Exploring Type Containment

Mid-term exam for MPRI 2-4 course / Examen partiel du cours MPRI 2-4

2015/12/04 — Duration / durée: 2h30

Answers will also be judged by their clarity and concision.

Our goal is to study type containment, some variations on the type-containment relation, and how it can be extended to other programming language features such as products, side effects, existential types. We will study some language meta-theoretical properties.

1 Preliminaries / Préliminaires

We use the same conventions as in the course. We write \( a \) (resp. \( M \)) for implicitly (resp. explicitly) typed expressions. Implicitly typed expressions and types are

\[
a ::= x \mid \lambda x. a \mid a a
\]

We recall the typing rules of the implicitly typed version of System \( F \).

\[
\begin{array}{c}
\text{VAR} \\
\text{GEN} \\
\text{ABS} \\
\text{APP} \\
\text{SUB}
\end{array}
\begin{array}{c}
x : \tau \in \Gamma \\
\Gamma, \alpha \vdash a : \tau \\
\Gamma, x : \tau \vdash a : \sigma \\
\Gamma \vdash a : \tau \\
\Gamma \vdash a : \alpha. \tau \\
\Gamma \vdash \lambda x. a : \tau \rightarrow \sigma
\end{array}
\begin{array}{c}
\Gamma \vdash a : \tau \\
\tau \rightarrow \sigma \\
\forall \alpha. \tau
\end{array}
\begin{array}{c}
\tau \leq \tau' \leq \sigma' \\
\tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'
\end{array}
\begin{array}{c}
\forall \alpha. \sigma \leq \forall \beta. \sigma[\alpha \mapsto \tau] \\
\tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'
\end{array}
\begin{array}{c}
\Gamma \vdash a : \tau \\
\tau \leq \sigma \\
\forall \alpha. \tau \leq \forall \alpha. \sigma \\
\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)
\end{array}
\begin{array}{c}
\Gamma \vdash a : \tau \\
\Gamma \vdash a' : \tau \\
\tau \leq \sigma \\
\ftv(\tau) \in \text{dom}(\Gamma)
\end{array}
\begin{array}{c}
\Gamma \vdash a : \tau \\
\Gamma \vdash a : \sigma \\
\alpha \notin \ftv(\tau)
\end{array}
\begin{array}{c}
\tau_1 \leq \tau_2 \\
\tau_2 \leq \tau_3 \\
\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)
\end{array}
\begin{array}{c}
\forall \alpha. \sigma \leq \forall \beta. \sigma[\alpha \mapsto \tau] \\
\tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'
\end{array}
\begin{array}{c}
\forall \alpha. \tau \leq \forall \alpha. \sigma \\
\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)
\end{array}
\begin{array}{c}
\forall \alpha. \sigma \leq \forall \beta. \sigma[\alpha \mapsto \tau] \\
\tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'
\end{array}
\begin{array}{c}
\forall \alpha. \tau \leq \forall \alpha. \sigma \\
\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)
\end{array}

Our presentation \( F^\leq \) is however parameterized by a type-containment relation \( \leq \). We write \( \ftv(\tau) \) for the set of free type variables of \( \tau \). For System \( F \), \( \leq \) is defined by the following Rule \text{InstGen} alone. In the course, we defined \( F^\eta \) as \( F(\leq) \) where \( \leq \) is the smallest relation satisfying the following rules:

\[
\begin{array}{c}
\text{InstGen} \\
\text{Arrow} \\
\text{All} \\
\text{Trans} \\
\text{Dist-Right}
\end{array}
\begin{array}{c}
\beta \notin \ftv(\forall \alpha. \sigma) \\
\tau \leq \tau' \leq \sigma' \\
\forall \alpha. \tau \leq \forall \alpha. \sigma \\
\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)
\end{array}
\begin{array}{c}
\tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'
\end{array}
\begin{array}{c}
\forall \alpha. \tau \leq \forall \alpha. \sigma \\
\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)
\end{array}
\begin{array}{c}
\alpha \notin \ftv(\tau)
\end{array}
\begin{array}{c}
\tau_1 \leq \tau_2 \\
\tau_2 \leq \tau_3 \\
\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)
\end{array}
\begin{array}{c}
\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)
\end{array}
\begin{array}{c}
\forall \alpha. \sigma \leq \forall \beta. \sigma[\alpha \mapsto \tau] \\
\tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'
\end{array}
\begin{array}{c}
\forall \alpha. \tau \leq \forall \alpha. \sigma \\
\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)
\end{array}
\begin{array}{c}
\forall \alpha. \sigma \leq \forall \beta. \sigma[\alpha \mapsto \tau] \\
\tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'
\end{array}
\begin{array}{c}
\forall \alpha. \tau \leq \forall \alpha. \sigma \\
\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)
\end{array}
\begin{array}{c}
\forall \alpha. \sigma \leq \forall \beta. \sigma[\alpha \mapsto \tau] \\
\tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'
\end{array}
\begin{array}{c}
\forall \alpha. \tau \leq \forall \alpha. \sigma \\
\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)
\end{array}

Question 1

Show that Rule Dist-Right is actually an equivalence, i.e. whenever \( \alpha \notin \ftv(\tau) \), we also have \( \tau \rightarrow \forall \alpha. \sigma \leq \forall \alpha. (\tau \rightarrow \sigma) \).

Montrez que la règle Dist-Right est en fait une équivalence, i.e. lorsque \( \alpha \notin \ftv(\tau) \), nous avons également \( \tau \rightarrow \forall \alpha. \sigma \leq \forall \alpha. (\tau \rightarrow \sigma) \).
The kernel of \( \preceq \) is the equivalence relation \( (\preceq) \cap (\succeq) \), which we write \( \approx \).

**Question 2**

Show that \( \approx \) contains the elimination of useless binders, i.e. \( \forall \alpha. \tau \approx \tau \) whenever \( \alpha \notin \tau \).

Montrez que \( \approx \) contient l’élimination des lieurs inutiles, i.e. \( \forall \alpha. \tau \approx \tau \) dès que \( \alpha \notin \tau \).

**Question 3**

What other equivalence (besides those of Question 1 and Question 2) is in \( \approx \) (no proof is required)? Define a rewriting relation \( \rightsquigarrow \) that puts types in canonical form for \( \approx \).

Quelle autre équivalence (à part celles des Question 1 et Question 2) est dans \( \approx \) (aucune preuve n’est demandée)? Définissez une relation de réécriture \( \rightsquigarrow \) qui met les types en forme canonique pour \( \approx \).

We write \( \text{ftv}^+ (\tau) \) (resp. \( \text{ftv}^- (\tau) \)) for the subset of free type variables that occur positively (resp. negatively) in \( \tau \). We also write \( \text{ftv}^\oplus (\tau) \) for free type variables that occurs only positively in \( \tau \), i.e. \( \text{ftv}^+ (\tau) \setminus \text{ftv}^- (\tau) \).

Nous notons \( \text{ftv}^+ (\tau) \) (resp. \( \text{ftv}^- (\tau) \)) le sous-ensemble des variables de types libres qui apparaissent positivement (resp. négativement) dans \( \tau \). Nous notons également \( \text{ftv}^\oplus (\tau) \) les variables de types libres qui apparaissent seulement positivement dans \( \tau \), i.e. \( \text{ftv}^+ (\tau) \setminus \text{ftv}^- (\tau) \).

\[
\begin{align*}
\text{ftv}^+ (\alpha) &= \{ \alpha \} \\
\text{ftv}^+ (\tau \rightarrow \sigma) &= \text{ftv}^- (\tau) \cup \text{ftv}^+ (\sigma) \\
\text{ftv}^+ (\forall \alpha. \tau) &= \text{ftv}^+ (\tau) \setminus \{ \alpha \}
\end{align*}
\]

\[
\begin{align*}
\text{ftv}^- (\alpha) &= \emptyset \\
\text{ftv}^- (\tau \rightarrow \sigma) &= \text{ftv}^+ (\tau) \cup \text{ftv}^- (\sigma) \\
\text{ftv}^- (\forall \alpha. \tau) &= \text{ftv}^- (\tau) \setminus \{ \alpha \}
\end{align*}
\]

**Question 4**

We write \( \bot \) for \( \forall \beta. \beta \). Show that

\[
\tau [\alpha \mapsto \bot] \preceq \tau \quad \text{if} \quad \alpha \notin \text{ftv}^- (\tau)
\]

and

\[
\tau \preceq \tau [\alpha \mapsto \bot] \quad \text{if} \quad \alpha \notin \text{ftv}^+ (\tau)
\]

**Question 5**

In fact, Rule Dist-Right is itself unnecessarily too general and can be replaced by the more restrictive rule Dist-Right-Neg:

\[
\begin{align*}
\text{Dist-Right-Neg} & \\
\alpha & \notin \text{ftv}(\tau) & \alpha & \in \text{ftv}^- (\sigma) \\
\forall \alpha. (\tau \rightarrow \sigma) \preceq \tau \rightarrow (\forall \alpha. \sigma)
\end{align*}
\]

Show that Rule Dist-Right-Only-Pos is admissible without using rules Dist-Right nor Dist-Right-Neg.

En fait, la règle Dist-Right est elle-même inutillement trop générale et peut être remplacée par la règle plus restrictive Dist-Right-Neg :

\[
\begin{align*}
\text{Dist-Right-Only-Pos} & \\
\alpha & \notin \text{ftv}(\tau) & \alpha & \in \text{ftv}^\oplus (\sigma) \\
\forall \alpha. (\tau \rightarrow \sigma) \preceq \tau \rightarrow (\forall \alpha. \sigma)
\end{align*}
\]

Montrez que la règle Dist-Right-Only-Pos est admissible sans utiliser les règles Dist-Right ni Dist-Right-Neg.

**2 Type containment with pairs**

We now consider an extension of System \( F \) with pairs.

Nous considérons maintenant une extension du Système \( F \) avec des paires.

\[
\begin{align*}
\tau & ::= \ldots \mid \tau \times \tau' & a & ::= \ldots \mid (a, a) \mid \pi_i a \\
\frac{\Gamma \vdash a_1 : \tau_1 \quad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1 \times a_2 : \tau_1 \times \tau_2} & \quad \frac{\Gamma \vdash a : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i a : \tau_i}
\end{align*}
\]
Question 6
How would you extend the type containment relation? (Give the new rules without justification.)

We wish to show that type containment is coherent in the following sense: if \( \tau \leq \sigma \) then it cannot be the case that \( \tau \) and \( \sigma \) have two different top constructors. However, \( \tau \) or \( \sigma \) may not have a top constructor. So, we define

\[
\text{top}(\alpha) = \alpha \\
\text{top}(\forall \alpha. \tau) = \perp \\
\text{top}(\tau \neq \alpha) = \text{top}(\tau) \\
\text{top}(\tau \to \sigma) = (\to) \\
\text{top}(\tau \times \sigma) = (\times)
\]

Question 7
Show that

\[ \tau \leq \sigma \implies \text{top}(\tau) = \text{top}(\sigma) \lor \text{top}(\tau) = \perp \]

3 Type containment and effects

The language is equipped with a small-step call-by-value reduction semantics. We now extend the language with references. Hence, we use a store semantics and restrict generalization to non-expansive expressions \( u \).

\[
a ::= \ldots | c \\
c ::= \ell | \text{ref} | (!) | (:) \\
u ::= x | v
\]

Question 8
Tell for each constant whether it is a constructor or a destructor.

Dites pour chaque constante si c’est un constructeur ou un destructeur.

Question 9
Give the grammar for values \( v \).

Donnez la grammaire des valeurs \( v \).

Question 10
Give an example showing that type containment is unsafe in the presence of side effects. (Justify.)

Donnez un exemple montrant que le type containment est dangereux en présence d’effets de bords. (Justifiez.)

Question 11
We keep references but remove Rule Dist-Right. Show that the following typing rule is admissible:

\[
\frac{\Gamma, \alpha \vdash a : \tau \quad \alpha \in \text{ftv}^\tau(\tau)}{\Gamma \vdash a : \forall \alpha. \tau}
\]
Question 12

We assume that the language $F^\eta$ without Dist-Right is sound in the presence of effects. Is Rule Gen+ sound in System $F$? Même question dans ML?

4 Type containment with existential types

We now consider an extension of $F^\eta$ with existential types (ignoring products for the moment):

$$
\tau ::= \ldots | \exists \alpha. \tau \\
a ::= \ldots | \text{let } x = a \text{ in } a
$$

$$
\Gamma \vdash a : \tau[\alpha \mapsto \tau'] \\
\Gamma \vdash a : \exists \alpha. \tau \\
\Gamma, \alpha, x : \alpha \vdash a' : \tau' \\
\Gamma \vdash \tau'
$$

Question 13

Regarding type containment for existential types, what would you propose for the dual of rules InstGen and All?

Question 14

As for distributivity, we could add the equivalence:

$$(\exists \alpha. \tau) \to \tau' \approx \forall \alpha. (\tau \to \tau') \text{ if } \alpha \notin \text{ftv}(\tau')$$

Justify this equivalence by exhibiting for both directions a term of System $F$ that is $\beta\eta$-equivalent to the identity taking an expression of one type to the other type. Use explicitly typed terms instead of writing their typing derivation.

Question 15

Can you do the same for the equivalence:

$$
\exists \alpha. (\tau \to \tau') \approx? (\forall \alpha. \tau) \to \tau' \text{ if } \alpha \notin \text{ftv}(\tau')
$$

In the rest of this section, we assume that there are both existential types and products. We ignore distributivity rules. We also assume constants (unit, integers, . . . ). We will use existential types to simulate width subtyping between $n$-ary products ($a_1, \ldots, a_n$), which in System $F$ have the following typing rule:

$$
\Gamma \vdash a_1 : \tau_1 \quad \ldots \quad \Gamma \vdash a_n : \tau_n \\
\Gamma \vdash (a_1, \ldots, a_n) : \Pi(\tau_1, \ldots, \tau_n)
$$

4
Question 16

Give the typing rule in System $F$ for the projection $\pi_i a$ that returns the $i$-th component of an $n$-ary product $a$ provided $i$ is in $1..n$.

Has the function $(\lambda x. \pi_i x)$ a principal type in System $F$?

Question 17

Propose a uniform encoding $[(a_1, \ldots, a_n)]$ of tuples into pairs so that the encoding of the projection $[\pi_i]$ has a principal type in System $F$. Give both the encodings and their types.

Question 18

Show that the encoding of a tuple with $n$ fields can also be typed in $F^\eta$ as if it were a truncated tuple with only the first $i$ fields.

Question 19

We wish to model subtyping, but in the absence of polymorphism, for instance to be used in ML. We introduce constants $\perp$ for $\forall \alpha. \alpha$ and $\top$ for $\exists \alpha. \alpha$ and then restrict type containment to monomorphic types, but we keep $\perp$ and $\top$.

Give a direct definition of $\leq$ between monomorphic types.

5 Explicit type-containment

We now study the meta-theoretical properties of the language, which is easier on its explicitly typed version. We introduce coercions to witness the use of type containment in expressions. For the sake of simplicity, we ignore distributivity.

We thus write coercions inside explicitly-typed terms. We include primitive products, but leave constants out for the sake of brevity.

\[
\varphi, \psi ::= \Lambda \vec{\alpha} \cdot \vec{\tau} \mid \varphi, \tau \rightarrow \varphi \mid \varphi \times \varphi \mid \forall \alpha. \varphi \mid \varphi; \varphi
\]

\[
\text{InstGen} \quad \Gamma \vdash \forall \vec{\alpha}. \sigma \quad \Gamma, \vec{\beta} \vdash \vec{\tau} \\
\Gamma \vdash \Lambda \vec{\beta} \cdot \vec{\tau} : \forall \vec{\alpha}. \sigma \leq \forall \vec{\beta}. \sigma[\vec{\alpha} \mapsto \vec{\tau}]
\]

\[
\text{Arrow} \quad \Gamma \vdash \varphi : \tau' \leq \tau \quad \Gamma \vdash \psi : \sigma \leq \sigma' \\
\Gamma \vdash \varphi, \tau' \rightarrow \psi : \tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'
\]

\[
\text{Prod} \quad \Gamma \vdash \varphi : \tau \leq \tau' \quad \Gamma \vdash \psi : \sigma \leq \sigma' \\
\Gamma \vdash \varphi \times \psi : \tau \times \sigma \leq \tau' \times \sigma'
\]

\[
\text{All} \quad \Gamma, \alpha \vdash \varphi : \tau \leq \sigma \\
\Gamma \vdash \forall \alpha. \varphi : \forall \alpha. \tau \leq \forall \alpha. \sigma
\]

\[
\text{Trans} \quad \Gamma \vdash \varphi : \tau_1 \leq \tau_2 \quad \Gamma \vdash \psi : \tau_2 \leq \tau_3 \\
\Gamma \vdash \varphi; \psi : \tau_1 \leq \tau_3
\]

Nous écrivons donc des coercions à l’intérieur des termes explicitement typés. Nous incluons des produits primitifs, mais excluons les constantes par souci de concision.
\[ M ::= x \mid \lambda x. M \mid M M \mid \Lambda \alpha. M \mid \langle \varphi \rangle M \]

\[
\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \langle \varphi \rangle M : \tau'}\]

Values and evaluation contexts:

\[ V ::= \lambda x : \tau. M \mid \Lambda \alpha. V \mid (V, V) \]

\[ E ::= [\ ] M [\ ] | (\ ], \ ] M | \ ] M | \ ] M | \ ] \Lambda \alpha. [\ ] | \ ] \langle \varphi \rangle [\ ] \]

Some reduction rules:

- **R-Context**
  \[
  \frac{M_1 \rightarrow M_2}{E[M_1] \rightarrow E[M_2]}\]

- **R-Beta-Arrow**
  \[
  (\lambda x : \tau. M) V \rightarrow M[x \rightarrow V]
  \]

- **R-Beta-Prod**
  \[
  \pi_i (V_1, V_2) \rightarrow V_i
  \]

- **R-Iota-InstGen**
  \[
  \langle \Lambda \vec{\beta} \cdot \vec{\tau} \rangle (\Lambda \vec{\alpha}. V) \rightarrow \Lambda \vec{\beta}. V[\vec{\alpha} \rightarrow \vec{\tau}]
  \]

- **R-Iota-Arrow**
  \[
  \langle \varphi : \sigma \rightarrow \psi \rangle (\lambda x : \tau. M) \rightarrow \lambda x : \sigma. \langle \psi \rangle M[x \rightarrow \langle \varphi \rangle x]
  \]

In the following questions, you may use standard helper lemmas provided you clearly cite them.

**Question 20**

Give the missing reduction rules.

**Question 21**

State the subject reduction lemma and prove the case Red-Iota-Arrow.

**Question 22**

State the progress lemma and prove it. Only give the cases for pairs and coercions.

**Question 23**

Show that there cannot be an infinite number of ι-reduction steps without a β-reduction step. Why is this important?
6 Solutions

Question 1
This is proved by the following derivation:

\[
\begin{array}{c}
\text{InstGen} \\
\tau \rightarrow \forall \alpha. (\tau \rightarrow \forall \alpha. \sigma) \\
\forall \alpha. (\tau \rightarrow \forall \alpha. \sigma) \leq \forall \alpha. (\tau \rightarrow \sigma) \\
\tau \rightarrow \forall \alpha. \sigma \leq \forall \alpha. (\tau \rightarrow \sigma)
\end{array}
\]

Question 2
Both directions are proved with Rule InstGen. In the direct way, we use the substitution \([\alpha \mapsto \alpha]\) and \(\bar{\beta}\) is empty. In the inverse way, we use the empty substitution and \(\bar{\beta}\) is \(\alpha\).

Question 3
The permutation of binders are also part of \(\approx\). We define \(\Rightarrow\) by the following rules:

\[
\begin{align*}
\alpha \notin \text{ftv}(\tau) & \quad \Rightarrow \quad \tau \rightarrow \forall \alpha. \sigma \Rightarrow \forall \alpha. (\tau \rightarrow \sigma) \\
\forall \alpha. \tau \Rightarrow \tau & \quad \Rightarrow \quad \forall \beta. \forall \alpha. \tau \Rightarrow \forall \alpha. \forall \beta. \tau \\
C[\tau] & \Rightarrow \quad C[\sigma]
\end{align*}
\]

where the order of appearance is from left to right (for example) and \(C\) is any type context.

Question 4
Let \(\leq_+\) be \(\leq\) and \(\leq_-\) be \(\geq\). We let \(\epsilon\) range over \(\{+,-\}\) and write \(\bar{\epsilon}\) for the inverse of the sign of \(\epsilon\). We can factor the two cases as

\[
\tau \leq_\epsilon \tau[\alpha \mapsto \bot] \quad \text{if} \quad \alpha \notin \text{ftv}(\tau)
\]

The proof is by induction on the size of \(\tau\). When \(\alpha\) is a variable, the case is empty if \(\epsilon\) is +; otherwise, the conclusion is \(\bot \leq_+ \tau\) which follows from Rule InstGen. If \(\tau\) if of the form \(\forall \beta. \sigma\), we may assume w.l.o.g. that \(\beta\) and \(\alpha\) are distinct. We conclude by induction hypothesis applied to \(\sigma\) and rule All. If \(\tau\) is of the form \(\sigma' \rightarrow \sigma\), then \(\alpha \notin \text{ftv}(\sigma') \cup \text{ftv}(\sigma)\). By induction hypothesis, we have \(\sigma'[\alpha \mapsto \bot] \leq_\epsilon \sigma'\) and \(\sigma \leq_\epsilon \sigma[\alpha \mapsto \bot]\); thus we may conclude by rule Arrow.

Question 5
Assume \(\alpha \in \text{ftv}(\sigma)\), which implies \(\alpha \notin \text{ftv}^-(\sigma)\). We write \(\theta\) for \([\alpha \mapsto \bot]\) On the one hand, we have \(\sigma\theta \leq_\epsilon \sigma\) by Question 4. Thus by All, we have \(\sigma\theta \leq \forall \alpha. \sigma\). (Since since \(\alpha\) is not free in \(\forall \alpha. \sigma\), we have \(\sigma\theta \approx \forall \alpha. \sigma\\).) By rule Arrow, we then have \(\tau \rightarrow \sigma \theta \leq \tau \rightarrow (\forall \alpha. \sigma)\) (4). On the other hand, we have \(\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow \sigma \theta\) (2) by InstGen. We conclude \(\forall \alpha. (\tau \rightarrow \sigma) \leq \tau \rightarrow (\forall \alpha. \sigma)\) by transitivity of (2) and (4).

Question 6
We need a congruence rule and a rule pushing quantifiers inside. (Moving quantifiers outside would be derivable exactly as on the right-hand sides of arrow types.)

\[
\begin{array}{c}
\text{Prod} \\
\tau \leq \tau' \quad \sigma \leq \sigma' \\
\tau \times \sigma \leq \tau' \times \sigma'
\end{array}
\]

\[
\begin{array}{c}
\text{Dist-Prod} \\
\forall \alpha. (\tau \times \sigma) \leq (\forall \alpha. \tau) \times (\forall \alpha. \sigma)
\end{array}
\]
Question 7

The proof is by induction on the derivation of $\tau \leq \sigma$.

Case InstGen: If $\top(\tau)$ is $\perp$, then the property holds by definition. Otherwise, if $\top(\tau)$ is a variable $\beta$, then this variable is not bound and $\tau_2$ will be the same variable $\beta$, so the property holds. Else, $\top(\tau)$ and $\top(\sigma)$ are the same.

Case Arrow, Dist-Right: Both sides have the arrow top symbol.

Case Prod and Dist-Prod: Both sides have the product top symbol.

Case All: By induction hypothesis, the property holds for the premise. If the left-hand side of the premise has the top symbol $\perp$, then so does $\tau$. Otherwise both sides of the premise have the same top symbol, and so do $\tau$ and $\sigma$.

Case Trans: The premises are $\tau \leq \rho$ and $\rho \leq \sigma$. If $\top(\tau)$ is $\perp$ then the conclusion holds. Otherwise by induction hypothesis, we have $\top(\tau) = \top(\rho)$ (1), which implies that $\top(\rho)$ is not $\perp$. Hence by induction hypothesis, we have $\top(\rho) = \top(\sigma)$ (2). Combined with (1), this implies $\top(\tau) = \top(\sigma)$.

Question 8

Locations $\ell$ are constructors. All store operations are destructors.

Question 9

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

Question 10

The type of $\text{ref}$ is $\forall \alpha. \alpha \rightarrow \text{ref} \alpha$. By Rule InstGen, we may use it at the instance $\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \text{ref} (\alpha \rightarrow \alpha)$, which is a sound type. Then, by Rule Dist-Right, we could assign it type $(\forall \alpha. (\alpha \rightarrow \alpha)) \rightarrow (\forall \alpha. \text{ref} (\alpha \rightarrow \alpha))$, which is unsound. So Rule Dist-Right is to be blamed.

Indeed, this would allow passing the identity function of type $\forall \alpha. \alpha \rightarrow \alpha$ and get back a polymorphic reference of type $\forall \alpha. \text{ref} (\alpha \rightarrow \alpha)$. From there, we could store not of type $\text{bool} \rightarrow \text{bool}$ in the reference and read it back as a value of type $\text{int} \rightarrow \text{int}$ and apply it to 0, which would lead to the stuck program not 0.

Question 11

It suffices to show that we can prove the conclusion from the premises without using Dist-Right. Assume $\Gamma, \alpha \vdash a : \tau$ (1) and $\alpha \in \text{ftv}(\beta)$ (2). By type substitution applied to (1) we have $\Gamma \vdash a : \tau[\alpha \mapsto \perp]$ (3). By Question 4 applied to (2), we have $\tau[\alpha \mapsto \perp] \leq \tau$. Hence, by All, $\forall \alpha. \tau[\alpha \mapsto \perp] \leq \forall \alpha. \tau$ (1). Since $\alpha$ does not appear in $\tau[\alpha \mapsto \perp]$, we have $\forall \alpha. \tau[\alpha \mapsto \perp] \approx \tau[\alpha \mapsto \perp]$, hence $\tau[\alpha \mapsto \perp] \leq \forall \alpha. \tau$ (4). By Sub applied to (3) and (4), we have $\Gamma \vdash a : \forall \alpha. \tau$.

Question 12

This rule is safe in $F^\eta$ without Dist-Right, so it is safe in System $F$ as well, which is a strict subset. For the same reason, it is also safe in ML. It actually implements (a restriction of) the solution called “the relaxed value restriction” used in OCaml.

Question 13

\[
\begin{align*}
\text{Inst-Exi} & \quad \beta \notin \text{ftv}(\exists \alpha. \tau) \\
\exists \beta. \tau[\alpha \mapsto \vec{\tau}] & \leq \exists \alpha. \tau
\end{align*}
\]

\[
\begin{align*}
\text{Exi} & \quad \tau_1 \leq \tau_2 \\
\exists \alpha. \tau_1 & \leq \exists \alpha. \tau_2
\end{align*}
\]
Question 14

\[ g \overset{\text{def}}{=} \lambda f : (\exists \alpha. \tau) \rightarrow \tau'. \lambda \alpha. \lambda x : \tau. f (\text{pack } x \text{ as } \exists \alpha. \tau) \]

\[ g^{-1} \overset{\text{def}}{=} \lambda f : \forall \alpha. (\tau \rightarrow \tau'). \lambda x : \forall \alpha. \tau. \text{let } x' = \beta = \text{unpack } x \text{ in } f \beta x' \]

The erasure of both \( g \) and \( g^{-1} \) is the application \( \lambda f. \lambda x. f x \), which is \( \eta \)-equivalent to the identity.

Question 15

We can only convert in one direction (namely, from left to right):

\[ g \overset{\text{def}}{=} \lambda f : \exists \alpha. (\tau \rightarrow \tau'). \lambda x : \exists \alpha. \tau. \text{let } x' = \beta = \text{unpack } f \text{ in } f \beta x' \]

Its type erasure is only \( \beta\eta \)-equivalent to the identity (as one linear let-binding must be reduced). However, the other direction cannot be defined.

Question 16

\[ \Gamma \vdash a : \Pi(\tau_1, \ldots, \tau_n) \quad i \in 1..n \]

\[ \Gamma \vdash \pi_i a : \tau_i \]

Question 16 (continued)

No, because we do not know how many fields the argument may contain, so we need an infinite collection of types:

\[ \forall \alpha_1 \ldots \alpha_n. \Pi(\alpha_1, \ldots, \alpha_i, \ldots, \alpha_n) \rightarrow \alpha_i \quad n \geq i \]

Question 17

We write "( )" for the unit value of type unit. We add a unit extra field to every tuple and hide the type of its last component.

\[ \llbracket \Pi(a_1, \ldots, a_n) \rrbracket \overset{\text{def}}{=} (a_1, (a_2, \ldots, (a_n, ()'))) : \tau_1 \times (\tau_2 \times \ldots \times (\tau_n \times \text{unit})) \]

\[ \llbracket \pi_a \rrbracket \overset{\text{def}}{=} \text{fst} \circ \text{snd}^i : \forall \alpha_1 \ldots \alpha_n. \beta. \alpha_1 \times (\alpha_2 \times \ldots \times (\alpha_n \times \beta)) \rightarrow \alpha_i \]

Question 18

We write \( \top \) for \( \exists \alpha. \alpha \). The unit value may always be considered of type \( \top \). Hence, the encoding of a tuple can also be given type \( \tau_1 \times (\tau_2 \times \ldots \times (\tau_n \times \top)) \). Moreover, the tail of the product can always be hidden by type-containment as follows:

\[
\begin{align*}
\text{INSTExi} \\
\prod_{\tau_p \times (\tau_{p+1} \times \ldots \times (\tau_n \times \top)) \leq \tau_p \times \top} \\
\prod_{\tau_1 \times \ldots (\tau_p \times (\tau_{p+1} \times \ldots (\tau_n \times \top)) \leq \tau_1 \times \ldots (\tau_p \times \top)}
\end{align*}
\]

Question 19

It is the smallest relation satisfying the following rules:

\[
\begin{align*}
\alpha & \leq \alpha \\
\bot & \leq \tau \\
\tau & \leq \top \\
\frac{\tau_1' \leq \tau_1}{\tau_1} & \leq \frac{\tau_2 \leq \tau_2'}{\tau_2} \\
& \frac{\tau_1 \leq \tau_1'}{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2}
\end{align*}
\]

(Adding the transitivity rule is correct, but it is unnecessary as transitivity is provable.)
Question 20
R-IOTA-PROD
\( (\langle \varphi_1 \times \varphi_2 \rangle)(V_1, V_2) \rightarrow (\langle \varphi_1 \rangle V_1, \langle \varphi_2 \rangle V_2) \)

R-IOTA-ALL
\( \langle \forall \alpha. \varphi \rangle(\Lambda \alpha. V) \rightarrow \Lambda \alpha. \langle \varphi \rangle V \)

R-IOTA-TRANS
\( \langle \varphi_1 ; \varphi_2 \rangle V \rightarrow \langle \varphi_1 \rangle \langle \varphi_2 \rangle V_2 \)

Question 21
Assume \( \Gamma \vdash M_0 : \tau_0 \) and \( M_0 \rightarrow M_0' \). We show that \( \Gamma \vdash M_0' : \tau_0 \) by induction on the proof of reduction.

Case R-IOTA-ARRROW: Then \( M_0 \) is \( \langle \varphi, \sigma \rightarrow \psi \rangle(\lambda \tau : x.M) \) and \( M_0' \) is \( \lambda y : \sigma. \langle \psi \rangle M[x \mapsto \langle \varphi \rangle y] \). and \( \tau_0 \) is \( \sigma \rightarrow \sigma' \). By inversion of typing, we have \( \Gamma, x : \tau \vdash M : \tau' \), \( \Gamma \vdash \varphi : \sigma \rightarrow \tau \) (2), and \( \Gamma \vdash \psi : \tau' \rightarrow \sigma' \). By weakening (which we assume extends to coercion judgments as well), we have \( \Gamma, x : \tau, y : \sigma \vdash M : \tau' \) (1), \( \Gamma, x : \tau, y : \sigma \vdash \varphi : \sigma \rightarrow \tau \) (2), and \( \Gamma, y : \sigma \vdash \psi : \tau' \rightarrow \sigma' \) (3). By \( \text{Var} \) applied to (1) then \( \text{Sub} \) applied with (3), we have \( \Gamma, x : \tau, y : \sigma \vdash \langle \psi \rangle M[x \mapsto \langle \varphi \rangle y] : \tau' \). Applying \( \text{Sub} \) with (3), we obtain \( \Gamma, y : \sigma \vdash \langle \psi \rangle M[x \mapsto \langle \varphi \rangle y] : \sigma' \). The conclusion follows by Rule Abs.

Question 22
Assume that \( \Gamma \vdash M : \tau \) where \( \Gamma \) is reduced to type variables and \( M \) is irreducible. We show that \( M \) is a value \( V \) by induction on \( M \).

Case \( M \) is \( \pi_i M_1 \): By inversion of typing, \( \Gamma \vdash M_1 : \tau_1 \times \tau_2 \) where \( \tau \) is \( \tau_i \). Since \( \pi_i [\ ] \) is an evaluation context, \( M_1 \) must be irreducible. By induction hypothesis, it must be a value. By classification, it must be a pair. Then \( M \) would reduce, so this case is not possible.

Case \( M \) is \( \langle \varphi \rangle M' \): By inversion \( \Gamma \vdash M' : \tau' \) for some type \( \tau' \). Since \( \langle \varphi \rangle E \) is an evaluation context, \( M' \) is irreducible as well. By induction, \( M' \) is a value \( V \), whose shape is determined by \( \tau' \). We show by induction on \( \varphi \) that \( V \) is such that some reduction rule applies to \( M \), which is a contradiction and implies that this case cannot occur.

Subcase \( \varphi \) is \( \Lambda \beta. \bar{\tau} \). Then \( \tau' \) is of the form \( \forall \alpha. \sigma \) and thus \( V \) is of the form \( \Lambda \alpha. V' \). \( M \) reduces by \( \text{R-IOTA-INST} \).

Subcase \( \varphi \) is \( \varphi_1, \sigma \rightarrow \varphi_2 \). Then \( \tau' \) is an arrow type and thus \( V \) is an abstraction. \( M \) reduces by \( \text{R-IOTA-ARRROW} \).

Subcase \( \varphi \) is \( \varphi_1 \times \varphi_2 \). Then \( \tau' \) is a product type and thus \( V \) is a pair. \( M \) reduces by \( \text{R-IOTA-PROD} \).

Subcase \( \varphi \) is \( \forall \alpha. \varphi' \). Then \( \tau' \) is a type abstraction and \( V \) is of the form \( \Lambda \alpha. V' \). \( M \) reduces by \( \text{R-IOTA-ALL} \).

Subcase \( \varphi \) is \( \varphi_1; \varphi_2 \). Then \( M \) unconditionally reduces by \( \text{R-IOTA-TRANS} \).

Question 23
\( \iota \)-reduction consumes a coercion \( \varphi \) and when it introduces a new coercion inside the term it is a strictly sub-coercion of \( \varphi \). Formally, we count the number of coercions appearing inside terms, including sub-coercions of coercions. Then, this number strictly decreases during reduction.

It is important that \( \iota \)-reduction terminates for the semantics to be type erasing.