Extensible records

Mid-term exam, MPRI 2-4

2022/30/11 — 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.

The goal of this exam is the formalization, typing, and compilation of extensible records in System F. We recall the definition of System F in Figure 1. It is parameterized by a set $\mathcal{G}$ of type constants and set $\mathcal{C}$ of constants that is split into constructors and destructors. We use $(\mathcal{A})$ to refer to the assumptions made on constants (as detailed in the course) to ensure the soundness of F. We write τ for types, M for explicitly typed terms, Γ for typing environments, Δ for the initial typing environment describing types of constants.

Let $\mathcal{L}$ be a (possibly large) set of ordered labels. When L is a subset of $\mathcal{L}$ and we write an expression of the form $(P)^{\ell\in L}$, we always assume that labels ℓ in L are enumerated in order.

In examples written in OCaml, you may use all features of the language such as pairs, datatypes, let-bindings, pattern matching, *etc.*. In examples, we (explicitly) restrict $\mathcal{L}$ to the set $\{\mathsf{foo}, \mathsf{bar}\}$, which we call foobar for short. identifiers written name_ℓ^L in math are written $\mathsf{name\_L\_}\ell$ in OCaml code, such as $\mathsf{proj\_foobar\_foo}$ which stands for $\mathsf{proj}_{\mathsf{foo}}^{\mathsf{foobar}}$.

1 Simple records

We call *simple records* over $\mathcal{L}$ and write $\mathsf{F}_{\mathcal{L}}^{\mathsf{Simple}}$ for the the instance of F that models OCaml-style record type declarations

$$\mathsf{type}\ \bar{\alpha}_\ell\ \mathsf{rcd}^L = \{(\ell : \alpha_\ell)^{\ell\in L}\}$$

by introducing, for each non-empty subset L of $\mathcal{L}$:

- a new type constant rcd^L of arity $|L|$,
- a new constructor make^L,
- a new family of destructors proj_m^L, where $m \in L$.

Notice that this introduces new constants but no new primitive. Simple records $\mathsf{F}_{\{\mathsf{foo},\mathsf{bar}\}}^{\mathsf{Simple}}$ could be provided in OCaml by a module with the following signature:

Syntax

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \mid g\ \bar{\tau} \\ M & ::= & x \mid \lambda x\!:\!\tau.\, M \mid M\ M \mid \Lambda\alpha.M \mid M\ \tau \mid c \end{array}$$

Typing rules

$$\textsc{Var}\ \frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau} \qquad \textsc{Const}\ \frac{c:\tau \in \Delta}{\Gamma \vdash c:\tau} \qquad \textsc{Abs}\ \frac{\Gamma, x:\tau_1 \vdash M:\tau_2}{\Gamma \vdash \lambda x:\tau_1.\, M:\tau_1 \to \tau_2} \qquad \textsc{Tabs}\ \frac{\Gamma, \alpha \vdash M:\tau}{\Gamma \vdash \Lambda\alpha.M:\forall\alpha.\tau}$$

$$\textsc{App}\ \frac{\Gamma \vdash M_1:\tau_1 \to \tau_2 \qquad \Gamma \vdash M_2:\tau_1}{\Gamma \vdash M_1\ M_2:\tau_2} \qquad \textsc{Tapp}\ \frac{\Gamma \vdash M:\forall\alpha.\tau}{\Gamma \vdash M\ \tau' : \tau[\alpha \leftarrow \tau']}$$

Semantics

$$\begin{array}{rcll} V & ::= & \lambda x:\tau.\, M \mid \Lambda\alpha.V \mid c\ \bar{\tau}\ V_1 \ldots\ V_n & (1) \\ E & ::= & [\,]\ M \mid V\ [\,] \mid [\,]\ \tau \mid \Lambda\alpha.[\,] & \end{array}$$

$$\begin{array}{rcl} (\lambda x:\tau.\, M)\ V & \longrightarrow & M[x \leftarrow V] \\ (\Lambda\alpha.V)\ \tau & \longrightarrow & V[\alpha \leftarrow \tau] \end{array}$$

$$\textsc{Context}\ \frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}$$

(**1**) with $n < arity(c)$ or $n = arity(c)$ when c is a constructor.

Figure 1: Definition of System F.

```
module type SIMPLE = sig
    type α rcd_foo
    val make_foo : α → α rcd_foo
    val proj_foo_foo : α rcd_foo → α
    type α rcd_bar
    val make_bar : α → α rcd_bar
    val proj_bar_bar : α rcd_bar → α
    type (α, β) rcd_foobar
    val make_foobar : α → β → (α, β) rcd_foobar
    val proj_foobar_foo : (α, β) rcd_foobar → α
    val proj_foobar_bar : (α, β) rcd_foobar → β
end
```

Question 1 *Give an implementation of the interface SIMPLE in OCaml (with the expected semantics).* Answer

□

Question 2 *Give the family of δ-rules needed for* $F_{\mathcal{L}}^{Simple}$. Answer

□

Question 3 ***State assumptions*** *on constants* ($\mathcal{A}$) *that must be verified to ensure the soundness of* $F_{\mathcal{L}}^{Simple}$ *and* ***prove them****. You may assume that standard auxiliary lemmas hold, provided you clearly name them.* Answer

□

Question 4 *How would you add empty records to* $F_{\mathcal{L}}^{Simple}$*? Would this extension still be*

safe (no justification is needed)? Answer

□

2 Primitive records

We consider an alternative language $\mathsf{F}_{\mathcal{L}}^{\mathsf{Prim}}$ that *extends* F with *primitive records*. The syntax and typing rules are extended as follows, where L ranges over all non-empty subsets of $\mathcal{L}$:

$$\tau ::= \ldots \mid \{(\ell : \tau_\ell)^{\ell \in L}\} \qquad M ::= \ldots \mid \{(\ell = M_\ell)^{\ell \in L}\} \mid M \cdot \ell$$

$$\textsc{Record}\ \frac{\forall \ell \in L \quad \Gamma \vdash M_\ell : \tau_\ell}{\Gamma \vdash \{(\ell = M_\ell)^{\ell \in L}\} : \{(\ell : \tau_\ell)^{\ell \in L}\}} \qquad \textsc{Access}\ \frac{\Gamma \vdash M : \{(\ell : M_\ell)^{\ell \in L}\} \qquad m \in L}{\Gamma \vdash M \cdot \ell : \tau_m}$$

The semantics of primitive records is described as follows:

$$\begin{array}{lcll} V & ::= & \ldots \mid \{(\ell = V)^{\ell \in L}\} & L \subseteq \mathcal{L} \\ E & ::= & \ldots \mid ??? & \\ \multicolumn{4}{c}{\{(\ell = V_\ell)^{\ell \in L}\} \cdot m \longrightarrow V_m \qquad m \in L \subseteq \mathcal{L}} \end{array}$$

Question 5 *Complete the definition of new evaluation contexts.* Answer

□

Question 6 *Informally (and briefly) describe the main difference (other than syntactic) between* $\mathsf{F}_{\mathcal{L}}^{\mathsf{Simple}}$ *and* $\mathsf{F}_{\mathcal{L}}^{\mathsf{Prim}}$*.* Answer

□

Question 7 *Let m be fixed and A be the set $\{\sigma \mid \exists \bar{\alpha}, \tau,\ \emptyset \vdash \Lambda\bar{\alpha}.\lambda x : \tau.\, x \cdot m : \sigma\}$. Give a more explicit description of A.* Answer
We define the instance relation $\preceq$ on types by $\forall \bar{\alpha}.\, \sigma \preceq \forall \bar{\beta}.\, \sigma[\bar{\alpha} \leftarrow \bar{\sigma}']$ when $\bar{\beta} \notin \mathsf{ftv}(\forall \bar{\alpha}.\, \sigma)$. Give a minimal subset B of A so that all elements of A are still instances of an element of B. (B is commonly called a set of principal solutions.) Answer

□

Question 8 *We wish to show type soundness of* $\mathsf{F}_{\mathcal{L}}^{\mathsf{Prim}}$*. Explain why* $\mathsf{F}_{\mathcal{L}}^{\mathsf{Prim}}$ *cannot be seen as an instance of* F *presented in Fig 1 (and also in the course), even when seeing constructs as syntactic sugar and replacing them by applications of constants.*
Answer

□

Question 9 *Describe a small extension of* F *that would make* $\mathsf{F}_{\mathcal{L}}^{\mathsf{Prim}}$ *an instance of* F *after replacing record constructs by applications of constants.* Answer

□

Question 10 *An alternative approach to prove type soundness of $\mathsf{F}_{\mathcal{L}}^{\mathsf{Prim}}$ is to exhibit a translation of $\mathsf{F}_{\mathcal{L}}^{\mathsf{Prim}}$ into $\mathsf{F}_{\mathcal{L}}^{\mathsf{Simple}}$ as an elaboration judgment $\Gamma \vdash M : \tau \rightsquigarrow N$ where $\Gamma \vdash M : \tau$ in $\mathsf{F}_{\mathcal{L}}^{\mathsf{Prim}}$ and N is in $\mathsf{F}_{\mathcal{L}}^{\mathsf{Simple}}$ that is semantics-preserving in the sense that the semantics of a program can be put in close correspondence with the semantics of its translation.*

Propose such a translation together with a translation of types $[\![\cdot]\!]$ and typing contexts so that $\Gamma \vdash M : \tau \rightsquigarrow N$ also implies $[\![\Gamma]\!] \vdash N : [\![\tau]\!]$. (Just give the definitions, you do not have to prove any property about them.) Answer

□

Question 11 *State the fact that the elaboration establishes a forward simulation between $\mathsf{F}_{\mathcal{L}}^{\mathsf{Prim}}$ and $\mathsf{F}_{\mathcal{L}}^{\mathsf{Simple}}$. (Just write the statement that should hold, you do not have to prove it.)* Answer

□

Question 12 *You may assume that the elaboration is deterministic. What other statement is needed to ensure that the elaboration is semantics-preserving? (Give the precise statements that one would have to prove, but do not prove them.)* Answer

□

3 Records with polymorphic access

We now explore an alternative typing for $\mathsf{F}_{\mathcal{L}}^{\mathsf{Prim}}$, which we call $\mathsf{F}_{\mathcal{L}}^{\mathsf{Poly}}$. For that purpose, we temporarily consider the simpler case where $\mathcal{L}$ is $\{\mathsf{foo}, \mathsf{bar}\}$.

We assume given two abstract types, which could be declared in OCaml as

```
type abs
type α pre
```

where pre is injective.[1]

We propose the following interface POLY to model $\mathsf{F}_{\{\mathsf{foo},\mathsf{bar}\}}^{\mathsf{Poly}}$ in OCaml.

```
module type POLY = sig
  type (α, β) rcd
  val make_foo : (*∀α.*) α → (α pre, abs) rcd
  val make_bar : (*∀α.*) α → (abs, α pre) rcd
  val make_foobar : (*∀α.∀β.*) α → β → (α pre, β pre) rcd
  val proj_foo : (*∀α.∀β.*) (α pre, β) rcd → α
  val proj_bar : (*∀α.∀β.*) (α, β pre) rcd → β
end
```

Quantifiers written in comments allow to read back this OCaml interface as types in F with explicit (and ordered) quantifiers.

Question 13 *We wish to show that this is a safe interface for records with polymorphic access. For this purpose, we see make_ declarations as record* constructors *and proj_*

[1] Injectivity means that τ pre $= \sigma$ pre implies $\tau = \sigma$, which ensures that pre can be used to restrict the codomain of a constructor in a GADT definition. In OCaml, we actually need to leave it concrete, e.g. "type 'a pre = P : 'a → 'a pre" so that the typechecker can check that it is indeed injective. However, you should assume that it is abstract and injective.

declarations as record destructors. *You will treat them as explicitly typed. Give the δ-rules that should be used for* $F^{Poly}_{\mathsf{foobar}}$ *in this setting.* Answer

□

Question 14 *Prove that the δ-rules satisfy the subject reduction part of the assumption* ($\mathcal{A}$). *You may assume that standard auxiliary lemmas hold, provided you clearly name them.* Answer

□

Question 15 *Prove the progress property part of the assumption* ($\mathcal{A}$) *for record constants. You may assume that standard auxiliary lemmas hold, provided you clearly name them.* Answer

□

4 A naive implementation

The goal of this section is to provide a safe, although inefficient implementation of records with polymorphic access as a module of interface POLY in OCaml. The semantics should correspond to that described in the previous section. That is: make_foo, make_bar, make_foobar build structures that behave as records of domains {foo}, {bar}, and {foobar}, respectively, while proj_foo and proj_bar behave as their projections on fields foo and bar, respectively.

We begin with a simpler instance of this problem, namely the implementation of a one-field record, called a box, that may or may not be defined. We propose the following unsafe OCaml implementation:

```
module Box_unsafe = struct
  type α box = Empty : α box | Full : α → α box
  let empty = Empty
  let box x = Full x
  let unbox x = match x with Full x → x | Empty → assert false
end
```

Question 16 *Give the OCaml interface BOX_unsafe of this module that leaves the implementation of type box abstract.* Answer

□

Question 17 *Explain why this interface is unsafe.* Answer

□

We may provide a safe implementation of Box by just changing types (and removing the dead branch with the assertion), giving it the following interface:

```
module type BOX = sig
  type α box
  val empty : abs box
  val box : α → α pre box
  val unbox : α pre box → α
end
```

Question 18 *Give a safe implementation Box of interface BOX in OCaml.*

(Hint: you may use a GADT.) Answer

□

Question 19 *Fill in the dots in the skeleton below to provide an (inefficient)* OCaml *implementation of the interface* POLY *that uses* Box *to represent record fields.*

```
module Poly_boxed (Box : BOX) : POLY = struct
  open Box ...
end
```

Answer

□

We thus have a safe, well-type implementation of records with polymorphic access.

```
module Poly_ineff = Poly_boxed (Box)
```

Question 20 *Explain why this implementation is inefficient.* Answer

□

5 A better implementation of polymorphic access

The goal is now to build a better implementation of polymorphic access. For this, we are going to build an OCaml implementation of records with polymorphic access, *i.e.* a structure of signature POLY, that only uses simple records, which are efficient.

We can proceed in two steps: we first define an implementation Poly_simple of polymorphic records that uses records with polymorphic access but only with accesses to records of known domains, as a functor of the form:

```
module Poly_simple (R : POLY) : POLY = struct
  ...
end
```

Let Poly_copy be the following trivial but unsatisfactory implementation of this signature as it just copies the implementation from its argument:

```
module Poly_copy (R : POLY) : POLY = struct
  type (α, β) rcd = (α, β) R.rcd
  let make_foo x = R.make_foo x
  let make_bar x = R.make_bar x
  let make_foobar x y = R.make_foobar x y
  let proj_foo x = R.proj_foo x
  let proj_bar x = R.proj_bar x
end
```

However, we can *transform* this implementation, changing the body of the functor in gray, so as obtain the desired implementation of Poly_simple.

Question 21 *Give an implementation* Poly_simple*. This can be done by a transformation called* concretization *method that is similar to* defunctionalization*: the key step is the new definition of type* rcd *which uses a (GADT) sum type to represent records of different domains.* Answer

□

Question 22 *Finally, transform the code for using simple records instead of* simple uses of *polymorphic records, hence filling in the code in the following skeleton:*

```
module Poly (R : SIMPLE) : POLY = struct
  ...
end
```

Answer

□

Often, polymorphic records also feature record extension, *i.e.* an operation $\{M \text{ with } \ell = N\}$ that extends a record M with a new field m, ignoring the previous value of this field if it is defined). Formally, its semantics is defined by the following reduction rule

$$\{\{(\ell = V_\ell)^{\ell \in L}\} \text{ with } m = V\} \longrightarrow \{(\ell = V_\ell)^{\ell \in L, \ell < m}, m = V, (\ell = V_\ell)^{\ell \in L, \ell > m}\}$$

for each subset L of $\mathcal{L}$ and each label m in L.

Question 23 *Complete the one line below that should be added to the interface* POLY *to describe extension on field* foo *as a new constant:*

```
val with_foo : ...
```

Complete the code

```
let with_foo = ...
```

to be added to (the end of) the implementation of Poly*. (Hence, the code may use prior definitions.)* Answer

□

Question 24 *Should* with_foo *be treated as a destructor or a constructor? (Justify very briefly.)* Answer

□

6 Solutions

Question 1

```
module Simple : SIMPLE = struct
  type α rcd_foo = {foo : α}
  let make_foo x = {foo = x}
  let proj_foo_foo  r = r.foo
  type β rcd_bar = {bar : β}
  let make_bar x = {bar = x}
  let proj_bar_bar  r = r.bar
  type (α, β) rcd_foobar = {foo : α; bar : β}
  let make_foobar x y = {foo = x; bar = y}
  let proj_foobar_foo  r = r.foo
  let proj_foobar_bar  r = r.bar
end
```

(Other solutions are indeed possible using tuples instead of records.)

Question 2

The δ-reduction is composed of all δ-rules

$$\mathsf{proj}^L_m\ (\tau_\ell)^{\ell\in L}\ (\mathsf{make}^L\ (\sigma_\ell)^{\ell\in L}(V_\ell)^{\ell\in L}) \qquad (\delta_{L.m})$$

when L ranges over non-empty subsets of $\mathcal{L}$ and m is in L.

Question 3

The δ-rules should *preserve typings* and *ensure progress for record destructors*. We call δ the union of δ-rules.

Case Subject reduction for δ-rules. if $\Gamma \vdash M : \tau$ and $M \longrightarrow_\delta M'$ then $\Gamma \vdash M' : \tau$.

Proof. Assume $\Gamma \vdash M : \tau$ (**3**) and $M \longrightarrow_{\delta_{L.m}} M'$ (**4**). That is M must be of the form $\mathsf{proj}^L_m\ (\tau_\ell)^{\ell\in L}\ (\mathsf{make}^L\ (\sigma_\ell)^{\ell\in L}\ (V_\ell)^{\ell\in L})$ where $m \in L$. By inversion of typing (1) applied to (3) twice, we have $\Gamma \vdash V_\ell : \tau_\ell$ for every $\ell \in L$, and τ is equal to τ_m. In particular, we have $\Gamma \vdash V_m :: \tau$ as expected.

Case Progress for constants. for any M of the form $\mathsf{proj}^L_m\ (\tau_\ell)^{\ell\in L}\ V$, if $\vdash M : \tau'$ then $M \longrightarrow_\delta M'$ for some M'.

Proof. Assume $\bar\alpha \vdash \mathsf{proj}^L_m\ (\tau_\ell)^{\ell\in L}\ V : \tau$. By construction $m \in L$. By inversion of typing (1), τ must be equal to τ_m and $\bar\alpha \vdash V : \mathsf{rcd}^L\ (\tau_\ell)^{\ell\in L}$. By classification lemma (2), V must be a record value of the form $\mathsf{make}^L\ (\tau_\ell)^{\ell\in L}\ (W_\ell)^{\ell\in L}$. Hence $M \longrightarrow W$ by $\delta_{L.m}$.

(**1**) and (**2**) are standard, auxiliary lemmas.

Question 4

Add a nullary type constant $\mathsf{rcd}^\emptyset$ and a nullary constructor $\mathsf{make}^\emptyset$. No destructor, hence no new reduction rule. This extension is safe.

Question 5

$$\begin{array}{rcll} E & ::= & \ldots \mid E \centerdot m & m \in \mathcal{L} \\ & & \mid \left\{(\ell = V_\ell)^{\ell \in L}, m = E, (\ell = M_\ell)^{\ell \in L'}\right\} & m \notin L \uplus L' \subseteq \mathcal{L} \end{array}$$

Question 6

Primitive records have a single construct $M \centerdot \ell$ for each label ℓ to project all records that are defined on field ℓ, regardless of other fields. By contrast, simple records use a different operation to project records of different domains.

Question 7

$$A \;=\; \{\forall \bar{\alpha}.\, \{\ell : \tau_\ell\}^{\ell \in L} \to \tau_m \;\mid\; L \subseteq \mathcal{L} \;\wedge\; m \in L \;\wedge\; \forall \ell \in L,\ \mathsf{ftv}(\tau_\ell) \subseteq \bar{\alpha}\}$$

Question 7 (continued)

$$B \;=\; \{\forall \bar{\alpha}.\, \{\ell : \alpha_\ell\}^{\ell \in L} \to \alpha_m \mid L \subseteq \mathcal{L}, m \in L\}$$

Question 8

We cannot model record access by a constant with a *unique* type scheme. Indeed, it should have all the types of the previous questions.

Question 9

It suffices to allow type constants to be overloaded, *i.e.* to be given in Δ a set of types instead of a single type. Accordingly, $c : \tau \in \Delta$ should mean $\tau \in \Delta(c)$ in rule Const.

Question 10

Translation of types:

$$[\![\alpha]\!] = \alpha \qquad [\![\tau \to \tau]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!] \qquad [\![\{(\ell : \tau_\ell)^{\ell \in L}\}]\!] = \mathsf{rcd}^L\,(\tau_\ell)^{\ell \in L} \qquad [\![\forall \alpha.\, \tau]\!] = \forall \alpha.\, [\![\tau]\!]$$

Translation of typing contexts:

$$[\![\Gamma, \alpha]\!] = [\![G]\!], \alpha \qquad\qquad [\![\Gamma, x : \tau]\!] = [\![G]\!], x : [\![\tau]\!]$$

Elaboration:

$$\textsc{Elab-Record}\quad \frac{\forall \ell \in L,\ \Gamma \vdash M_\ell : \tau_\ell \rightsquigarrow N_\ell}{\Gamma \vdash \{\ell = M_\ell\}^{\ell \in L} : \{\ell : \tau_\ell\}^{\ell \in L} \rightsquigarrow \mathsf{make}^L (M_\ell)^{\ell \in L}}$$

$$\textsc{Elab-Access}\quad \frac{\Gamma \vdash M : \{(\ell : \tau_\ell)^{\ell \in L}\} \rightsquigarrow N \qquad m \in L}{\Gamma \vdash M.\ell : \tau_m \rightsquigarrow \mathsf{proj}^L_m\ N}$$

$$\textsc{Var}\quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow x} \qquad \textsc{Const}\quad \frac{c : \tau \in \Delta}{\Gamma \vdash c : \tau \rightsquigarrow c}$$

$$\textsc{Elab-Fun}\quad \frac{\Gamma, x : \tau \vdash M : \sigma \rightsquigarrow N}{\Gamma \vdash \lambda x : \tau.\, M : \tau \to \sigma \rightsquigarrow \lambda x : [\![\tau]\!].\, N} \qquad \textsc{Elab-App}\quad \frac{\Gamma \vdash M_1 : \sigma \to \tau \rightsquigarrow N_1 \qquad \Gamma \vdash M_2 : \sigma \rightsquigarrow N_2}{\Gamma \vdash M_1\ M_2 : \tau \rightsquigarrow N_1\ N_2}$$

$$\textsc{Tabs}\quad \frac{\Gamma, \alpha \vdash M : \tau \rightsquigarrow N}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\tau \rightsquigarrow \Lambda\alpha.N} \qquad \textsc{Tapp}\quad \frac{\Gamma \vdash M : \forall\alpha.\tau \rightsquigarrow N}{\Gamma \vdash M\ \tau' : \tau[\alpha \leftarrow \tau'] \rightsquigarrow N\ [\![\tau']\!]}$$

Question 11

$\Gamma \vdash M : \tau \rightsquigarrow N \wedge M \longrightarrow M' \implies \exists N',\ \Gamma \vdash M' : \tau \rightsquigarrow N' \wedge M' \longrightarrow N'.$

Question 12

Values coincide: $\Gamma \vdash M : \tau \rightsquigarrow N \implies (M \text{ value} \iff \text{N value})$. (Hence, backward simulation holds as a corollary.)

Question 13

$$\begin{aligned}
\mathsf{proj_foo}\ \tau\ \sigma\ (\mathsf{make_foo}\ \tau'\ V) &\longrightarrow V && (\delta_{\mathsf{foo.foo}})\\
\mathsf{proj_bar}\ \tau\ \sigma\ (\mathsf{make_bar}\ \tau'\ V) &\longrightarrow V && (\delta_{\mathsf{bar.bar}})\\
\mathsf{proj_foo}\ \tau\ \sigma\ (\mathsf{make_foobar}\ \tau'\ \sigma'\ V\ W) &\longrightarrow V && (\delta_{\mathsf{foobar.foo}})\\
\mathsf{proj_bar}\ \tau\ \sigma\ (\mathsf{make_foobar}\ \tau'\ \sigma'\ V\ W) &\longrightarrow W && (\delta_{\mathsf{foobar.bar}})
\end{aligned}$$

Question 14

We only treat case $\delta_{\mathsf{foobar.foo}}$ as other cases are similar. Assume $\bar\alpha \vdash \mathsf{proj\_foo}\ \tau_1\ \sigma_1\ (\mathsf{make\_foobar}\ \tau_2\ \sigma_2\ V\ W) : \tau$ (**2**). We must show $\bar\alpha \vdash V : \tau$ (**3**). By inversion of typing (1) applied to (2), we have, successively, $\bar\alpha \vdash \mathsf{make\_foobar}\ \tau_2\ \sigma_2\ V\ W : (\tau\ \mathsf{pre}, \sigma)\ \mathsf{rcd}$ with τ_1 equal to τ, then $\bar\alpha \vdash V : \tau$, *i.e.* (3) as expected.

(**1**) is a standard lemma.

Question 15

Assume that M is well-typed full applications of a records destructor. There are two destructors proj_foo and proj_bar. We only consider a full application of proj_foo, as the prof is similar for proj_bar. Hence M is the form proj_foo τ σ V and such that $\bar\alpha \vdash M : \tau_0$ (**3**). We must show that M reduces (**4**). By inversion of typing (1) applied to (3) we have $\bar\alpha \vdash V : (\tau\ \mathsf{pre}, \sigma)\ \mathsf{rcd}$. By the classification lemma (2), either

- V is of the form $\mathsf{make\_foo}\ \tau'\ W$ and M reduces by $(\delta^{\mathsf{foo}}_{\mathsf{foo}})$, or
- V is of the form $\mathsf{make\_foobar}\ \tau'\ \sigma'\ W\ W'$. and M reduces by $(\delta^{\mathsf{foobar}}_{\mathsf{foo}})$.

In both cases, M reduces, which proves (4).

(**1**) and (**2**) are omitted, standard lemmas.

Question 16

```
module type BOX_unsafe = sig
    type α box
    val empty : α box
    val box : α → α box
    val unbox : α box → α
end
```

Question 17

This implementation is unsafe because the assertion fails during the evaluation of the well-typed expression unbox empty.

Question 18

```
module Box : BOX = struct
  type α box = Empty : abs box | Full : α → α pre box
  let empty = Empty
  let box x = Full x
  let unbox x = match x with Full x → x
end
```

Question 19

```
module Poly_boxed (Box : BOX) : POLY = struct
  open Box
  type (α, β) rcd = { foo : α box; bar : β box }
  let make_foo (x : α) = {foo = box x; bar = empty}
  let make_bar y = {foo = empty; bar = box y}
  let make_foobar x y = { foo = box x; bar = box y }
  let proj_foo (type a) (r : (a pre, β) rcd) = unbox r.foo
  let proj_bar (type b) (r : (α, b pre) rcd) = unbox r.bar
end
```

Question 20

It allocates memory proportional to $\mathcal{L}$ for each record while a memory proportional to the size of its domain L should suffice.

Question 21

```
module Poly_simple (R : POLY) : POLY = struct
  type (α, β) rcd =
      | Hfoo    : (α pre, abs)     R.rcd → (α pre, abs) rcd
      | Hbar    : (abs, β pre)     R.rcd → (abs, β pre) rcd
      | Hfoobar : (α pre, β pre) R.rcd → (α pre, β pre) rcd
  let make_foo x = Hfoo (R.make_foo x)
  let make_bar x = Hbar (R.make_bar x)
  let make_foobar x y = Hfoobar (R.make_foobar x y)
  let proj_foo (type a) (type b) (x : (a pre, b) rcd) =
     match x with Hfoo x → R.proj_foo x | Hfoobar x → R.proj_foo x
  let proj_bar (type a) (type b) (x : (a, b pre) rcd) =
     match x with Hbar x → R.proj_bar x | Hfoobar x → R.proj_bar x
end
```

Question 22

```
module Poly (R : SIMPLE) : POLY = struct
  type (α, β) rcd =
      | Hfoo    : α R.rcd_foo            → (α pre, abs) rcd
      | Hbar    : β R.rcd_bar            → (abs, β pre) rcd
      | Hfoobar : (α, β) R.rcd_foobar → (α pre, β pre) rcd
  let make_foo x = Hfoo (R.make_foo x)
  let make_bar x = Hbar (R.make_bar x)
  let make_foobar x y = Hfoobar (R.make_foobar x y)
  let proj_foo (type a) (type b) (r : (a pre, b) rcd) =
     match r with Hfoo r → R.proj_foo_foo r | Hfoobar r → R.proj_foobar_foo r
  let proj_bar (type a) (type b) (r : (a, b pre) rcd) =
     match r with Hbar r → R.proj_bar_bar r | Hfoobar r → R.proj_foobar_bar r
end
```

Question 23

```
val with_foo : (α, β) rcd → γ → (γ pre, β) rcd

let with_foo (type a) (type b) (r : (a, b) rcd) x : (γ pre, b) rcd =
  match r with
  | Hfoo _    → make_foo x
  | Hbar _    → make_foobar x (proj_bar r)
  | Hfoobar _ → make_foobar x (proj_bar r)
```

Question 24

As a destructor, since it reduces when fully applied.