nous définissons la traduction comme une extension de la relation de typage pour \text{F}_{\text{ref}}, qui devient :

$$
\Gamma \vdash a : \tau \mid \varphi \Rightarrow b
$$

Dans cette relation, la nouvelle composante $b$ est un terme du système \text{F} plus monade d’état. Ce $b$ est la traduction monadique du terme $a$ de \text{F}_{\text{ref}}, vu avec le type $\tau$ et l’effet $\varphi$.

Voici quelques unes des règles qui définissent cette relation de typage et de traduction:
Question 13

Fill in the "???" holes in the typing-and-translation rule TApp for function applications above.

Question 14

Give typing-and-translation rules TGen and TInst for generalization and instantiation of type variables α.

We expect the optimized monadic translation to preserve types, in the following sense: the translation of a well-typed \( F_{ref} \) term \( a \) is a well-typed \( F \) term \( b \), and moreover the type of \( b \) is determined by the type and the effect of \( a \). More formally:

\[
\text{if } \Gamma \vdash a : \tau \mid \varphi \Rightarrow b, \text{ then } \left[ \left[ \Gamma \right] \right] \vdash b : \left[ \left[ \tau \right] \right]_{\varphi}
\] (1)

The translations \( \left[ \tau \right]_{\varphi} \) of a type-and-effect \( \tau, \varphi \) and \( \left[ \Gamma \right] \) of a typing environment \( \Gamma \) have the following shape:

\[
\begin{align*}
\left[ \tau \right]_{\bullet} &= \text{mon } \left[ \tau \right]_{o} \\
\left[ \alpha \right]_{o} &= \alpha \\
\left[ \tau_{1} \rightarrow \langle \varphi \rangle \tau_{2} \right]_{o} &= \left[ \tau_{1} \right]_{o} \rightarrow ???
\end{align*}
\]

Les traductions \( \left[ \tau \right]_{\varphi} \) d’un type-avec-effet \( \tau, \varphi \) et \( \left[ \Gamma \right] \) d’un environnement de typage \( \Gamma \) ont la forme suivante :

\[
\begin{align*}
\left[ \tau \right]_{\bullet} &= \text{mon } \left[ \tau \right]_{o} \\
\left[ \text{ref } \tau \right]_{o} &= \text{ref } \left[ \tau \right]_{o} \\
\left[ \forall \alpha. \tau \right]_{o} &= \forall \alpha. \left[ \tau \right]_{o} \\
\left[ x_{1} : \tau_{1}, x_{2} : \tau_{2}, \ldots \right] &= x_{1} : ???, x_{2} : ???, \ldots
\end{align*}
\]

Question 15

Explain informally why we have \( \left[ \tau \right]_{\bullet} = \text{mon } \left[ \tau \right]_{o} \).

Question 16

Fill in the "???" blanks above in the translation of function types and of typing environments.
Question 17

Check that the typing preservation property (1) holds in the case of a function abstraction (rule TLam) and of a function application (rule TApp). There is no need to do a full proof of property (1): just outline why it works in the TLam and TApp cases.

Question 18

Likewise, check the typing preservation property (1) for the typing-and-translation rule TInst for type instantiation. If it does not hold for the TInst rule that you proposed in question 14, try to rewrite this rule so that type preservation holds.

Question 19

Do you see a difficulty in extending the optimized monadic translation to full $F_{ref}$ with effect variables $\varepsilon$ that can be universally quantified in types such as $\forall \varepsilon. \tau \rightarrow \langle \varepsilon \rangle \tau$? (Hint: consider a term that has this polymorphic type $\forall \varepsilon. \tau \rightarrow \langle \varepsilon \rangle \tau$ and is later used once after instantiating $\varepsilon = \circ$, once after instantiating $\varepsilon = \bullet$.)

3 Tracing exceptions / Garder trace des exceptions

We now consider another variant $F_{exn}$ of System $F$ that does not have references but instead has exceptions. For simplicity, we provide exceptions with a primitive construct instead of constants.

\[
\begin{align*}
a & ::= \ldots | \text{raise } a | \text{try } a \text{ with } a \\
\tau & ::= \alpha | \tau \rightarrow \langle \phi \rangle \tau | \forall \gamma. \tau | \text{exn} \\
c & ::= \ldots
\end{align*}
\]

In types $\tau$, we just replace $\text{ref } \tau$ by a fixed type $\text{exn}$. We assume given some constants $c$ to construct and destruct values of type $\text{exn}$—without raising or catching exceptions. An example would be to take $\text{unit}$ for $\text{exn}$ and () for the single constant, but we leave it open for generality, which also provides better intuitions.

We recall the typing rules for exceptions in System $F$:

\[
\begin{align*}
\text{Raise} & \\
\Gamma \vdash a : \text{exn} & \quad \Gamma \vdash \text{raise } a : \tau
\end{align*}
\]

Dans les types $\tau$, on a remplacé $\text{ref } \tau$ par un unique type $\text{exn}$. On se donne des constantes $c$ pour construire et détruire des valeurs de type $\text{exn}$ mais sans jamais lever ou rattraper d’exceptions. Par exemple, on peut prendre unit pour $\text{exn}$ et () comme unique constante, mais nous préférons rester plus général, ce qui fournit aussi de meilleures intuitions.

On rappelle les règles de typage des exceptions dans le système $F$:

\[
\begin{align*}
\text{Try} & \\
\Gamma \vdash a_1 : \tau & \quad \Gamma \vdash a_2 : \text{exn} \rightarrow \tau & \quad \Gamma \vdash \text{try } a_1 \text{ with } a_2 : \tau
\end{align*}
\]
In $F_{\text{exn}}$, we use a type-and-effect system similar to that of part 1 to statically know a superset of exceptions that may be raised during evaluation. The $\bullet$ effect means that an exception can be raised during execution, while the $\circ$ effect ensures that no exception can be raised. Formally, we want the following safety property to hold:

$$\text{If } \Gamma \vdash a : \tau \mid \circ \text{ then the reduction of } a \text{ does not raise an uncaught exception.} \quad (1)$$

**Question 20**

Give the type-and-effect rules in $F_{\text{exn}}$ for the two constructs $\text{raise } a$ and $\text{try } a_1 \text{ with } a_2$.

**Question 21**

Remarkably, we can just copy the other typing rules from $F_{\text{ref}}$ unchanged and the safety property is satisfied. Still, one rule is unnecessarily too restrictive. Tell which one and give a less restrictive version of this rule (no justification is required).

**Question 22**

Let $v$ be a value of type $\text{exn}$ and $a$ be the expression $\text{try } (\lambda x. \text{raise } x) \ v \text{ with } \lambda z. \ z$.

a) Give a useful typing for $a$.

b) What can we learn from this on the evaluation of $a$?

c) Give the reduction sequence for $a$.

**Question 23**

In System $F$, the progress lemma states that a well-typed irreducible closed term is either a value or an uncaught exception $\text{raise } v$.

a) Can you give a refined version of this lemma in $F_{\text{exn}}$?

b) Show that the safety property (1) holds. (You may assume that basic lemmas and subject reduction hold in $F_{\text{exn}}$.)

**Question 24**

We would like to combine the languages $F_{\text{ref}}$ and $F_{\text{exn}}$ and trace both effects in the store and exceptional effects, simultaneously. How would you do so? (Explain the idea briefly, without all the details.)
Solution

Question 1
To ensure that their evaluation does not allocate a piece of store, since a piece of store cannot be polymorphic.

Question 2
Since the semantics of the language is call-by-value, when the evaluation of an expression reaches the occurrence of a variable, this variable will have been substituted by a value.

Question 3

\[ v ::= \lambda x. a \mid c \mid (:=) v \]

(Notice the partial application of the assignment constant \((:=)\) of arity 2; all constants \(\text{ref}\), !, and \((:=)\) are part of \(c\).)

Question 4
The latter is more general: although a function is pure, we may wish to also see it as impure, \(e.g.\) to mix it with another impure function or to pass it an impure argument.

Question 5
All tripplets of the form \((\varphi, \varphi', \varphi)\) plus \((\circ, \varphi'_1, \bullet)\) are correct ones (thanks to admissible rule \(\text{Sub}\)—see Question 12). In particular, \((\bullet, \varphi', \circ)\) is not possible.

Question 6
This program is rejected in System \(F\) because the body of the type abstraction \(\text{ref} (\lambda z. z)\), say \(a\) is not a value, hence the type abstraction \(a\) is syntactically ill-formed in \(F\).

Question 6 (continued)
The expression is syntactically well-formed in \(F_{\text{ref}}\) but ill-typed, because \(a\) has the following typing \(\alpha \vdash a : \text{ref} (\alpha \to \alpha) \mid \bullet\) with an impure effect and, in particular, it does not admit the typing \(\alpha \vdash a : \text{ref} (\alpha \to \alpha) \mid \circ\); thus rule \(\text{Gen}\) cannot be applied.

Question 7
Let \(a\) be let \(y = a_0\) \(a_0\) in \(y\) \(y\) where \(a_0\) is \((\lambda x. x)\). We have \(\vdash a_0 : \forall \alpha. \alpha \to (\circ) \alpha \mid \circ\). Hence the application \(a_0\) \(a_0\) is pure and can be generalized: \(\vdash a_0 : \forall \alpha. \alpha \to (\circ) \alpha \mid \circ\). Therefore, we have \(\vdash a : \forall \alpha. \alpha \to (\circ) \alpha \mid \varphi\).

Question 7 (continued)
The typing rules ensure that the evaluation of \(a_0\) \(a_0\) will not allocate a piece of store, hence it can safely be generalize.

Question 8
A program typable in \(F\) obeys the value-restriction. All value forms should be typable with effect \(\circ\)—and hence still generalizable in \(F_{\text{ref}}\)—while all expressions can be typed with the effect \(\bullet\).
Question 9

We reason by induction on the derivation of $\Gamma \vdash E[a] : \tau' | \circ$. In each case, we start by inversion of the typing judgment.

**Case App (E is \{ a \}_1):** We immediately have $\Gamma \vdash a : \tau_1 \rightarrow \tau | \circ$ for some type $\tau_1$.

**Case App (E is \{ v \}_1):** We immediately have $\Gamma \vdash a : \tau_1 | \circ$ for some type $\tau_1$.

**Case Inst:** The premise is of the form $\Gamma \vdash E[a] : \forall \gamma.\tau_1 | \circ$. We conclude by induction hypothesis.

**Case Gen:** The premise is of the form $\Gamma, \gamma \vdash E[a] : \tau' | \circ$. We conclude by induction hypothesis.

Other cases are not possible: they would contradict the shape of $E$.

Question 10

We show $\mu_1 = \mu_2$ (1) by induction on the proof of the reduction $\mu_1 a_1 \rightarrow a_2 / \mu_2$. By inversion of $\vec{\gamma} \vdash a_1 / \mu_1 : \tau | \circ$ and rule CONFIG, there exists a store typing $\Sigma$ such that $\vec{\gamma}, \Sigma \vdash a_1 : \tau | \circ$ (2) and $\vec{\gamma}, \Sigma \vdash \mu_1 : \Sigma$ (3).

**Case Beta-V:** the property (1) immediately holds.

**Case Context:** Then $a_1$ is $E[a]$. By Question 9 applied to (2), we have $\vec{\gamma}, \Sigma, \vec{\gamma}' \vdash a : \tau' | \circ$ for some type $\tau'$ and variables $\vec{\gamma}'$. By the permutation lemma, we have $\vec{\gamma} \vec{\gamma}', \Sigma \vdash a : \tau' | \circ$. By weakening of (3), we also have $\vec{\gamma} \vec{\gamma}', \Sigma \vdash \mu_1 : \Sigma$. Hence, by rule CONFIG, we have $\vec{\gamma} \vec{\gamma}' \vdash a / \mu_1 : \tau' | \circ$. We conclude by the induction hypothesis.

**Case \delta-rules Ref, Set and Get:** Then $a$ is a full application of a constant, which cannot be typed with the $\circ$ effect. This is in contradiction with (2). So this case cannot occur.

Question 11

The evaluation of pure expressions should not either read the store. We can formalize this by saying that the evaluation will proceed identically when replacing the store with any other store that preserves well-typedness of the configuration (which ensures that the configuration have no dangling pointers):

If $\Gamma \vdash a_1 / \mu_1 : \tau | \circ$ and $a_1 / \mu_1 \rightarrow a_2 / \mu_2$, then for any store $\mu$ so that $\Gamma \vdash a_1 / \mu : \tau | \circ$, we have $a_1 / \mu \rightarrow a_2 / \mu$.

Question 12

It suffices to show that the conclusion is a consequence of the premise in $F_{ref}$. Assume that $\Gamma \vdash a : \tau | \circ$. Let $\gamma$ be a variable that is not in the domain of $\Gamma$. By weakening, we have $\Gamma, \gamma \vdash a : \tau | \circ$. By Rule Gen, we have $\Gamma \vdash a : \forall \gamma.\tau | \varphi$. By Rule Inst, we have $\Gamma \vdash a : \tau | \varphi$ as expected.

(This rule is not derivable in our presentation, because weakening is needed, which requires a non local rearrangement of the typing derivation.)

Question 13

$$\begin{align*}
\text{TApp} & \quad \frac{\Gamma \vdash a_1 : \tau_2 \rightarrow (\varphi) \tau_1 | \varphi \Rightarrow b_1}{\Gamma \vdash a_1 \ a_2 : \tau | \varphi \Rightarrow \begin{cases} b_1 \ b_2 & \text{if } \varphi = \circ \\ \text{bind } b_1 (\text{bind } b_2 (\text{bind } \lambda x_1. b_2 (\lambda x_2. x_1 \ x_2))) & \text{if } \varphi = \bullet \end{cases}}
\end{align*}$$
**Question 14**

Type generalization is translated very much like variables: the term \( a \) is pure, therefore its translation \( b \) is a non-monadic term; all we need to do is inject it into a monadic computation \( \text{ret} b \) if the desired effect is \( \bullet \).

\[
\text{TGen} \quad \frac{\Gamma, \alpha \vdash a : \tau | \circ \Rightarrow b}{\Gamma \vdash a : \forall \alpha.\tau | \varphi \Rightarrow \begin{cases} b & \text{if } \varphi = \circ \\ \text{ret} b & \text{if } \varphi = \bullet \end{cases}}
\]

For type instantiation, the effect is the same in the conclusion and in the premise, and moreover type instantiation has no computational content. Hence, we are tempted to take:

\[
\text{TInst} \quad \frac{\Gamma \vdash a : \forall \alpha.\tau | \varphi \Rightarrow b}{\Gamma \vdash a' : [\alpha \mapsto \tau'] | \varphi \Rightarrow b}
\]

This implements the correct dynamic semantics for the translated program. But, as we will see in question 18, this translation does not preserve typing, and a slightly more involved translation is needed.

**Question 15**

The guiding principle for the optimized monadic translation is that only terms with an impure or potentially impure effect are translated to a monadic computation: pure terms do not use the monad. Therefore, a term of type \( \tau \) and impure effect \( \bullet \) is translated to a monadic computation, which has a type of the form \( \text{mon} \tau' \). The type \( \tau' \) is the type of the value that this monadic computation will eventually return. A term with the same type \( \tau \) but pure effect \( \circ \) is translated to a non-monadic term that directly computes a value of type \( \tau' \). Hence \( \llbracket \tau \rrbracket _\bullet = \text{mon} \llbracket \tau \rrbracket _\circ \).

**Question 16**

\[
\llbracket \tau_1 \rightarrow \langle \varphi \rangle \tau_2 \rrbracket _\circ = \llbracket \tau_1 \rrbracket _\circ \rightarrow \llbracket \tau_2 \rrbracket _\varphi
\]

\[
[x_1 : \tau_1, x_2 : \tau_2, \ldots] = x_1 : \llbracket \tau_1 \rrbracket _\circ, x_2 : \llbracket \tau_2 \rrbracket _\circ, \ldots
\]

**Question 17**

For TLam, assuming that property (1) holds for the premise of the rule, we know

\[
\llbracket \Gamma, x : \tau_1 \rrbracket \vdash b : \llbracket \tau_2 \rrbracket _\varphi
\]

that is,

\[
\llbracket \Gamma, x : \llbracket \tau_1 \rrbracket _\circ \rrbracket \vdash b : \llbracket \tau_2 \rrbracket _\varphi
\]

It follows that

\[
\llbracket \Gamma \rrbracket \vdash \lambda x. b : \llbracket \tau_1 \rrbracket _\circ \rightarrow \llbracket \tau_2 \rrbracket _\varphi = \llbracket \tau_1 \rightarrow \langle \varphi \rangle \tau_2 \rrbracket _\circ
\]

This is the expected result if the source lambda, \( \lambda x. a \), is typed with effect “pure” \( (\varphi_0 = \circ \) in rule TLAM\). In the other cases \( (\varphi_0 = \bullet) \), the translation puts a \( \text{ret} \) around the lambda:

\[
\llbracket \Gamma \rrbracket \vdash \text{ret} (\lambda x. b) : \text{mon} (\llbracket \tau_1 \rrbracket _\circ \rightarrow \llbracket \tau_2 \rrbracket _\varphi) = \llbracket \tau_1 \rightarrow \langle \varphi \rangle \tau_2 \rrbracket _\bullet
\]

For TApp, the premises of the rule imply that

\[
\llbracket \Gamma \rrbracket \vdash b_1 : \llbracket \tau_2 \rightarrow \langle \varphi \rangle \tau_1 \rrbracket _\varphi \quad \llbracket \Gamma \rrbracket \vdash b_2 : \llbracket \tau_2 \rrbracket _\varphi
\]

In the case \( \varphi = \circ \), this simplifies into

\[
\llbracket \Gamma \rrbracket \vdash b_1 : \llbracket \tau_2 \rrbracket _\circ \rightarrow \llbracket \tau_1 \rrbracket _\circ \quad \llbracket \Gamma \rrbracket \vdash b_2 : \llbracket \tau_2 \rrbracket _\circ
\]

from which it follows that the generated application \( b_1 b_2 \) is well typed, with type \( \llbracket \tau_1 \rrbracket _\circ \).
In the other case $\varphi = \bullet$, we have
\[
\Gamma \vdash b_1 : \text{mon} ([\tau_2]_o \rightarrow \text{mon} [\tau_1]_o) \quad \Gamma \vdash b_2 : \text{mon} [\tau_2]_o
\]
We see that the generated monadic computation,
\[
\text{bind } b_1 (\lambda x_1. \text{bind } b_2 (\lambda x_2. x_2))
\]
is well-typed, taking $x_1 : [\tau_2]_o \rightarrow \text{mon} [\tau_1]_o$ and $x_2 : [\tau_2]_o$, and its type is $\text{mon} [\tau_1]_o = [\tau_1]_{\bullet}$, as expected.

**Question 18**

From the premise of $\text{TInst}$, $\Gamma \vdash a : \forall \alpha. \tau$ | $\varphi \Rightarrow b$, we can assume that $\Gamma \vdash b : [\forall \alpha. \tau]_\varphi$.

If $\varphi = \circ$, the type of $b$ is $[\forall \alpha. \tau]_\circ = [\forall \alpha. \tau]_o$ which we can instantiate to $[\alpha \mapsto [\tau']_o][\tau]_o$. It is easy to see that the latter type is equal to $[[\alpha \mapsto \tau']_\circ][\tau]_o$, establishing the desired result.

If $\varphi = \bullet$, we are in trouble. The type of $b$ is $[\forall \alpha. \tau]_{\bullet} = \text{mon} \forall \alpha. [\tau]_o$. In this type, the $\forall$ is not in outermost position, and therefore cannot be instantiated. We need, somehow, to pull the $\forall \alpha. [\tau]_o$ “outside” of the monad, then perform the type instantiation, then reinject the instantiated term into the monad. This can be done by a $\text{bind}$ followed by a $\text{ret}$:
\[
b' = \text{bind } b (\lambda x. \text{ret } x)
\]
The $x$ parameter to the $\lambda$ has type $\forall \alpha. [\tau]_o$. In $\text{ret } x$, it can be instantiated to type
\[
[\alpha \mapsto [\tau']_o][\tau]_o = [[\alpha \mapsto \tau']_\circ][\tau]_o
\]
In the end, $b'$ has type
\[
\text{mon} [[\alpha \mapsto \tau']_\circ][\tau]_o = [[\alpha \mapsto \tau']_\bullet][\tau]_o
\]
as expected. This leads to the following type-preserving translation rule for type instantiation:

\[
\text{TInst'}
\]

\[
\frac{\Gamma \vdash a : \forall \alpha. \tau | \varphi \Rightarrow b}{\Gamma \vdash a' : [\alpha \mapsto \tau']_\circ | \varphi \Rightarrow \begin{cases} b & \text{if } \varphi = \circ \\ \text{bind } b (\lambda x. \text{ret } x) & \text{if } \varphi = \bullet \end{cases}}
\]

The dynamic semantics of the translated term is unchanged with respect to rule $\text{TInst}$, because, by the second law of monads, $\text{bind } b (\lambda x. \text{ret } x)$ is expected to behave exactly like $b$.

**Question 19**

Consider a term $a : \forall \varepsilon. \tau \rightarrow \langle \varepsilon \rangle \tau$. It can be let-bound and later used in a context $E_1[]$ with type $\tau \rightarrow \langle \varepsilon \rangle \tau$ (by instantiation $\varepsilon = \circ$), as well as in a context $E_2[]$ with type $\tau \rightarrow \langle \bullet \rangle \tau$ (by instantiation $\varepsilon = \bullet$):
\[
\text{let } x = a \text{ in } \ldots E_1[x] \ldots E_2[x] \ldots
\]
After monadic translation, the context $E_1$ expects $x$ to be a pure $\text{F}$ function, with type $[\tau]_o \rightarrow [\tau]_o$. However, the context $E_2$ expects $x$ to be a function returning a monadic computation, with type $[\tau]_o \rightarrow \text{mon} [\tau]_o$.

Therefore, we do not know how to translate $a$ in a way that satisfies those two expectations at the same time. For instance, if $a = \lambda x. x$, the translation that $E_1$ expects is $\lambda x. x$, and the translation that $E_2$ expects is $\lambda x. \text{ret } x$.

A similar tension can be seen in the Haskell standard library, where some popular functions like $\text{map}$ over lists are defined twice, once in pure functional style, with type $(\alpha \rightarrow \beta) \rightarrow (\text{list } \alpha \rightarrow \text{list } \beta)$, and once in monadic style, with type $(\alpha \rightarrow \text{mon } \beta) \rightarrow (\text{list } \alpha \rightarrow \text{mon } (\text{list } \beta))$.

A possible solution is to parameterize the monadic translation of effect-polymorphic terms by monads (one monad per effect variable involved), then apply the resulting translation of the form $\lambda x_{\text{monad}, a_{\text{body}}}$ to either the identity monad or the state monad when an effect variable is instantiated to $\circ$ or $\bullet$. However, this requires type parametrization mechanisms that go beyond System $\text{F}$ and include at least System $\text{F}_\omega$. 

12
Question 20

\[
\begin{align*}
\text{Raise} & \quad \frac{}{Γ ⊢ a : \text{exn} | ϕ} \\
& \quad \frac{}{Γ ⊢ \text{raise } a : τ | •} \\
\text{Try} & \quad \frac{Γ ⊢ a_1 : τ | ϕ' \quad Γ ⊢ a_2 : \text{exn} \rightarrow (ϕ) τ | ϕ}{Γ ⊢ \text{try } a_1 \text{ with } a_2 : τ | ϕ}
\end{align*}
\]

Question 21

For exceptions, we don’t need to restrict polymorphism to value forms. Hence, we could have replacing \( \text{Gen} \) by \( \text{Gen-Plus} \) and a primitive version of \( \text{Sub} \) (reminding below).

\[
\begin{align*}
\text{Gen-Plus} & \quad Γ, γ ⊢ a : τ | ϕ \\
& \quad Γ ⊢ a : ∀γ.τ | • \\
\text{Sub} & \quad Γ ⊢ a : τ | • \\
& \quad Γ ⊢ a : ∀γ.τ | • \\
\text{Gen-Minus} & \quad Γ, γ ⊢ a : τ | • \\
& \quad Γ ⊢ a : ∀γ.τ | • \\
\end{align*}
\]

In \( F_{ref} \) we could (and probably should) have already split the original rule \( \text{Gen} \) into a more restrictive version \( \text{Gen-Minus} \) and an independent explicit Rule \( \text{Sub} \).

Question 22

a) \( ⊢ a : \text{exn} | • \)
b) The safety properties then ensures that the evaluation of \( a \) cannot raise an uncaused exception.
c) \( a \rightarrow \text{try } (λx. \text{raise } x) v \text{ with } λz. z \)
\( \rightarrow \text{try } \text{raise } v \text{ with } λz. z \)
\( \rightarrow (λz. z) v \)
\( \rightarrow v \)

Question 23

If \( a \vdash a : τ | ϕ \) and \( a \) is irreducible then \( a \) is either a value or an uncaused exception. Moreover, if \( ϕ \) is \( • \) then \( a \) is a value.

(Formally, it suffices to show that \( a \vdash \text{raise } v : τ | ϕ \) implies that \( ϕ \) is \( • \) by structural induction on its typing derivation: only rules \( \text{Gen} \), \( \text{INST} \) and \( \text{RAISE} \) may end the derivation and all three cases are trivial.)

Question 23 (continued)

Assume that \( a \vdash a : τ | • \) and \( a \) reduces to \( a' \) (in any number of steps). By subject reduction, we have \( a \vdash a' : τ | • \). By the refined version of progress, either \( a' \) is a value or further reduces. In particular, it cannot be an uncaused exception \( \text{raise } v \), which is not a value and does not reduce.

Question 24

One can split effects \( ϕ \) as pairs of single effects \( ψ_1 \star ψ_2 \) where, e.g. \( ψ_1 \) stands for the store effect and \( ψ_2 \) stands for exceptional effects:

\[
ϕ ::= ψ\star ψ \\
ψ ::= ε | • | o
\]

Types of primitives must be adjusted accordingly, as well as the type of \( \text{Gen} \), \( \text{RAISE} \), \( \text{TRY} \). For instance, we have

\[
\begin{align*}
\text{Gen} & \quad Γ, γ ⊢ a : τ | o \star ψ \\
& \quad Γ ⊢ a : ∀γ.τ | ψ' \star ψ \\
\text{Try} & \quad Γ ⊢ a_1 : τ | ψ \star ψ_1 \\
& \quad Γ ⊢ a_2 : \text{exn} \rightarrow τ | ψ \star ψ_2 \\
& \quad Γ ⊢ \text{try } a_1 \text{ with } a_2 : τ | ψ \star ψ_2
\end{align*}
\]