A simple object encoding / Un codage simple des objets

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

2018/11/30 — Duration / durée: 3h00

Sections are not ordered by increasing level of difficulty. Parts 2–3 and 5–7 are largely independent of one another.

Answers are judged by their correctness, but also by their clarity, conciseness, and precision. You don't have to justify answers when not asked.

Math displays are one-column and spread over the whole page width.

Les sections ne sont pas ordonnées par niveau de difficulté croissant. Les parties 2-3 et 5-7 sont largement indépendantes l'une de l'autre.

Les réponses sont jugées d'après leur correction, mais aussi d'après leur clarté, leur concision et leur précision. Vous n'avez pas besoin de justifier vos réponses lorsque ce n'est pas demandé.

Les formules mathématiques sont en simple colonne et s'étendent sur toute la largeur de la page.

Erratum.

Le début de l'énoncé contenait au moins une erreur. Dans le but de simplifier la grammaire des contextes d'évaluation E, nous avons restreint la grammaire du calcul et imposé, dans de nombreuses constructions, que les sous-expressions soient des valeurs: appel de fonction $v\ v$, appel de méthode $v\#\ell$, addition $v + v$, etc. Ce langage restreint est bien stable par substitution d'une valeur pour une variable. Toutefois, par erreur, les variables x ont été rangées dans la catégorie des expressions, alors qu'elles auraient dû apparaître dans la catégorie des valeurs. (Cette erreur est réparée dans l'énoncé qui suit.) Cela a pu empêcher certains de répondre correctement à la première question, où il était nécessaire qu'une variable apparaisse en position de valeur. Nous n'avons donc pas tenu compte de cette question.

Par ailleurs, le fait que $e_1\ e_2$ soit un sucre si e_1 ou e_2 n'est pas une valeur, mais une construction primitive si e_1 et e_2 sont des valeurs, a pu être gênant lorsqu'il s'agissait d'écrire précisément une séquence de réductions. Nous avons noté de façon libérale.

We are interested in a calculus named OBJLAM, which can be presented as a λ-calculus extended with several primitive concepts: integers, records, and objects. The syntax of this calculus is as follows:

On s'intéresse à un calcul nommé OBJLAM, que l'on peut présenter comme un λ-calcul auquel on a ajouté plusieurs notions primitives : entiers, enregistrements, et objets. La syntaxe de ce calcul est la suivante :

values	v	$::=$	x	variable
			$\lambda x.e$	function
			$\widehat{k}$	literal integer
			$\{\ell_i = v_i\}^{i\in I}$	literal record
			$\langle \ell_i = \zeta x_i.b_i\rangle^{i\in I}$	literal object
expressions	e, b	$::=$	v	value
			$v\ v$	function application
			$\mathsf{let}\ x = e\ \mathsf{in}\ e$	local variable definition / sequencing
			$v + v$	integer addition
			$v.\ell$	field access
			$v\{\ell \leftarrow v\}$	field update
			$v\#\ell$	method call
			$v\langle \ell \leftarrow \zeta x.b\rangle$	method override
evaluation contexts	E	$::=$	$\mathsf{let}\ x = [\,]\ \mathsf{in}\ e$	

A record $\{\ell_i = v_i\}^{i\in I}$ consists of a set of fields, where each field carries a label ℓ_i and contains a value v_i. The construct $v.\ell$ allows accessing a field. The construct $v\{\ell \leftarrow v'\}$ allows updating a field. Records are not mutable: this construct yields a new record, which is distinct from the original record.

Un enregistrement $\{\ell_i = v_i\}^{i\in I}$ est constitué d'un ensemble de champs, où chaque champ est étiqueté ℓ_i et contient une valeur v_i. La construction $v.\ell$ permet d'accéder à la valeur d'un champ. La construction $v\{\ell \leftarrow v'\}$ permet de mettre à jour un champ. Un enregistrement n'est pas mutable : cette construction produit un nouvel enregistrement, distinct de l'original.

Similarly, an object $\langle \ell_i = \zeta x_i.b_i\rangle^{i\in I}$ consists of a set of methods $\zeta x_i.b_i$. In the body b_i of a method, the variable x_i refers to the object itself: it is sometimes said that this variable represents *self* or *this*. The construct $v\#\ell$ is a method call. The construct $v\langle \ell \leftarrow \zeta x.b\rangle$ is a method override: it produces a new object, which is distinct from the original object.

De façon similaire, un objet $\langle \ell_i = \zeta x_i.b_i\rangle^{i\in I}$ est constitué d'une collection de méthodes $\zeta x_i.b_i$. Dans le corps b_i d'une méthode, la variable x_i fait référence à l'objet lui-même : on dit parfois que cette variable représente *self* ou *this*. La construction $v\#\ell$ est un appel de méthode. La construction $v\langle \ell \leftarrow \zeta x.b\rangle$ permet de redéfinir une méthode : elle produit un nouvel objet, distinct de l'original.

The small-step operational semantics of the calculus OBJLAM is as follows:

La sémantique opérationnelle à petits pas du calcul OBJLAM est la suivante :

Function Call
$$(\lambda x.e)\ v \longrightarrow e[v/x]$$

Sequencing
$$\mathsf{let}\ x = v\ \mathsf{in}\ e \longrightarrow e[v/x]$$

Addition
$$\widehat{k_1} + \widehat{k_2} \longrightarrow \widehat{k_1 + k_2}$$

Record Access
$$\frac{v = \{\ell_i = v_i\}^{i\in I}}{v.\ell_j \longrightarrow v_j}$$

Record Update
$$\frac{v = \{\ell_i = v_i\}^{i\in I}}{v\{\ell_j \leftarrow v'\} \longrightarrow \{\ell_j = v'; \ell_i = v_i{}^{i\in I\setminus\{j\}}\}}$$

Method Call
$$\frac{v = \langle \ell_i = \zeta x_i.b_i\rangle^{i\in I}}{v\#\ell_j \longrightarrow b_j[v/x_j]}$$

Method Override
$$\frac{v = \langle \ell_i = \zeta x_i.b_i\rangle^{i\in I}}{v\langle \ell_j \leftarrow \zeta x_j.b'_j\rangle \longrightarrow \langle \ell_j = \zeta x_j.b'_j; \ell_i = \zeta x_i.b_i{}^{i\in I\setminus\{j\}}\rangle}$$

Context
$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

The rules Record Access, Record Update, Method Call and Method Override implicitly require $\ell_j \in \{\ell_i\}^{i \in I}$, which means that the field or method labeled ℓ_j must exist already. In particular, an update or override operation cannot create a new field or method.

Les règles Record Access, Record Update, Method Call et Method Override exigent implicitement $\ell_j \in \{\ell_i\}^{i \in I}$, c'est-à-dire que le champ ou la méthode étiquetée ℓ_j doit exister déjà. En particulier, une opération de mise à jour de champ ou de redéfinition de méthode ne peut pas créer un nouveau champ ou une nouvelle méthode.

The calculus OBJ is the subset of OBJLAM whose syntax is restricted to variables, sequencing, objects, method calls, and method override.

Le calcul OBJ est le sous-ensemble d'OBJLAM dont la syntaxe est restreinte aux variables, la séquence, les objets, les appels de méthode, et la redéfinition de méthode.

1 Warmup / Échauffement

You may have noticed that the application of an expression to an expression, $e_1\ e_2$, is not a valid expression. Indeed, a valid function application must be of the form $v_1\ v_2$. This decision allows us to simplify the syntax of evaluation contexts.

Vous aurez peut-être remarqué que l'application d'une expression à une expression, $e_1\ e_2$, n'est pas une expression valide. En effet, une application de fonction doit être de la forme $v_1\ v_2$. Cette décision nous permet d'alléger la grammaire des contextes d'évaluation.

Question 1

In spite of this restriction, one might still wish to write $e_1\ e_2$ as sugar for a valid expression. Propose a sensible definition for this notation. Similarly, propose a definition for the notation $e\#\ell$.

Malgré cette restriction, on peut souhaiter pouvoir écrire $e_1\ e_2$ et le considérer comme un sucre pour une expression valide. Proposez une définition raisonnable de cette notation. De manière similaire, proposez une définition pour la notation $e\#\ell$.

□

The sugar proposed above will be used several times in the following.

Le sucre proposé ci-dessus sera utilisé plusieurs fois dans la suite.

For the sake of brevity, in preparation for the next question, we introduce a shorthand, which is used in the question, and which you should use also in your answer. If k is an integer, we write $P[k]$ for the following object:

Par souci de concision, en vue de la prochaine question, on introduit une abréviation, qui est utilisée dans la question et que vous utiliserez également dans votre réponse. Pour un entier k quelconque, on note $P[k]$ l'objet suivant :

$$\left\langle \begin{array}{l} \mathsf{get} = \varsigma \mathit{self}.\widehat{k} \\ \mathsf{set} = \varsigma \mathit{self}.\lambda x.\mathit{self}\langle \mathsf{get} \leftarrow \varsigma \mathit{self}.x\rangle \\ \mathsf{inc} = \varsigma \mathit{self}.\mathit{self}\#\mathsf{set}\ (\mathit{self}\#\mathsf{get} + \widehat{1}) \end{array} \right\rangle$$

This object intuitively represents a point on a 1-dimensional line. Its coordinate, which is returned by its get method, is k. It is worth remarking that the sugar introduced in Question 1 is used in the definition of $P[k]$.

Cet objet représente intuitivement un point sur une ligne unidimensionnelle. Sa coordonnée, qui est renvoyée par sa méthode get, est k. On notera que le sucre introduit lors de la Question 1 est employé dans la définition de $P[k]$.

Question 2

What is the use of the set and inc methods, and what do they return? A brief and informal answer is expected.

À quoi servent les méthodes set et inc, et que renvoient-elles ? On attend une réponse brève et informelle.

□

Question 3

What maximal reduction sequence arises out of the term $P[k]\#\mathsf{get}$? (The sugar $e_1\ e_2$ or $e\#\ell$ is used in some places in this example; it must be expanded as needed.)

Quelle séquence de réductions maximale est issue du terme $P[k]\#\mathsf{get}$? (Le sucre $e_1\ e_2$ ou $e\#\ell$ est utilisé en quelques endroits dans cet exemple ; il doit être expansé au besoin.)

□

Question 4

What maximal reduction sequence arises out of the following term? (The sugar $e_1\ e_2$ or $e\#\ell$ is used in some places in this example; it must be expanded as needed.)

Quelle séquence de réductions maximale est issue du terme suivant ? (Le sucre $e_1\ e_2$ ou $e\#\ell$ est utilisé en quelques endroits dans cet exemple ; il doit être expansé au besoin.)

$$\mathsf{let}\ point = P[0]\ \mathsf{in}\ point\#\mathsf{inc}\#\mathsf{get}$$

□

2 Codage des objets / Encoding objects

We define a transformation of OBJ into (a subset of) OBJLAM which eliminates objects by encoding them in terms of records and functions. The transformation is as follows:

On définit une traduction de OBJ vers (un sous-ensemble de) OBJLAM qui fait disparaître les objets en les codant à l'aide d'enregistrements et de fonctions. La traduction est la suivante :

$$\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\langle \ell_i = \zeta x_i.b_i\rangle^{i\in I}]\!] & = & \{\ell_i = \lambda x_i.[\![b_i]\!]\}^{i\in I} \\
[\![v\#\ell]\!] & = & \mathsf{let}\ x = [\![v]\!]\ \mathsf{in}\ x.\ell\ x \\
[\![v\langle \ell \leftarrow \zeta x.b\rangle]\!] & = & [\![v]\!]\{\ell \leftarrow \lambda x.[\![b]\!]\} \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] & = & \mathsf{let}\ x = [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!]
\end{array}$$

Question 5

Prove that this transformation preserves reduction: $e \longrightarrow e'$ implies $[\![e]\!] \longrightarrow^+ [\![e']\!]$. Please announce the general structure of the proof, then deal explicitly only with the proof cases of Method Call and Method Override. If an auxiliary lemma is needed, please state this lemma and use it, without proving it.

Montrez que cette traduction préserve la réduction : $e \longrightarrow e'$ implique $[\![e]\!] \longrightarrow^+ [\![e']\!]$. Vous annoncerez la structure générale de la preuve mais ne traiterez explicitement que les cas Method Call et Method Override. Si un lemme auxiliaire est nécessaire, énoncez ce lemme et utilisez-le sans le démontrer.

□

Question 6

Prove that this transformation is semantics-preserving. Please indicate first which statement(s) must be shown, then do so. A brief answer is expected.

Montrez que cette traduction préserve la sémantique. Indiquez d'abord précisément quel(s) énoncé(s) il faut vérifier, puis faites-le. On attend une réponse brève.

□

3 Codage des fonctions / Encoding functions

We define a transformation of the pure λ-calculus into OBJ, which eliminates functions by encoding them in terms of objects. The transformation is as follows:

On définit une traduction du λ-calcul pur vers OBJ, qui fait disparaître les fonctions en les codant à l'aide d'objets. La traduction est la suivante :

$$\begin{aligned}
(\!|x|\!) &= x \\
(\!|\lambda x.e|\!) &= \langle \mathsf{arg} = ?\,; \mathsf{res} = \zeta z.\mathsf{let}\ x = z\#\mathsf{arg}\ \mathsf{in}\ (\!|e|\!)\rangle \\
(\!|v_1\ v_2|\!) &= ((\!|v_1|\!)\langle \mathsf{arg} \leftarrow \zeta\_.(\!|v_2|\!)\rangle)\#\mathsf{res}
\end{aligned}$$

In the above definition, the body of the arg method, written $?$, is left unspecified.

Dans la définition ci-dessus, le corps de la méthode arg, noté $?$, n'est pas précisé.

Question 7

Prove that this transformation preserves reduction: $e \longrightarrow e'$ implies $(\!|e|\!) \longrightarrow^+ (\!|e'|\!)$. Please deal explicitly only with the proof cases that you deem interesting. If an auxiliary lemma is needed, please state this lemma and use it, without proving it.

Montrez que cette traduction préserve la réduction : $e \longrightarrow e'$ implique $(\!|e|\!) \longrightarrow^+ (\!|e'|\!)$. Vous ne traiterez explicitement que les cas de preuve que vous jugerez intéressants. Si un lemme auxiliaire est nécessaire, énoncez ce lemme et utilisez-le sans le démontrer.

□

4 OBJ$_1$: a type system for OBJ / un système de types pour OBJ

We consider OBJ$_1$, an *implicitly-typed* first-order type system for OBJ. Object types are the *only* types, therefore defined by the grammar:

On considère OBJ$_1$, une version *implicitement typée* de premier ordre de OBJ. Les types objets sont les *seuls* types, et sont ainsi définis par la grammaire :

$$\tau ::= \langle \ell_i : \tau_i \rangle^{i \in I}$$

A typing environment Γ binds variables to types. The empty environment is $\emptyset$ and $\Gamma, x : \tau$ extends Γ with a new binding provided $x \notin dom(\Gamma)$. Typing judgments are of the form $\Gamma \vdash e : \tau$ and defined by the following rules:

Un environnement de typage Γ lie des variables à des types. L'environnement vide est $\emptyset$ and $\Gamma, x : \tau$ étend Γ avec un nouveau binding pourvu que $x \notin dom(\Gamma)$. Les jugements de typage sont de la forme $\Gamma \vdash e : \tau$ et définis par les règles suivantes :

Var
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Object
$$\frac{(\Gamma, x_i : \tau \vdash b : \tau_i)^{i \in I} \qquad \tau = \langle \ell : \tau_i \rangle^{i \in I}}{\Gamma \vdash \langle \ell_i = \zeta x_i.b_i \rangle^{i \in I} : \tau}$$

Call
$$\frac{\Gamma \vdash v : \langle \ell : \tau_i \rangle^{i \in I} \qquad j \in I}{\Gamma \vdash v\#\ell_j : \tau_j}$$

Override
$$\frac{\Gamma \vdash v : \tau \qquad \tau = \langle \ell_i : \tau_i \rangle^{i \in I} \qquad j \in I \qquad \Gamma, x_j : \tau \vdash b_j : \tau_j}{\Gamma \vdash v\langle \ell_j \leftarrow \zeta x_j.b_j \rangle : \tau}$$

Let
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

Question 8

Consider:

Considérez :

$$v_1 \overset{\mathsf{def}}{=} \langle \mathsf{val} = \zeta x.\langle\rangle ; \mathsf{run} = \zeta x.\langle \mathsf{ret} = \zeta y.x\#\mathsf{val}\rangle\rangle$$

Give the type τ_1 of v_1 and its typing derivation.

Donnez le type τ_1 de v_1 et sa dérivation de typage.

□

You may assume the basic inversion, permutation, and weakening lemmas; you may also asume that the substitution lemma for expressions and the compositionality lemma hold. You do not have to state them, but **you must explicitly invoke them when using it**.

Vous pouvez utiliser les lemmes de bases d'inversion, de permutation et d'affaiblissement ; vous pouvez aussi supposer que le lemme de substitution des expressions et de compositionalité sont satisfaits. Vous n'avez pas à les énoncer, mais **vous devez explicitement les invoquer lorsque vous les utiliser**.

Question 9

State and prove subject reduction for OBJ$_1$. (You may omit the cases for method overriding, sequencing, and evaluation under a context.)

Formulez et montrez l'auto-réduction pour OBJ$_1$. (Vous pourrez omettre le cas pour l'overdide de méthodes, les séquences et l'évaluation sous contexte.)

□

Question 10

State and show the progress lemma for OBJ$_1$.

Formulez and prouvez le lemme de progression pour OBJ$_1$.

□

Question 11

Give, without justification, derived typing rules for the syntactic sugar $e\#\ell$ and $e\langle \ell \leftarrow \zeta x.b\rangle$.

Donnez, sans justification, des règles de typage dérivées pour le sucre syntaxique $e\#\ell$ et $e\langle \ell \leftarrow \zeta x.b\rangle$.

□

5 Type preservation by $[\![\cdot]\!]$ / Préservation du typage par $[\![\cdot]\!]$

We wish to show that the image of OBJ$_1$ by the encoding given in Section 2 is well-typed. We consider LAM$_1$, the implicitly-typed simply-typed lambda-calculus with primitive records and equi-recursive types, which is a subset of OBJLAM. The types of LAM$_1$ are:

Nous souhaitons montrer que l'image de OBJ$_1$ par le codage donné dans la Section 2 est bien typée. Nous considérons LAM$_1$, le lamba-calcul simplement typé implicitement typé étendu avec des enregistrements primitifs et des types équi-récusrifs, qui forme un sous-ensemble de OBJLAM. Les types de LAM$_1$ sont :

$$\tau ::= \alpha \mid \tau \to \tau \mid \{\ell_i : \tau_i\}^{i \in I} \mid \mu\alpha.\tau$$

with the restriction that $\mu\alpha.\tau$ is only allowed when τ is an arrow type or a record type.

avec la restriction que $\mu\alpha.\tau$ n'est autorisé que lorsque τ est un type flèche ou un type enregistrement.

Question 12

Say briefly why we impose this syntactic restriction on the formation of recursive types.

Dire brièvement pourquoi nous imposons cette restriction syntaxique sur la formaltion des types récursifs.

□

We consider equality of recursive types only up to the fold/unfold rule:

Nous considérons l'égalité des types récursifs uniquement modulo la règle fold/unfold :

$$\mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha]$$

The typing rules of LAM$_1$ are rules Var and Let, as defined for OBJ$_1$, unchanged, plus the following five rules:

Les règles de typages de LAM$_1$ sont les règles Var et Let telles que définies pour OBJ$_1$, plus les cinq règles suivantes :

Lam
$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

App
$$\frac{\Gamma \vdash v_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash v_1\ v_2 : \tau_1}$$

Record
$$\frac{(\Gamma \vdash v_i : \tau_i)^{i \in I}}{\Gamma \vdash \{\ell_i : v_i\}^{i \in I} : \{\ell_i : \tau_i\}^{i \in I}}$$

Access
$$\frac{(\Gamma \vdash v : \{\ell_i : \tau_i\}^{i \in I} \qquad j \in I}{\Gamma \vdash : v.\ell_j : \tau_j}$$

Update
(Intentionally omitted)

Question 13

Give the typing rule Update for record update.

Donnez la règle de typage Update la mise à jour des enregistrements.

□

We now consider the encoding $[\![\cdot]\!]$ of Section 2 restricted to terms of OBJ$_1$ and we wish to show that the image by the encoding is in LAM$_1$, *i.e.* $[\![\mathrm{OBJ}_1]\!] \subseteq \mathrm{LAM}_1$.

Nous considérons maintenant le codage $[\![\cdot]\!]$ de la section 2 restreint aux termes de OBJ$_1$ et nous souhaitons montrer que l'image par le codage est dans LAM$_1$, , *i.e.* $[\![\mathrm{OBJ}_1]\!] \subseteq \mathrm{LAM}_1$.

Question 14

Explain why there is no hope that the encoding preserves well-typedness if we remove recursive types in LAM$_1$.

Expliquez pourquoi il est sans espoir que le codage préserve le typage si nous retirons les types récursifs dans LAM$_1$.

□

Question 15

Propose an encoding $[\![\cdot]\!]$ of types that extends point-wise to an encoding of typing contexts such that for any e in OBJ$_1$, $\Gamma \vdash e : \tau$ implies $[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]$. □

Question 16

Show $[\![\mathrm{OBJ}_1]\!] \subseteq \mathrm{LAM}_1$. (You may ommit the case of method override.)

Montrez que $[\![\mathrm{OBJ}_1]\!] \subseteq \mathrm{LAM}_1$. (Vous pourrez omettre le cas de method override.)

□

6 Type preservation by $(\!|\cdot|\!)$ / Préservation du typage par $(\!|\cdot|\!)$

Question 17

The encoding $(\!|\cdot|\!)$ of functions given in Section 3 uses a question mark "?" that can be filled with any term in the untyped encoding. Give a method body $\zeta x.b_?$ for the method arg that could be used in place of the question mark in the untyped encoding, so that the encoding has a chance to send well-typed terms of LAM$_1$ into well-typed terms of OBJ$_1$.

Le codage $(\!|\cdot|\!)$ des fonctions donné dans la Section 3 utilise le point d'interrogation "?" qui peut être remplacé par n'importe quel terme dans le codage non typé. Donnez un corps de méthode $\zeta x.b_?$ de la méthode arg qui pourrait être utilisé à la place du point d'interrogationa dans le codage non typé, de telle façon que le codage ait une chance de traduire les termes bien typés de LAM$_1$ en des termes bien typés de OBJ$_1$.

□

Question 18

The encoding of LAM$_1$ into OBJ$_1$ does not always preserve well-typedness. Can you give one example and explain the problem briefly?

Le codage de LAM$_1$ dans OBJ$_1$ ne préserve pas toujours le typage. Pouvez-vous donner un exemple et expliquer le problème brièvement ?

□

We may design a more expressive version OBJ$_2$ of OBJ$_1$, equipped with equi-recursive types:

Nous pouvons concevoir une version OBJ$_2$ plus expressive que OBJ$_1$ en l'équipant de types equi-récursifs :

$$\tau ::= \alpha \mid \mu.\tau \mid \langle \ell_i : \tau_i \rangle^{i \in I}$$

where $\mu\alpha$ can only be used in front of object types. This induces the fold/unfold equality between object types:

où $\mu\alpha$ ne peut être utilisé que devrant des types objets. Cela induit l'égalité fold/unfold entre les types récursifs :

$$\mu\alpha.\langle \ell_i : \tau_i \rangle^{i \in I} = \langle \ell_i : \tau_i[\mu\alpha.\langle \ell_i : \tau_i \rangle^{i \in I}/\alpha] \rangle$$

Question 19

Give an expression e that is not well-typed in OBJ$_1$ and a type τ such that $e : \tau$ holds in OBJ$_2$. Give its typing derivation.

Donnez une expression e non typable dans OBJ$_1$ et un type τ tel que $e : \tau$ soit satisfait dans OBJ$_2$. Donnez sa dérivation de typage.

□

Question 20

Assuming OBJ 2 to be extended with all constructs of OBJLAM, what would be the type of $P[k]$? (The typing derivation is not requested.)

□

7 Subyping (short but difficult) / Sous-typage (court mais difficile)

We may extend OBJ$_1$ with width subtyping, *i.e.* with a subtyping relation $\leq$ defined by rule WIDTH and a subtyping rule SUB.

Nous pouvons étendre OBJ$_1$ avec du sous-typage en largeur, *i.e.* avec une relation de sous-typage définie par la règle WIDTH et une règle de sous-typage SUB.

Width

$$\frac{J \subseteq I}{\langle \ell_i : \tau_i \rangle^{i \in I} \leq \langle \ell_i : \tau_i \rangle^{i \in J}}$$

Sub

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}$$

Question 21

Does the encoding $[\![\cdot]\!]$ preserve well-typedness if we also extend LAM$_1$ with width subtyping for records?

Est-ce que le codage $[\![\cdot]\!]$ préserver le typage si nous étendons également LAM$_1$ avec du sous-typage en largeur pour les enregistrements ?

$\square$

Question 22

Can we also add depth subtyping in OBJ$_1$? *i.e.* the the subtyping rule Depth.

Pouvons nous aussi ajouter du sous-typage en profondeur dans OBJ$_1$? *i.e.* la règle de sous-typage Depth.

Depth

$$\frac{(\tau_i \leq \tau_i{}')^{i \in I}}{\langle \ell_i : \tau_i \rangle^{i \in I} \leq \langle \ell_i : \tau_i{}' \rangle^{i \in I}}$$

$\square$

Question 23

Could we add the following subtyping rule in OBJ$_2$?

Pourrions nous aussi ajouter la règle de sous-typage suivante dans OBJ$_2$?

Mu-Width

$$\frac{J \subseteq I}{\mu\alpha.\langle \ell_i : \tau_i \rangle^{i \in I} \leq \mu\alpha.\langle \ell_i : \tau_i \rangle^{i \in J}}$$

$\square$

8 Solutions

Question 1

If left-to-right evaluation is desired, a standard definition of the syntactic sugar $e_1\ e_2$ would be $\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ x_1\ x_2$, where x_1 and x_2 are fresh variables. Similarly, $e\#\ell$ could be syntactic sugar for $\mathsf{let}\ x = e\ \mathsf{in}\ x\#\ell$, where x is a fresh variable.

Question 2

Strictly speaking, the set method takes no parameter and returns a λ-abstraction $\lambda x.\mathit{self}\langle\mathsf{get} \leftarrow \zeta\mathit{self}.x\rangle$. At an intuitive level, however, it is preferable to think of it as a method that takes one parameter x and returns $\mathit{self}\langle\mathsf{get} \leftarrow \zeta\mathit{self}.x\rangle$. This is a new object, an updated 1-dimensional point whose coordinate, encoded within its get method, is x.

The inc method uses a method call $\mathit{self}\#\mathsf{get}$ to read this point's current coordinate, then increments this coordinate by 1, and uses another method call $\mathit{self}\#\mathsf{set}\ (\ldots)$ to produce an updated point. In short, it produces a new point, which compared to this point is located one unit towards the right.

Question 3

Let us first note that, for every integer k, the expression $P[k]$ is a value, so $P[k]\#\mathsf{get}$ is a valid expression. (It is not sugar.) According to the reduction rule Method Call, this expression reduces in one step to $\widehat{k}$.

Question 4

When e is an expression, we view $e\#\ell$ as a meta-level notation for $\mathsf{let}\ x = e\ \mathsf{in}\ x\#\ell$. As such, this notation can be unfolded or folded back at will. (The assignment was not very clear about this.) Thus, we note that, when e reduces to e', $e\#\ell$ reduces in one or two steps to $e'\#\ell$. Indeed, $e\#\ell$ is the same as $\mathsf{let}\ x = e\ \mathsf{in}\ x\#\ell$, which reduces to $\mathsf{let}\ x = e'\ \mathsf{in}\ x\#\ell$, which is the same as $e'\#\ell$ (if e' is not a value) or reduces to $e'\#\ell$ (if e' is a value). Thus, roughly speaking, one can think of $[\,]\#\ell$ as an evaluation context.

The sugar for applications is slightly more complicated, as it involves two subexpressions, so we will expand it explicitly in the following.

We have the following reduction sequence:

$$\begin{array}{rl}
 & \mathsf{let}\ \mathit{point} = P[0]\ \mathsf{in}\ \mathit{point}\#\mathsf{inc}\#\mathsf{get} \\
\longrightarrow & P[0]\#\mathsf{inc}\#\mathsf{get} \\
\longrightarrow & (P[0]\#\mathsf{set}\ (P[0]\#\mathsf{get} + \widehat{1}))\#\mathsf{get} \\
\equiv & (\mathsf{let}\ x_1 = P[0]\#\mathsf{set}\ \mathsf{in}\ \mathsf{let}\ x_2 = P[0]\#\mathsf{get} + \widehat{1}\ \mathsf{in}\ x_1\ x_2)\#\mathsf{get} \\
\longrightarrow & (\mathsf{let}\ x_1 = (\lambda x.P[0]\langle\mathsf{get} \leftarrow \zeta\mathit{self}.x\rangle)\ \mathsf{in}\ \mathsf{let}\ x_2 = P[0]\#\mathsf{get} + \widehat{1}\ \mathsf{in}\ x_1\ x_2)\#\mathsf{get} \\
\longrightarrow & (\mathsf{let}\ x_2 = P[0]\#\mathsf{get} + \widehat{1}\ \mathsf{in}\ (\lambda x.P[0]\langle\mathsf{get} \leftarrow \zeta\mathit{self}.x\rangle)\ x_2)\#\mathsf{get} \\
\longrightarrow & (\mathsf{let}\ x_2 = \widehat{0} + \widehat{1}\ \mathsf{in}\ (\lambda x.P[0]\langle\mathsf{get} \leftarrow \zeta\mathit{self}.x\rangle)\ x_2)\#\mathsf{get} \\
\longrightarrow & (\mathsf{let}\ x_2 = \widehat{1}\ \mathsf{in}\ (\lambda x.P[0]\langle\mathsf{get} \leftarrow \zeta\mathit{self}.x\rangle)\ x_2)\#\mathsf{get} \\
\longrightarrow & ((\lambda x.P[0]\langle\mathsf{get} \leftarrow \zeta\mathit{self}.x\rangle)\ \widehat{1})\#\mathsf{get} \\
\longrightarrow & (P[0]\langle\mathsf{get} \leftarrow \zeta\mathit{self}.\widehat{1}\rangle)\#\mathsf{get} \\
\equiv & P[1]\#\mathsf{get} \\
\longrightarrow & \widehat{1}
\end{array}$$

Question 5

First, it is necessary to remark that the translation $[\![v]\!]$ of a value v is a value. Then, one proves the statement by induction on the derivation of $e \longrightarrow e'$. There is one proof case per reduction rule. Only two cases, Method Call and Method Override, are nontrivial.

In the case of Method Call, the reduction step is

$$v\#\ell_j \longrightarrow b_j[v/x_j]$$

where $v = \langle \ell_i = \zeta x_i.b_i \rangle^{i\in I}$. We have:

$$\begin{array}{rl}
 & [\![v\#\ell_j]\!] \\
\equiv & \mathsf{let}\ x = [\![v]\!]\ \mathsf{in}\ x.\ell_j\ (x) \\
\equiv & \mathsf{let}\ x = \{\ell_i = \lambda x_i.[\![b_i]\!]\}^{i\in I}\ \mathsf{in}\ x.\ell_j\ (x) \\
\longrightarrow & \{\ell_i = \lambda x_i.[\![b_i]\!]\}^{i\in I}.\ell_j\ (\{\ell_i = \lambda x_i.[\![b_i]\!]\}^{i\in I}) \\
\equiv & \mathsf{let}\ x_1 = \{\ell_i = \lambda x_i.[\![b_i]\!]\}^{i\in I}.\ell_j\ \mathsf{in}\ \mathsf{let}\ x_2 = \{\ell_i = \lambda x_i.[\![b_i]\!]\}^{i\in I}\ \mathsf{in}\ x_1\ x_2 \\
\longrightarrow & \mathsf{let}\ x_1 = \lambda x_j.[\![b_j]\!]\ \mathsf{in}\ \mathsf{let}\ x_2 = \{\ell_i = \lambda x_i.[\![b_i]\!]\}^{i\in I}\ \mathsf{in}\ x_1\ x_2 \\
\longrightarrow^2 & (\lambda x_j.[\![b_j]\!])\ \{\ell_i = \lambda x_i.[\![b_i]\!]\}^{i\in I} \\
\longrightarrow & [\![b_j]\!][\{\ell_i = \lambda x_i.[\![b_i]\!]\}^{i\in I}/x_j] \\
\equiv & [\![b_j]\!][[\![v]\!]/x_j] \\
\equiv & [\![b_j[v/x_j]]\!]
\end{array}$$

The last equality in the series is an application of a substitution lemma, which states that the transformation commutes with substitution: $[\![e]\!][[\![v]\!]/x]$ is $[\![e[v/x]]\!]$. This lemma naturally holds because the image of a variable x through the transformation is x itself.

In the case of Method Override, the reduction step is

$$v\langle \ell_j \leftarrow \zeta x_j.b'_j \rangle \longrightarrow \langle \ell_j = \zeta x_j.b'_j ; \ell_i = \zeta x_i.b_i{}^{i\in I\setminus\{j\}} \rangle$$

where $v = \langle \ell_i = \zeta x_i.b_i \rangle^{i\in I}$. We have:

$$\begin{array}{rl}
 & [\![v\langle \ell_j \leftarrow \zeta x_j.b'_j \rangle]\!] \\
\equiv & [\![v]\!]\{\ell_j \leftarrow \lambda x_j.[\![b'_j]\!]\} \\
\equiv & \{\ell_i = \lambda x_i.[\![b_i]\!]\}^{i\in I}\{\ell_j \leftarrow \lambda x_j.[\![b'_j]\!]\} \\
\longrightarrow & \{\ell_j = \lambda x_j.[\![b'_j]\!]; \ell_i = \lambda x_i.[\![b_i]\!]^{i\in I\setminus\{j\}}\} \\
\equiv & [\![\langle \ell_j = \zeta x_j.b'_j ; \ell_i = \zeta x_i.b_i{}^{i\in I\setminus\{j\}} \rangle]\!]
\end{array}$$

Question 6

We have noted already that if v is a value, then $[\![v]\!]$ is a value. In combination with the result of Question 5, we find that $e \longrightarrow^\star v$ implies $[\![e]\!] \longrightarrow^\star [\![v]\!]$: if e ultimately reduces to a value, then $[\![e]\!]$ ultimately reduces to a value as well. Furthermore, by Question 5, $e \longrightarrow^\infty$ implies $[\![e]\!] \longrightarrow^\infty$: if e diverges, then $[\![e]\!]$ diverges as well. Thus, both convergence to a value and divergence are preserved. One might wish to also prove that the property of going wrong (that is, reducing in zero or more steps to a stuck term) is preserved; we do not do so here, although it would be easy.

Question 7

First, one must note again that if v is a value, then $(\!|v|\!)$ is a value as well. Then, one proves the statement by induction on the derivation of $e \longrightarrow e'$.

In the case of Function Call, the reduction step is

$$(\lambda x.e)\ v \longrightarrow e[v/x]$$

We have:

$$\begin{array}{rl}
 & (\!|(\lambda x.e)\ v|\!) \\
\equiv & ((\!|\lambda x.e|\!)\langle \mathsf{arg} \leftarrow \zeta\_.(\!|v|\!)\rangle)\#\mathsf{res} \\
\equiv & (\langle \mathsf{arg} = ?\,; \mathsf{res} = \zeta z.\mathsf{let}\ x = z\#\mathsf{arg}\ \mathsf{in}\ (\!|e|\!)\rangle\langle \mathsf{arg} \leftarrow \zeta\_.(\!|v|\!)\rangle)\#\mathsf{res} \\
\equiv & \mathsf{let}\ x = (\langle \mathsf{arg} = ?\,; \mathsf{res} = \zeta z.\mathsf{let}\ x = z\#\mathsf{arg}\ \mathsf{in}\ (\!|e|\!)\rangle\langle \mathsf{arg} \leftarrow \zeta\_.(\!|v|\!)\rangle)\ \mathsf{in}\ x\#\mathsf{res} \\
\longrightarrow & \mathsf{let}\ x = (\langle \mathsf{arg} = \zeta\_.(\!|v|\!); \mathsf{res} = \zeta z.\mathsf{let}\ x = z\#\mathsf{arg}\ \mathsf{in}\ (\!|e|\!)\rangle)\ \mathsf{in}\ x\#\mathsf{res} \\
\longrightarrow & \langle \mathsf{arg} = \zeta\_.(\!|v|\!); \mathsf{res} = \zeta z.\mathsf{let}\ x = z\#\mathsf{arg}\ \mathsf{in}\ (\!|e|\!)\rangle\#\mathsf{res} \\
\longrightarrow & \mathsf{let}\ x = \langle \mathsf{arg} = \zeta\_.(\!|v|\!); \mathsf{res} = \zeta z.\mathsf{let}\ x = z\#\mathsf{arg}\ \mathsf{in}\ (\!|e|\!)\rangle\#\mathsf{arg}\ \mathsf{in}\ (\!|e|\!) \\
\longrightarrow & \mathsf{let}\ x = (\!|v|\!)\ \mathsf{in}\ (\!|e|\!) \\
\longrightarrow & (\!|e|\!)[(\!|v|\!)/x] \\
\equiv & (\!|e[v/x]|\!)
\end{array}$$

As in the answer to Question 5, the last equality in the series is an application of a substitution lemma, which states that the transformation commutes with substitution: $(\!|e|\!)[(\!|v|\!)/x]$ is $(\!|e[v/x]|\!)$.

It is worth noting that the method body ? is overwritten when the arg method is overridden, so its definition does not matter.

Question 8

τ_1 is $\langle \mathsf{val} : \langle\rangle ; \mathsf{run} : \langle \mathsf{ret} : \langle\rangle\rangle\rangle$ and we have

$$\dfrac{\textsc{Var}\;\dfrac{}{x : \tau_1 \vdash \langle\rangle : \langle\rangle} \qquad \dfrac{\dfrac{\dfrac{}{x : \tau_1, y : \langle \mathsf{ret} : \langle\rangle\rangle \vdash x : \tau_1}\;\textsc{Var}}{x : \tau_1, y : \langle \mathsf{ret} : \langle\rangle\rangle \vdash x\#\mathsf{val} : \langle\rangle}\;\textsc{Access}}{x : \tau_1 \vdash \langle \mathsf{ret} = \zeta y.x\#\mathsf{val}\rangle : \langle \mathsf{ret} : \langle\rangle\rangle}\;\textsc{Object}}{\vdash v_1 : \tau_1}$$

□

Question 9

Subject Reduction If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$.

Subject reduction could be stated (and proved directly) for an context Γ.

Proof. . We reason by induction on the reduction.

Case Method-Call: $v\#\ell_j \longrightarrow b_j[v/x_j]$ where v is $\langle \ell_i = b_i\rangle^{i\in I}$ and $j \in I$: We assume $\Gamma \vdash v\#\ell_j : \tau_j$ (**1**), and show $\Gamma \vdash b_j[v/x_j] : \tau_j$ (**2**). By inversion of typing applied to (1), we have $\Gamma \vdash v : \tau$ (**3**), hence $\Gamma, x_i : \tau \vdash b_i : \tau_i$ for $i \in I$, and in particular, $\Gamma, x_j : \tau \vdash b_j : \tau_j$ (**4**). The conclusion (2) follows by the substitution lemma applied to (4) and (3).

Case Method Override $v\langle \ell \leftarrow \zeta x_j.b_j\rangle \longrightarrow \langle \ell_j = \zeta x_j.b'_j; \ell_i = \zeta x_i.b_i^{i\in I\setminus\{j\}}\rangle$ where v is $\langle \ell_i = b_i\rangle^{i\in I}$ and $j \in I$: Omitted.

Case Sequencing $\mathsf{let}\ x = v\ \mathsf{in}\ e \longrightarrow e[v/x]$: Omitted.

Case Context $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \longrightarrow \mathsf{let}\ x = {e_1}'\ \mathsf{in}\ e_2$ where $e_1 \longrightarrow {e_1}'$: Omitted.

Question 10

Progress If $\emptyset \vdash e : \tau$ then either e is a closed value $\langle \ell_i = b_i\rangle^{i\in I}$ or there exists e' such that $e \longrightarrow e'$.

Proof. We assume $\emptyset \vdash e : \tau$, then reason by structural induction on e.

Case e is x: By inversion of typing, this case is not possible.

Case e is $\langle \ell_i = b_i\rangle^{i\in I}$: This is an object.

Case e is $v\#\ell_j$: By inversion of typing, we have $\emptyset \vdash v : \tau'$ (**1**) where τ' is $\langle \ell_i : b_i\rangle^{i\in I}$ and $j \in I$. Since v is a value, it must be an object. By inversion of typing it must be of the form $\langle \ell_i = b_i\rangle^{i\in I}$. Then, $v\#\ell_j$ reduces by Rule Method Call.

Case e is $v\langle \ell \leftarrow \zeta x_j.b_j\rangle$: By inversion of typing, we have $\emptyset \vdash v : \tau$. Hence, it is a closed value, *i.e.* an object of the form $\langle \ell_i = b_i\rangle^{i\in I}$. Therefore, $v\langle \ell \leftarrow \zeta x_j.b_j\rangle$ reduces by Method Override.

Case sequencing $\emptyset \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau$: If e_1 is a value, e reduces by Rule Sequencing. Otherwise, by inversion of typing, we have $\emptyset \vdash e_1 : \tau_1$. By IH, e_1 reduces and then e reduces by Rule Context.

Question 11

$$\textsc{Call'}\quad \dfrac{\Gamma \vdash e : \langle \ell : \tau_i\rangle^{i\in I} \qquad j \in I}{\Gamma \vdash e\#\ell_j : \tau_j} \qquad\qquad \textsc{Override'}\quad \dfrac{\Gamma \vdash e : \tau \qquad \tau = \langle \ell_i : \tau_i\rangle^{i\in I} \qquad j \in I \qquad \Gamma, x_j : \tau \vdash b_j : \tau_j}{\Gamma \vdash e\langle \ell_j \leftarrow \zeta x_j.b_j\rangle : \tau}$$

Question 12

$\mu\alpha.\alpha$ would be ill-defined—it does not define an infinite (regular) tree. $\mu\alpha_1.\mu\alpha_2.\tau$ is useless as α_1 and α_2 would bind the same tree.

Question 13

$$\frac{\Gamma \vdash v : \{\ell_i : \tau_i\}^{i\in I} \qquad \Gamma \vdash v_j : {\tau_j}'}{\Gamma \vdash v\{\ell_j \leftarrow v_j\} : \{\ell_j : {\tau_j}'; (\ell_i : \tau_i)^{i \in I\setminus\{j\}}\}}\ \textsc{Update}$$

(Note that we do not need to request $j \in I$ nor ${\tau_j}' = \tau_j$ whenever $j \in I$.)

Question 14

Programs of OBJ$_1$ do not necessarily terminate. For example, e_0 equal to $\langle \ell = \zeta x.x\#\ell\rangle\#\ell$ loops. Since the encoding preserves the semantics, $[\![e_0]\!]$ must also loop, but in the absence of recursive types in LAM$_1$, all programs, including $[\![e_0]\!]$ should terminate.

Question 15

$$[\![\langle \ell_i : \tau_i\rangle^{i\in I}]\!] = \mu\alpha.\{\ell_i : \alpha \to [\![\tau_i]\!]\}^{i\in I}$$

Indeed, the encoding of an object $\langle \ell_i : \zeta x_i.b_i\rangle^{i\in I}$ is a record of functions $\{\ell_i : \lambda x_i.b_i\}^{i\in I}$, hence of type $\{\ell_i : \tau \to [\![\tau_i]\!]\}$ where τ is the record type itself, that is the recursive type $\mu\alpha.\{\ell_i : \alpha \to [\![\tau_i]\!]\}^{i\in I}$.

Question 16

The proof is by induction on the typing derivation of e, and in each case starts by inversion of typing.

Case $\Gamma \vdash x : \tau$: We have $x : \tau \in \Gamma$, hence by definition of $[\![\Gamma]\!]$, we have $x : [\![\tau]\!] \in [\![\Gamma]\!]$ which implies $[\![\Gamma]\!] \vdash [\![x]\!] : [\![\tau]\!]$.

Case $\Gamma \vdash \langle \ell_i = \zeta x_i.b_i\rangle^{i\in I} : \tau$: We have $(\Gamma, x_i : \tau \vdash b : \tau_i)^{i\in I}$ and $\tau = \langle \ell : \tau_i\rangle^{i\in I}$. By IH, we have $[\![\Gamma]\!], x_i : [\![\tau]\!] \vdash [\![b]\!] : [\![\tau_i]\!]$, hence $[\![\Gamma]\!] \vdash [\![\lambda x_i.b]\!] : [\![\tau]\!] \to [\![\tau_i]\!]$ for all $i \in I$. Therefore, $[\![\Gamma]\!] \vdash [\![e]\!] : \{\ell_i : [\![\tau]\!] \to [\![\tau_i]\!]\}^{i\in I}$. The type $\{\ell_i : [\![\tau]\!] \to [\![\tau_i]\!]\}^{i\in I}$ happens to be the unfolding of (hence equal to) $[\![\tau]\!]$.

Case $\Gamma \vdash v\#\ell_j : \tau_j$: We have $\Gamma \vdash v : \tau'$ where $\tau' \overset{\text{def}}{=} \langle \ell : \tau_i\rangle^{i\in I}$ and $j \in I$. By IH, we have $[\![\Gamma]\!] \vdash [\![v]\!] : [\![\tau']\!]$. Since $[\![\tau']\!]$ is equal to $\mu\alpha.\{\ell_i : \alpha \to [\![\tau_i]\!]\}^{i\in I}$, *i.e.* also equal to $\{\ell_i : [\![\tau']\!] \to [\![\tau_i]\!]\}^{i\in I}$, we have $[\![\Gamma]\!], x : [\![\tau']\!] \vdash x\#\ell_j\ x : [\![\tau_j]\!]$, and also $[\![\Gamma]\!] \vdash \mathsf{let}\ x = v\ \mathsf{in}\ v.\ell_j\ x : [\![\tau_j]\!]$ by Rule Let for a fresh variable x, that is $[\![\Gamma]\!] \vdash [\![v\#\ell_j]\!] : [\![\tau_j]\!]$, as expected.

Case $\Gamma \vdash v\langle \ell_j \leftarrow \zeta x_j.b_j\rangle : \tau$: Omitted.

Case $\Gamma \vdash$ let $x = e_1$ in $e_2 : \tau_2$: Obvious by IH.

Question 17

We must find a method body $\zeta x.b_?$ for arg independent of the function we are encoding. This expression should have all possible types, hence it must loop. The simplest solution is $\zeta x.x\#\mathsf{arg}$. Other solutions should similarly end up calling a same method recursively.

Question 18

LAM$_1$ has recursive types and therefore can type the expression $e \overset{\text{def}}{=} \lambda x.x\ x$. The OBJ$_1$ has a primitive construction that encodes self application, but does not have recursive types. The encoding of e cannot be typed: in particular, the encoding of $x\ x$ involves a subterm $x\langle \mathsf{arg} \leftarrow \zeta z.x\rangle$ whose typing requires variable x to have an object type τ with a method arg of type τ—this is not possible without a form of recursive types.

Question 19

$$e \overset{\mathsf{def}}{=} \langle \ell = \zeta x.x \rangle \qquad \tau \overset{\mathsf{def}}{=} \mu\alpha.\langle \ell : \alpha \rangle \qquad \frac{x : \tau \vdash x : \tau \qquad \tau = \langle \ell : \tau \rangle}{\emptyset \vdash \langle \ell : \zeta x.x \rangle : \tau}\ \textsc{Object}$$

Question 20

$$\mu\alpha.\left\langle \begin{array}{l} \mathsf{get} : \mathsf{int} \\ \mathsf{set} : \mathsf{int} \to \alpha \\ \mathsf{inc} : \alpha \end{array} \right\rangle$$

Question 21

No, because the encoding of an object type is a recursive type at a negative occurence, hence subtyping does not propagate through the μ binder.

Question 22

No: depth subtyping and overriding are incompatible, because this would allow the following unsound example:

$$\langle \ell = \langle \ell = ? \rangle, \ell' = \zeta x.\ell \# \ell \# x \rangle \langle \ell \leftarrow \langle\rangle \rangle \# \ell'$$

This would be well-typed because the type of field ℓ could be weakened to that of an empty object before being overriden with a method returning an empty object. However, this would get stuck when the unmodified method ℓ' expecting ℓ to return an object with a method ℓ is being called.

Question 23

This is incorrect if α occurs in τ_i for some i in J, even positively, because the recursive occurence would imply a form of Width subtyping.