Program transformations and record concatenation
Transformations de programmes et concatenation des enregistrements

Mid-term exam for MPRI 2-4 course / Examen partiel du cours MPRI 2-4
2016/12/02 — Duration / durée: 2h30

Answers to the different parts are independent of one another Questions are not ordered by their difficulty or the length of expected answers, but some questions towards the end of the last part are more difficult.

Answers will also be judged by their clarity and concision. You don’t have to justify answers when not asked.

Notice that math displays are one column and spread other the whole page width.

1 Type preserving program transformations / Transformations de programmes préservant le typage

We now consider the implicitly typed presentation of System F. As in the course, we write $a$ for implicitly typed terms, and if need be $M$ for explicitly typed terms. Moreover, we assume that the language has a primitive let-binding construct $\text{let } x = a_1 \text{ in } a_2$. We admit that all results proved in the course still hold in the presence of let-bindings.

An important characteristic for a programming language is the set of program transformations that preserve typings. We recall that a typing of an expression $a$ is a pair $\Gamma, \tau$ such that $\Gamma \vdash a : \tau$ holds. We write $a_1 \subseteq a_2$ when all typings of $a_1$ are also typings of $a_2$.

Question 1

For example, the following two properties hold:

\textbf{Beta-Let-Value} \hfill \textbf{Beta-Value}
\begin{align*}
\text{let } x = v \text{ in } a & \subseteq a[x \mapsto v] \\
(\lambda x. a) \; v & \subseteq a[x \mapsto v]
\end{align*}

a) Justify this, using results that have been proved in the course.

Par exemple, nous avons les deux propriétés suivantes :

\textbf{Beta-Let-Value} \hfill \textbf{Beta-Value}
\begin{align*}
\text{let } x = v \text{ in } a & \subseteq a[x \mapsto v] \\
(\lambda x. a) \; v & \subseteq a[x \mapsto v]
\end{align*}

a) Justifiez-le en utilisant des résultats démontrés dans le cours.
b) Do we also have the following two properties? Justify, intuitively, without proving it.

\[ \text{FULL-BETA-LET} \]
\[
\text{let } x = a_0 \text{ in } a \subseteq a \ni \lambda x. a_0 \]

**Question 2**

Consider the converse inclusions:

\[ \text{INV-BETA-LET} \]
\[
a \ni \lambda x. a_0 \subseteq \text{let } x = a_0 \text{ in } a \]

a) Give an example where both inclusions are trivially false. b) What is the situation when we restrict to cases where \( x \) appears free in \( a \)?

**Question 3**

Consider the following two inclusions with \( x \) not appearing free in \( a \):

\[ \text{ETTA} \]
\[
\lambda x. a x \ni a 
\]

\[ \text{INV-ETTA} \]
\[
a \ni \lambda x. a x 
\]

Is ETTA true? (If not, justify, briefly.)

Is Rule INV-ETTA true?

**Question 4**

For reasoning, we also need the following rule where \( C \) is an arbitrary context, which does hold.

\[ a_1 \ni a_2 \implies C[a_1] \ni C[a_2] \]

**Context**

a) Name a similar property in the course? b) What cannot we reuse it, directly?

Another interesting property of a language is its ability to abstract over primitive constructs of the language. For example, can we replace the application function \( a_1 a_2 \) by a user-defined version of the application \( \lambda x. \lambda y. x y \)? (If we allowed side effects, this could for instance be used for logging or statistics purposes.) That is, we wish to find a type \( \sigma \) of the function \( \lambda x. \lambda y. x y \) so that if \( \text{app} \) is a constant of type \( \sigma \), then we have (for any \( a_1 \) and \( a_2 \)):

\[ a_1 \ni a_2 \ni \text{app } a_1 a_2 \]

b) A-t-on également les deux propriétés suivantes? Justifiez, intuitivement, sans le prouver.
Question 5

a) Give the principal type of $\lambda x. \lambda y. x\ y$ in ML.

b) Prove that taking the same type for $\sigma$, but in $F$, satisfies the property.

Question 6

The previous property shows that application can be abstracted away. What would be another good test to show that typechecking applications is not biased towards the left hand side or right hand side?

Question 7

We now replace $F$ by $ML$. Among the typing inclusions $Inv$-$Beta$-$Let$, $Inv$-$Beta$, $Eta$, $Inv$-$Beta$ which ones need to be modified, i.e. were true in $F$ and are false in $ML$, or conversely? Justify briefly.

2 The option type / Le type option

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

$$M ::= x \mid c \mid \lambda x. \tau. M \mid M \ M \mid \Lambda \alpha. M \mid M \ \tau$$

where the dots are type constants, such as int, list $\tau$, etc. Typing rules are given in Figure 1.

We wish to add an option type equivalent to the following OCaml type definition:

```ocaml
type 'a opt = S of 'a | N
```
Question 8

Give a formal definition of this extension in System F (syntactic changes, typechecking, and semantics), reusing if possible the formalism in the course.

Donner une définition formelle de cette extension dans le système F (modifications syntaxiques, typage et sémantique), en réutilisant si possible le formalisme du cours.

Question 9

Show that this extension preserves the soundness of the language. (You may reuse all results from the course.)

Montrez que cette extension préserve la sûreté du langage (Vous pouvez réutiliser tous les résultats du cours.)

3 Another option / Une autre option

We now consider the implicitly typed version of ML with pairs and an option type as defined in the previous part.

Nous considérons maintenant une version implicitement typée de ML avec des paires et un type somme comme celui défini dans la partie précédente.

Question 10

We may represent two-field records as pairs, and use type opt \( \tau \) to represent fields that may be undefined. For example, we may define the following records:

\[
\begin{align*}
r_0 & \overset{\text{def}}{=} (N, N) \\
r_1 & \overset{\text{def}}{=} (S \, 1, N) \\
r_2 & \overset{\text{def}}{=} (S \, \text{true}, S \, 2)
\end{align*}
\]

a) Give their types in ML.

Nous écrivons \( w \) pour les valeurs de la forme \( N \) ou \( S \, v \) appelées valeurs de champs.

b) We may reread the expressions \( r_0, r_1, \) and \( r_2 \) with this new definition. Give their new type schemes.

b) Nous pouvons relire les expressions \( r_0, r_1, r_2 \) avec cette nouvelle définition. Donner leurs nouveaux schémas de types.

c) What is the main difference between the two approaches?

c) Quel est la principale différence entre les deux approches?

Question 11

Give the code and principal type scheme of a function getfst that takes a record and returns the (content of the) first field, assuming that it is definitely there.

Donner le code et les types principaux de la fonction getfst qui prend un record et retourne le (contenu du) premier champ, en supposant qu’il est bien présent.

To which of the records \( r_0, r_1, \) and \( r_2 \) above will the application of getfst will be well-typed?

Auxquels des trois records \( r_0, r_1, r_2 \), l’application de getfst sera-t-elle bien typée?
Question 12

Give the code and the type scheme of a function \texttt{addfst} that takes a record \( r \) and a value \( x \) and returns a record that is equal to \( r \) except on the first field where its content is now \( x \).

Donner le code et le type principal de la fonction \texttt{addfst} qui prend un enregistrement \( r \) et une valeur \( x \) et retourne un enregistrement qui est égal à \( r \) sauf sur le premier champ où son contenu est maintenant \( x \).

Notice that all records may be constructed from \( r_0 \) with \texttt{addfst} and \texttt{addsnd} and destructed with \texttt{getfst} and \texttt{getsnd}. In particular, we could provide an interface with these five constants together with their type scheme but leaving the types \( s \tau \), \( n \), and the type of pairs abstract.

Question 13

What would be the advantage of giving such an interface?

Quel serait l’avantage de donner une telle interface ?

Question 14

We would like to define an operation \texttt{append} that takes two records \( r_1 \) and \( r_2 \) and return the record with field is taken from \( r_2 \) when defined, or from \( r_1 \) otherwise. It behaves like

\[
\texttt{append} \overset{\text{def}}{=} \lambda x_1. \lambda x_2. (\text{fst} x_1 \@ \text{fst} x_2, \text{snd} x_1 \@ \text{snd} x_2)
\]

where / où

\[
. \@_\text{def} = \lambda x. \lambda y. \textbf{match} \ y \textbf{ with } N \rightarrow x \mid S \ z \rightarrow y
\]

\( \text{a) Explain very briefly why the infix operator \@ is ill-typed.} \)

\( \text{a) Expliquer très brièvement pourquoi l’opérateur \@ est mal typé.} \)

\( \text{We instead treat \@ as a primitive with the following \( \delta \)-reduction.} \)

\( \text{À la place, nous traitons \@ comme une primitive. avec la \( \delta \)-réduction suivante :} \)

\[
w \@ N \rightarrow w
\]

\( \text{b) Give two possible type schemes for \@.} \)

\( \text{b) Donner deux schémas de type possible pour \@.} \)

\( \text{c) Does it have a principal type scheme? (if yes, give it)} \)

\( \text{c) Admet-il un type principal (si oui, le donner) ?} \)

Question 15

Let us write \( \@_a \) the partial application \( \lambda x. (x \@ a) \).

\( \text{a) Give possible types for \( \@_N \) and \( \@_Sv \) for some value \( v \) of type \( \tau \).} \)

\( \text{a) Donner des types possibles \( \@_N \) et \( \@_Sv \) pour une valeur \( v \) de type \( \tau \).} \)

\( \text{b) Give a function \texttt{get} such that for any field value \( w \) the application \texttt{get} \( \@_w \) reduces to \( w \). (It can be used to recover the value \( w \) of a field from its partial application \( \@_w \).)} \)

\( \text{b) Donner une fonction \texttt{get} tel que à partir de la valeur \( w \) d’un champ, l’application \texttt{get} \( \@_w \) se réduit en \( w \). (Elle peut être utilisée pour retrouver la valeur \( w \) d’un champ à partir de son application partielle \( \@_w \).)} \)
Question 16

We write \( \circ \) for the composition \( \lambda f. \lambda g. \lambda z. g (f z) \). Verify that \( @_{w_2} \circ @_{w_1} \) simulates \( @_{w_1} @_{w_2} \), that is, both expressions reduce to the same value when applied to the same field value \( w \).

We conclude by associativity of \( @ \), which we check by cases on \( w_2 \) then \( w_1 \).

The above observation suggests that \( @_w \) is a better encoding of field \( w \) than \( w \) itself since it allows to recover \( w \) while enabling type checking of concatenation.

We define an encoding \( \cdot \) that transforms all operations on records, \( \text{i.e.} \ r_0, \text{getfst, getsnd, addfst, and addsnd} \).

A field value \( w \) is encoded as \( @_w \) and the encoding of a record value is the record of its field encodings. We write \( \triangleright \) instead of \( \to \) in the types of encoded fields. For example, we now have:

\[
\begin{align*}
\text{r}_1^\dagger & \overset{\text{df}}{=} \text{addfst} \; r_0 \; 1 \quad \xrightarrow{\cdot} \quad \left( (\lambda y. \; S \; 1), (\lambda y. \; y) \right) : \forall \varphi. \forall \beta. (\beta \triangleright \text{int}) \times (\varphi \triangleright \varphi)
\end{align*}
\]

Question 17

Give the new definition and the type schemes of the constants \( r_0^\dagger, \text{addfst}^\dagger, \text{getfst}^\dagger \) (you may omit quantifiers, all free type variables being understood as universally quantified).

Donner la nouvelle définition et les schémas de types des constantes \( r_0^\dagger, \text{addfst}^\dagger, \text{getfst}^\dagger \) (vous pouvez omettre les quantificateurs, toutes les variables libres étant comprises comme universellement quantifiées).

Question 18

Define \( \text{append}^\dagger \) that appends two records in the new encoding.

Définir \( \text{append}^\dagger \) qui concatène deux records dans le nouvel encodage.

Question 19

a) Give the principal type schemes of the well-typed expressions among \( (\text{addfst} \; r_0 \; 1)^\dagger \) and \( (\text{append} \; r_1 \; r_2)^\dagger \) and \( (\text{append} \; r_2 \; r_1)^\dagger \).

b) For each of these expressions, say \( a \), is the type of the result of the evaluation of \( a \) more general than the type of \( a \)? (no justification is needed.)

a) Donner les schémas de types principaux des expressions bien typées parmi \( (\text{addfst} \; r_0 \; 1)^\dagger \) et \( (\text{append} \; r_1 \; r_2)^\dagger \) et \( (\text{append} \; r_2 \; r_1)^\dagger \).

b) Pour chacune de ces expressions, disons \( a \), est-ce que le type du résultat de l’évaluation de \( a \) est plus général que le type de \( a \)? (aucune justification n’est demandée.)

Question 20

a) Give the principal type of \( \lambda x. \text{append}^\dagger \; x \; x \).

a) Donner le type principal de \( \lambda x. \text{append}^\dagger \; x \; x \).
b) To which values among $r_0$, $r_1$, $r_2$ is its application well-typed? Explain why this is unsatisfactory.

b) À quelle valeurs parmi $r_0$, $r_1$, $r_2$ son application est-elle bien typée? Explain why this is unsatisfactory.

Question 21

The new encoding has been proposed to solve a typechecking issue, but we would like to keep the old, more direct and more efficient definitions, while reusing the types of the new encoding. More precisely, we propose to just keep the constants $r_0$, `getfst`, `addfst`, `append` with their implementation, but give them the new types.

However, the type system is not quite sound, because we have typechecked fields as functions, while they remain of the form $Sv$ or $N$.

a) Give an example that does not follow this convention and breaks type soundness.

b) How could we make it safe?
4 Solutions

Question 1
a) Both rules are a consequence of subject reduction, which holds for both explicitly typed terms an implicitly typed terms.

Question 1 (continued)
b) Yes, they both hold, because typings are also preserved by call-by-name or even full reduction, but this has not been proved in the course.

Question 2
a) Take an ill-typed value for \( a_0 \).

Note: \( \lambda x. x \ x \) is not an ill-typed term in System F!

b) They both remain false.

Question 3
\text{Eta}
is false. Otherwise, combined with Context shown below, \( F \) would be closed by \( \eta \)-reduction, and therefore equal to \( F_\eta \). (Eta is correct in System \( F_\eta \).)

Question 3 (continued)
\text{Inv-Eta}
is false: take any typing \( (\Gamma, \tau) \) where \( \tau \) is not an arrow type. (Still, all typings of the form \( \Gamma, \tau_1 \rightarrow \tau_2 \) are preserved by \( \eta \)-expansion.)

Question 4
a) This is compositionality of typing.

b) This has been stated (and proved) in the course, but only for evaluation contexts and for explicitly typed terms. (In particular, there is no counter-part to when \( C \) is \( \lambda x. [ ] \).)

Question 5
a) \( \forall \alpha \forall \beta. (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \).

Question 5 (continued)
b) We assume \( \Gamma \vdash a_1 \ a_2 : \tau \) (1) and show \( \Gamma \vdash \text{app} \ a_1 \ a_2 : \tau \) (2) by induction on the derivation of (1) then by inversion.

Subcase \text{App}. We have \( \Gamma \vdash a_1 : \tau_1 \rightarrow \tau \) and \( \Gamma \vdash a_2 : \tau_1 \). By Const and Tapp, we have \( \Gamma \vdash \text{app} : (\tau_1 \rightarrow \tau) \rightarrow \tau_1 \rightarrow \tau' \). The conclusion (2) follows by two successive applications of Rule App.

Subcase \text{Tabs}. We have \( \Gamma, \alpha \vdash a_1 \ a_2 : \tau_0 \) (3) with \( \tau \) equal to \( \forall \alpha. \tau_0 \). Applying the induction hypothesis to (3), we get \( \Gamma, \alpha \vdash \text{app} \ a_1 \ a_2 : \tau_0 \). We conclude by rule Tabs.

Subcase \text{Tapp}. We have \( \Gamma \vdash a_1 \ a_2 : \forall \alpha. \tau_0 \) (4) where \( \tau \) is \( \tau_0[\alpha \mapsto \sigma] \). Applying the induction hypothesis to (3), we get \( \Gamma, \alpha \vdash \text{app} \ a_1 \ a_2 : \forall \alpha. \tau_0 \). We conclude by rule Tapp.
Question 6
Introducing a constant \texttt{revapp} of type \(\forall \alpha_1. \forall \alpha_2. \alpha_1 \rightarrow (\alpha_1 \rightarrow \alpha_2) \rightarrow \alpha_2\) which is also the type of the function \(\lambda x. \lambda y. y x\) satisfies the following property:

\[ a_1 a_2 \subseteq \text{revapp} a_2 a_1 \]

Hence, the order of arguments does not influence well-typedness.

Question 7
All inclusions are preserved. Moreover, two new inclusions become true in ML

- Rule Inv-Beta-Let, since ML is the closure of the simply-typed \(\lambda\)-calculus by let-expansion, which is related to the fact that ML admits principal types (while F does not).
- Rule Inv-Eta, since type abstraction can only be assigned a simple type in ML.

Question 8
We introduce a new unary type symbol \texttt{opt}, a nullary constructor \texttt{N}, a unary constructor \texttt{S}, and a ternary destructor \texttt{matchopt} with the following type schemes (declared in \(\Delta\)):

\[
\begin{align*}
N & : \forall \alpha. \texttt{opt} \alpha \\
S & : \forall \alpha. \alpha \rightarrow \texttt{opt} \alpha \\
\text{matchopt} & : \forall \alpha. \forall \beta. \texttt{opt} \alpha \rightarrow \beta \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta
\end{align*}
\]

We add two \(\delta\)-reduction rules:

\[
\begin{align*}
\text{matchopt } \tau' \tau (N \tau'' v_N v_S) v & \rightarrow v_N \\
\text{matchopt } \tau' \tau (S \tau'' v) v_N v_S & \rightarrow v_S v
\end{align*}
\]

Note: Taking \(\tau''\) equal to \(\tau'\) would also be correct, as long as the corresponding proof obligation is checked in progress.

Question 9
We need to check subject-reduction and progress for constants. We first consider subject reduction.

Subcase Match-S. We assume \(\Gamma \vdash \text{matchopt } \tau' \tau (\text{S } \tau'' v) v_N v_S : \sigma (5)\). We show \(\Gamma \vdash v_S v : \sigma (6)\).

By inversion of typing applied to (5), we obtain \(\Gamma \vdash v_S : \tau' \rightarrow \tau (7)\), \(\Gamma \vdash v : \tau'' (8)\) and the equalities \(\tau' = \tau''\) and \(\tau = \sigma\). We obtain the conclusion (6) by rule \texttt{App} applied to (7) and (8) using the equalities.

Subcase Match-N. This case is similar but simpler (the details could have been omitted). We assume \(\Gamma \vdash \text{matchopt } \tau' \tau (N \tau'' v) v_N v_S : \sigma (9)\). We show \(\Gamma \vdash v_N : \sigma (10)\).

By inversion of typing applied to (9), we obtain \(\Gamma \vdash v_N : \tau\) and the equality \(\tau = \sigma\), which imply the conclusion (10).

We now prove progress. We have only the full application \texttt{matchopt } \tau' \tau v v_N v_S, say \(M\), to consider. Assume that \(\emptyset \vdash v : \sigma\). By inversion of typing, we know that \(v\) has type \texttt{opt } \tau'. By classification, \(v_0\) must be either \texttt{N } \tau'' or \texttt{S } \tau'' v'. Reduction of \(M\) may proceed by rule \texttt{Match-N} in the former case and by \texttt{Rule MATCH-S} in the latter case.

Question 10

a) \[
\begin{align*}
r_0 : \forall \alpha_1. \forall \alpha_2. \texttt{opt } \alpha_1 \times \texttt{opt } \alpha_2 \\
r_1 : \forall \alpha_2. \texttt{opt int } \times \texttt{opt } \alpha_2 \\
r_2 : \texttt{opt bool } \times \texttt{opt int}
\end{align*}
\]

b) \[
\begin{align*}
r_0 : n \times n \\
r_1 : \texttt{s int } \times n \\
r_2 : \texttt{s bool } \times \texttt{s int}
\end{align*}
\]
Question 10 (continued)
c) Typechecking tracks undefined fields in the second approach, but not in the first one. As a result it makes records with different sets of undefined fields incompatible in the second approach. (It tracks types of defined fields similarly in both approaches.)

Question 11
\[
\text{getfst} : \forall \alpha_1, \forall \varphi_2, s \alpha_1 \times \varphi_2 \rightarrow \alpha_1 \overset{\text{def}}{=} \lambda r. \text{match } \text{fst } r \text{ with } S \ x \rightarrow x
\]

Question 11 (continued)
The records \(r_1\) and \(r_2\), as expected.

Question 12
\[
\text{addfst} : \alpha_1 \times \alpha_2 \rightarrow \alpha_0 \rightarrow s \alpha_0 \times \alpha_2 \overset{\text{def}}{=} \lambda r. \lambda x. (S \ x; \text{snd } r)
\]

Question 13
It would hide the encoding and prevent mixing pairs representing records with other uses of pairs—and similarly for values representing fields.

Question 14
a) The patterns \(N\) and \(S \ z\) have types \(n\) and \(s \ \tau\) which are incompatible while they should both be the type of \(y\).

Question 14 (continued)
b) \[
\text{@N} : \forall \varphi. \varphi \rightarrow n \rightarrow \alpha \\
\text{@S} : \forall \varphi. \forall \alpha. \varphi \rightarrow (s \ \alpha) \rightarrow s \ \alpha \\
c) It does not have a principal type scheme.

Question 15
a) \[
\text{@N} : \forall \varphi. \varphi \rightarrow \varphi \\
\text{@S} : \forall \varphi. \varphi \rightarrow S \ \tau
\]

Question 15 (continued)
b) Take the expression \(\lambda x. x \ N\) for get.

Question 15 (continued)
\[
\text{@}_{w_1} \circ \text{@}_{w_2} w \rightarrow w \ @ (w_1 \ @ w_2) \\
(\text{@}_{w_2} \circ \text{@}_{w_1}) w \rightarrow \text{@}_{w_2} (\text{@}_{w_1} w) = (w \ @ w_1) \ @ w_2
\]
Question 17

\[ r_0^\dagger = ((\lambda y. y), (\lambda y. y)) : (\varphi_1 \triangleright \varphi_1) \times (\varphi_2 \triangleright \varphi_2) \]

\[ \text{addfst}^\dagger = \lambda r. \lambda x. ((\lambda y. S x), \text{snd } r) : \theta_1 \times \theta_2 \rightarrow \alpha \rightarrow (\varphi \triangleright s \alpha) \times \theta_2 \]

\[ \text{getfst}^\dagger = \lambda r. \text{match (fst } r \text{) N with S } y \rightarrow y : (n \triangleright s \alpha) \times \theta_2 \rightarrow \alpha \]

Note: Taking \( \varphi_1 \triangleright \varphi_1^\prime \) for \( \theta_i \) would also be an acceptable answer.

Question 18

\[ \text{append}^\dagger = \lambda r_1. \lambda r_2. \text{((fst } r_2 \text{) o (fst } r_1 \text{), (snd } r_2 \text{) o (snd } r_1 \text{))} \]

\[ : (\varphi_1 \triangleright \varphi'_1) \times (\varphi_2 \triangleright \varphi'_2) \rightarrow (\varphi'_1 \triangleright \varphi''_1) \times (\varphi'_2 \triangleright \varphi''_2) \rightarrow (\varphi_1 \triangleright \varphi_1^\prime) \times (\varphi_2 \triangleright \varphi_2^\prime) \]

Question 19

a)

\[ (\text{addfst } r_0 \ 1)^\dagger : (\beta \triangleright s \text{ int}) \times (\varphi \triangleright \varphi) \]

\[ (\text{append } r_1 \ 1)^\dagger : (\beta_1 \triangleright s \text{ bool}) \times (\beta_2 \triangleright s \text{ int}) \]

\[ (\text{append } r_2 \ 1)^\dagger : (\beta_1 \triangleright s \text{ int}) \times (\beta_2 \triangleright s \text{ int}) \]

Question 19 (continued)

b) No, they coincide, in all three cases.

Question 20

a)

\[ \forall \varphi_1. \forall \varphi_2. (\varphi_1 \triangleright \varphi_1) \times (\varphi_2 \triangleright \varphi_2) \rightarrow (\varphi_1 \triangleright \varphi_1) \times (\varphi_2 \triangleright \varphi_2) \]

Question 20 (continued)

b) The application to all three arguments is well-typed.

However, the result of these applications of \( r_1 \) and \( r_2 \), say \( r'_1 \) and \( r'_2 \), have less general types than \( r_1 \) and \( r_2 \), since all fields have types of the form \( \varphi \rightarrow \varphi \), i.e. as if they were empty fields. In particular, \( \text{getfst} \) can be applied to neither \( r'_1 \) nor \( r'_2 \). In fact, \( \lambda x. \text{getfst } (\text{append}^\dagger \ x \ x) \) is ill-typed.

Question 21

a) \( \text{fst } r_1 \) \text{ true} is well-typed while it reduces to \( S \ 1 \text{ true} \) which is stuck!

Question 21 (continued)

b) Make the encoding abstract, for example using an interface (as in OCaml modules) that makes \( \triangleright \) an abstract type constructor (so that it is no more an alias for the arrow), and only export \( r_0 \) and \( \text{addfst} \) and \( \text{addsnd} \) to create records; \( \text{getfst}, \text{getsnd}, \text{addfst}, \text{addsnd} \) and \( \text{append} \) to manipulate them.

Less importantly, we should also distinguish field type variables from regular type variables, using kinds.