École Normale Supérieure

# Langages de programmation et compilation

examen 2022–2023

Jean-Christophe Filliâtre

20 janvier 2023 — 8h30–11h30

L'épreuve dure 3 heures.
Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.
Les questions sont indépendantes, au sens où il n'est pas nécessaire d'avoir répondu aux questions précédentes pour traiter une question, mais en revanche les questions peuvent faire appel à des définitions ou à des résultats introduits dans les questions précédentes.
Les figures 2–4 sont regroupées en fin de sujet, page 7. Suggestion : détacher la dernière feuille.

Dans tout ce sujet, on considère un petit langage de programmation dont la syntaxe abstraite est donnée figure 2. C'est une variante du langage mini-ML étudié en cours, avec des entiers, des fonctions et des *enregistrements*. Un enregistrement est construit avec la syntaxe $\{a = 1; b = 2\}$, ici avec deux champs $a$ et $b$ de valeurs respectives 1 et 2. Si $r$ est un enregistrement possédant un champ $a$, alors $r.a$ désigne la valeur de ce champ. Dans la syntaxe abstraite, $f$ désigne un nom de champ. Une sémantique opérationnelle à petits pas est donnée figure 3. Elle traduit un passage par valeur, avec une évaluation de la gauche vers la droite.

**Question 1** Pour chacune des trois expressions suivantes, donner la suite de réductions et indiquer si elle se termine sur une valeur.

- `(fun (x:{a:int}) -> x.a) {a=42; b=55}`
- `let x = {a=1} in let y = {b=x.a; c=x.a} in y.a`
- `let x = {a=1} in (fun (y:{b:int}) -> x.a) x`

---

**Correction :**

```
   (fun (x:{a:int}) -> x.a) {a=42; b=55}
-> {a=42; b=55}.a
-> 42  // valeur

   let x = {a=1} in let y = {b=x.a; c=x.a} in y.a
-> let y = {b={a=1}.a; c={a=1}.a} in y.a
-> let y = {b=1; c={a=1}.a} in y.a
-> let y = {b=1; c=1} in y.a
-> {b=1; c=1}.a  // pas une valeur, calcul bloqué

   let x = {a=1} in (fun (y:{b:int}) -> x.a) x
-> (fun (y:{b:int}) -> {a=1}.a) {a=1}
-> {a=1}.a
-> 1  // valeur
```

---

**Analyse syntaxique.** On souhaite réaliser l'analyse syntaxique de notre petit langage avec l'outil Menhir. La figure 1 contient un fichier d'entrée pour Menhir contenant la grammaire des expressions de notre langage. Les actions sémantiques ne nous intéressent pas ici et sont omises (`{...}`).

```
expr:
| expr expr %prec app                        {...}
| LET IDENT EQ expr IN expr                  {...}
| FUN LPAR IDENT COLON typ RPAR ARROW expr   {...}
| CONST                                      {...}
| IDENT                                      {...}
| expr DOT IDENT                             {...}
| LBRA separated_list(SEMI, field) RBRA      {...}
| LPAR expr RPAR                             {...}
```

FIGURE 1 – Fichier Menhir.

**Question 2** Lorsque l'outil Menhir est lancé sur le fichier donné en figure 1, il déclare de nombreux conflits de type lecture/réduction (`shift/reduce`).

1. Donner un exemple de conflit.
2. Pour lever les conflits, on choisit de donner la priorité la plus forte à la construction $e.f$, puis à l'application (construction $e\ e$) puis enfin à toutes les autres constructions. Donner des déclarations de priorité et d'associativité que l'on peut indiquer à l'outil Menhir pour cela. La grammaire ne doit pas être modifiée. On notera que la règle d'application est nommée.

---

**Correction :**

1. Un exemple de conflit est `let x = e1 in e2 e3`, qui peut être compris comme `(let x = e1 in e2) e3` (si on favorise la réduction) ou comme `let x = e1 in (e2 e3)` (si on favorise la lecture).
2. On déclare ainsi les priorités :

```
%nonassoc LPAR LET FUN CONST IDENT LBRA ARROW IN
%nonassoc app
%nonassoc DOT
```

---

**Typage.** Notre langage est typé avec des types monomorphes, notés $\tau$ et définis figure 2. Le jugement de typage est noté $\Gamma \vdash e : \tau$, où l'environnement de typage $\Gamma$ donne le type des variables susceptibles d'apparaître dans $e$. Les règles de typage sont données figure 4. On ne cherche pas ici à faire de l'inférence de types, l'argument $x$ d'une fonction `fun` $(x : \tau) \to e$ venant avec un type $\tau$. Un enregistrement comme $\{a = 1; b = 2\}$ a un type de la forme $\{a : \texttt{int}; b : \texttt{int}\}$, qui donne le type de chacun de ses champs. (À la différence d'un langage comme OCaml, on ne déclare pas ici les types enregistrements.) Un enregistrement peut n'avoir aucun champ (c'est l'expression $\{\}$) et son type est alors $\{\}$.

La particularité de ce langage est le *sous-typage* qui découle des types enregistrements. L'idée est qu'un enregistrement comme $\{a = 1; b = 2\}$ peut être considéré comme une valeur de type $\{b : \texttt{int}\}$, car il possède *au moins* le champ $b$ attendu, avec le bon type. Plus précisément, on note $\tau \sqsubseteq \tau'$ la relation de sous-typage, qui s'interprète comme « toute valeur de type $\tau$ peut être acceptée comme une valeur de type $\tau'$ ». Cette relation est définie par les trois dernières règles de la figure 4. La relation de sous-typage est utilisée dans la règle de typage de l'application, pour accepter un argument $e_1$ d'un sous-type $\tau$ du type $\tau_1$ attendu par la fonction $e_2$.

**Question 3** Pour chacune des trois expressions suivantes, indiquer si elle est bien typée dans un environnement de typage vide. Le cas échéant, donner la dérivation de typage. Dans le cas contraire, justifier.

- `(fun (x:{a:int}) -> x.a) {a=42; b=55}`
- `fun (x:{a:int}->{b:int}) -> x {}`
- `let x = {} in { b=x; c=x }`

---

**Correction :**

- bien typée

```
  x:{a:int} |- x:{a:int}
  ----------------------
  x:{a:int} |- x.a : int       |-42:int   |-55:int
------------------------  --------------------------
|- fun ...: {a:int}->int   |-{a=42;b=55}:{a:int;b:int} {a:int;b:int}<={a:int}
-------------------------------------------------------------------------------
            |- (fun (x:{a:int}) -> x.a) {a=42; b=55} : int
```

- mal typée ; la dérivation aurait la forme suivante

```
    x:{a:int}->{b:int} |- x:{a:int}->{b:int}     |- {} : {}    ÉCHEC
    ---------------------------------------------------------------
                  x:{a:int}->{b:int} |- x {} : tau
   ------------------------------------------------------------------------
             |-fun (x:{a:int}->{b:int}) -> x {} : tau
```

  mais le type $\{\}$ n'est pas un sous-type de $\{a : \texttt{int}\}$.

- bien typée

```
                  ------------     ------------
                  x:{} |- x:{}     x:{} |- x:{}
    --------    --------------------------------
    |- {}:{}     x:{} |- {b=x;c=x} : {b:{};c:{}}
   ------------------------------------------------
       |- let x = {} in {b=x;c=x} : {b:{};c:{}}
```

---

**Question 4** Parmi les règles de la figure 4, on a notamment défini le sous-typage des types de fonctions de la manière suivante :

$$\frac{\tau_1' \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \tau_2'}{\tau_1 \ \to \ \tau_2 \sqsubseteq \tau_1' \ \to \ \tau_2'}$$

Justifier cette règle en donnant un exemple impliquant du sous-typage de fonctions, avec des types $\tau_1 \neq \tau_1'$ et $\tau_2 \neq \tau_2'$.

---

**Correction :** Considérons une expression comme

$$(\texttt{fun}\,(x : \{a : \texttt{int}; b : \texttt{int}\} \ \to \ \{c : \texttt{int}\}) \to \ldots x \ldots)\,(\texttt{fun}\,(y : \{b : \texttt{int}\}) \to \{c = 2; d = 3\})$$

La fonction attendue $x$ doit produire un résultat de type $\{c : \texttt{int}\}$, mais rien n'empêche la fonction effective de produire des valeurs d'un sous-type, à savoir ici $\{c : \texttt{int}; d : \texttt{int}\}$. Inversement, la fonction $x$ attend un argument de type $\{a : \texttt{int}; b : \texttt{int}\}$, ce qui veut dire qu'on lui passera des valeurs ayant au moins un champ $a$ et un champ $b$. Mais la fonction effective peut très bien exiger moins, à savoir ici n'attendre qu'un champ $b$.

---

**Question 5** Donner un exemple de « diminution » du typage pendant l'exécution, c'est-à-dire une expression $e$ telle que $\vdash e : \tau$ et une réduction $e \to e'$ telle que $\vdash e' : \tau'$ avec $\tau' \neq \tau$.

---

**Correction :** Il suffit d'appliquer l'identité à un enregistrement contenant « trop de champs » :

```
    (fun (x:{a:int}) -> x)  {a=1; b=2}   :   {a:int}
 -> {a=1; b:2}                           :   {a:int; b:int}
```

---

**Sûreté du typage.** On se propose maintenant de montrer que le typage de notre langage est sûr, au sens où l'évaluation d'une expression bien typée, si elle termine, conduit nécessairement à une valeur. On admet la propriété de progrès (si une expression $e$ est bien typée et n'est pas une valeur, alors $e \to e'$) et on se concentre ici sur la propriété de préservation, qui fait l'objet des questions suivantes.

**Question 6** Montrer que la relation de sous-typage $\sqsubseteq$ est transitive, c'est-à-dire que, si $\tau_1 \sqsubseteq \tau_2$ et $\tau_2 \sqsubseteq \tau_3$, alors $\tau_1 \sqsubseteq \tau_3$.

---

**Correction :** Par induction sur les types $\tau_1, \tau_2, \tau_3$ (la somme de leur tailles) et par cas sur $\tau_1 \sqsubseteq \tau_2$ et $\tau_2 \sqsubseteq \tau_3$.

- Si $\tau_1 = \tau_2$ ou $\tau_2 = \tau_3$, le résultat est immédiat.
- Si $\tau_1 = A \to B$, alors $\tau_2 = C \to D$ et $\tau_3 = E \to F$. On a $B \sqsubseteq D$ et $D \sqsubseteq F$ et par HR on a donc $B \sqsubseteq F$. De même, on a $E \sqsubseteq C$ et $C \sqsubseteq A$ et par HR on a donc $E \sqsubseteq A$. D'où $\tau_1 \sqsubseteq \tau_3$.
- Si $\tau_1 = \{f_i : \tau_i\}$, alors $\tau_2 = \{g_j : \tau_j\}$ et $\tau_3 = \{h_k : \tau_k\}$. Soit un champ $h_k : \tau_k$ de $\tau_3$. Comme $\tau_2 \sqsubseteq \tau_3$, il existe un champ $g_j : \tau_j$ de $\tau_2$ avec $g_j = h_k$ et $\tau_j \sqsubseteq \tau_k$. De même, comme $\tau_1 \sqsubseteq \tau_2$, il existe un champ $f_i : \tau_i$ de $\tau_1$ avec $f_i = g_j$ et $\tau_i \sqsubseteq \tau_j$. Par HR, on a $\tau_i \sqsubseteq \tau_k$ et il existe donc un champ $f_i : \tau_i$ de $\tau_1$ avec $f_i = h_k$ et $\tau_i \sqsubseteq \tau_k$.

---

**Question 7** Montrer que si $\Gamma + x : \tau_1 \vdash e_2 : \tau_2$ et $\tau_1' \sqsubseteq \tau_1$ alors $\Gamma + x : \tau_1' \vdash e_2 : \tau_2'$ avec $\tau_2' \sqsubseteq \tau_2$. *Indiquer sur quoi porte l'induction et traiter uniquement le cas d'une application,* i.e., $e_2 = e_4\ e_3$.

---

**Correction :** La preuve se fait par induction sur la dérivation de typage de $e_2$. On a

$$\frac{\Gamma + x : \tau_1 \vdash e_4 : \tau_3 \to \tau_4 \quad \Gamma + x : \tau_1 \vdash e_3 : \tau \sqsubseteq \tau_3}{\Gamma + x : \tau_1 \vdash e_4\ e_3 : \tau_4}$$

Notons $\Gamma' = \Gamma + x : \tau_1'$. Par HR, on a $\Gamma' \vdash e_4 : (\tau_3' \to \tau_4') \sqsubseteq (\tau_3 \to \tau_4)$ et $\Gamma' \vdash e_3 : \tau' \sqsubseteq \tau_3$. Or $\tau_3 \sqsubseteq \tau_3'$, donc $\tau' \sqsubseteq \tau_3'$ par transitivité (Q6). On en déduit

$$\frac{\Gamma' \vdash e_4 : \tau_3' \to \tau_4' \quad \Gamma' \vdash e_3 : \tau' \sqsubseteq \tau_3'}{\Gamma' \vdash e_4\ e_3 : \tau_4'}$$

et $\tau_4' \sqsubseteq \tau_4$.

---

**Question 8** Montrer la préservation du typage par substitution, à savoir que si $\Gamma + x : \tau_1 \vdash e_2 : \tau_2$ et $\Gamma \vdash e_1 : \tau_1'$ avec $\tau_1' \sqsubseteq \tau_1$ alors $\Gamma \vdash e_2[x \leftarrow e_1] : \tau_2'$ avec $\tau_2' \sqsubseteq \tau_2$. *Indiquer sur quoi porte l'induction et traiter uniquement les cas d'une application,* i.e., $e_2 = e_4\ e_3$, *et d'une projection,* i.e., $e_2 = e_3.f$.

---

**Correction :** Par induction sur la dérivation de typage de $e_2$.

- cas $e_2 = e_4\ e_3$ : on a

$$\frac{\Gamma + x : \tau_1 \vdash e_4 : \tau_3 \to \tau_2 \quad \Gamma + x : \tau_1 \vdash e_3 : \tau \sqsubseteq \tau_3}{\Gamma + x : \tau_1 \vdash e_4\ e_3 : \tau_2}$$

  Par HR, on a $\Gamma \vdash e_4[x \leftarrow e_1] : (\tau_3' \to \tau_2') \sqsubseteq (\tau_3 \to \tau_2)$ et $\Gamma \vdash e_3[x \leftarrow e_1] : \tau' \sqsubseteq \tau$. Or $\tau \sqsubseteq \tau_3 \sqsubseteq \tau_3'$. On en déduit

$$\frac{\Gamma \vdash e_4[x \leftarrow e_1] : \tau_3' \to \tau_2' \quad \Gamma \vdash e_3[x \leftarrow e_1] : \tau' \sqsubseteq \tau_3'}{\Gamma \vdash (e_4\ e_3)[x \leftarrow e_1] : \tau_2'}$$

  et $\tau_2' \sqsubseteq \tau_2$.

- cas $e_2 = e_3.f$ : on a

$$\frac{\Gamma + x : \tau_1 \vdash e_3 : \{\dots f : \tau_2 \dots\}}{\Gamma + x : \tau_1 \vdash e_3.f : \tau_2}$$

  Par HR, on a $\Gamma \vdash e_3[x \leftarrow e_1] : \tau' \sqsubseteq \{\dots f : \tau_2 \dots\}$. Donc $\exists f : \tau_2' \in \tau$ avec $\tau_2' \sqsubseteq \tau_2$, d'où

$$\frac{\Gamma \vdash e_3[x \leftarrow e_1] : \{\dots f : \tau_2' \dots\}}{\Gamma \vdash e_3.f[x \leftarrow e_1] : \tau_2'}$$

  et $\tau_2' \sqsubseteq \tau_2$.

---

**Question 9** Montrer la propriété de préservation du typage, à savoir que si $\Gamma \vdash e : \tau$ et $e \to e'$, alors $\Gamma \vdash e' : \tau'$ avec $\tau' \sqsubseteq \tau$. *Indiquer sur quoi porte l'induction et traiter uniquement les cas d'une application,* i.e., $e = e_2\ e_1$, *et d'un* `let`, i.e., $e =$ `let` $x = e_1$ `in` $e_2$.

---

**Correction :** Par induction sur la dérivation de typage $\Gamma \vdash e : \tau$.

- cas $e = e_2\ e_1$ : on a

$$\frac{\Gamma \vdash e_2 : \tau_2 \to \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \tau_1 \sqsubseteq \tau_2}{\Gamma \vdash e_2\ e_1 : \tau}$$

  — soit $e_2 \to e_2'$ : alors par HR $\Gamma \vdash e_2' : \tau_2' \to \tau' \sqsubseteq \tau_2 \to \tau$ et donc

$$\frac{\Gamma \vdash e_2' : \tau_2' \to \tau' \quad \Gamma \vdash e_1 : \tau_1 \quad \tau_1 \sqsubseteq \tau_2 \sqsubseteq \tau_2'}{\Gamma \vdash e_2'\ e_1 : \tau' \sqsubseteq \tau}$$

  — soit $e_2$ est une valeur et $e_1 \to e_1'$ : par HR $\Gamma \vdash e_1' : \tau_1' \sqsubseteq \tau$ et donc

$$\frac{\Gamma \vdash e_2 : \tau_2 \to \tau \quad \Gamma \vdash e_1' : \tau_1' \quad \tau_1' \sqsubseteq \tau_1 \sqsubseteq \tau_2}{\Gamma \vdash e_2\ e_1' : \tau}$$

  — soit $e_2$ et $e_1$ sont des valeurs, et dans ce cas $e_2 =$ `fun` $(x : \tau_3) \to e_3$ et $e \to e' = e_3[x \leftarrow e_1]$ et on peut appliquer le résultat de Q8.

- cas $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$ : on a

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

  — soit $e_1 \to e_1'$ : alors par HR $\Gamma \vdash e_1' : \tau_1' \sqsubseteq \tau_1$ et donc

$$\frac{\Gamma \vdash e_1' : \tau_1' \quad \Gamma + x : \tau_1' \vdash e_2 : \tau' \sqsubseteq \tau}{\Gamma \vdash \texttt{let}\ x = e_1'\ \texttt{in}\ e_2 : \tau'}$$

  par Q7.

  — soit $e_1$ est une valeur et alors $e \to e' = e_2[x \leftarrow e_1]$ et on peut appliquer le résultat de Q8.

---

**Ajout d'une conditionnelle.** On se propose d'ajouter une conditionnelle à notre langage, sous la forme d'une construction $\texttt{ifzero}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ qui évalue l'expression $e_1$, de type `int`, puis évalue l'expression $e_2$ si le résultat est zéro et l'expression $e_3$ sinon.

**Question 10** Ajouter cette nouvelle construction à la sémantique opérationnelle à petits pas de la figure 3, c'est-à-dire indiquer les modifications à apporter, le cas échéant, (1) à la notion de valeur, (2) aux réductions de tête et (3) aux contextes de réduction.

---

**Correction :** On ne modifie pas la notion de valeur. On ajoute deux nouvelles réduction de tête :

$$\begin{array}{rcl} \texttt{ifzero}\ 0\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \xrightarrow{\epsilon} & e_2 \\ \texttt{ifzero}\ n\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \xrightarrow{\epsilon} & e_3 \text{ avec } n \neq 0 \end{array}$$

Enfin, on ajoute un contexte de réduction :

$$\begin{array}{rcl} E & ::= & \ldots \\ & | & \texttt{ifzero}\ E\ \texttt{then}\ e\ \texttt{else}\ e \end{array}$$

---

**Type intersection.** On aimerait accepter au typage des programmes comme

```
if ... then { a=1; b=2 } else { b=3; c=4 }
```

ou encore

```
if ... then fun (x:{a:int;b:int}) -> 0
       else fun (x:{c:int})       -> 1
```

Pour cela, on propose la règle de typage suivante

$$\frac{\Gamma \vdash e : \texttt{int} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ifzero}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau_1 \vee \tau_2}$$

où l'opération $\tau_1 \vee \tau_2$ désigne le plus petit sur-type de $\tau_1$ et $\tau_2$, s'il existe, c'est-à-dire,

- $\tau_1 \sqsubseteq \tau_1 \vee \tau_2$
- $\tau_2 \sqsubseteq \tau_1 \vee \tau_2$
- pour tout type $\tau$, si $\tau_1 \sqsubseteq \tau$ et $\tau_2 \sqsubseteq \tau$ alors $\tau_1 \vee \tau_2 \sqsubseteq \tau$.

**Question 11** Dans chacun des cas suivants, indiquer si $\tau_1 \vee \tau_2$ existe et donner sa valeur le cas échéant :

| $\tau_1$ | $\tau_2$ | $\tau_1 \vee \tau_2$ |
|---|---|---|
| $\{a : \texttt{int}; b : \texttt{int}\}$ | $\{b : \texttt{int}; c : \texttt{int}\}$ | ??? |
| $\{a : \texttt{int}; b : \texttt{int}\} \to \texttt{int}$ | $\{c : \texttt{int}\} \to \texttt{int}$ | ??? |
| $\texttt{int}$ | $\texttt{int} \to \texttt{int}$ | ??? |

---

**Correction :**

| $\tau_1$ | $\tau_2$ | $\tau_1 \vee \tau_2$ |
|---|---|---|
| $\{a : \texttt{int}; b : \texttt{int}\}$ | $\{b : \texttt{int}; c : \texttt{int}\}$ | $\{b : \texttt{int}\}$ |
| $\{a : \texttt{int}; b : \texttt{int}\} \to \texttt{int}$ | $\{c : \texttt{int}\} \to \texttt{int}$ | $\{a : \texttt{int}; b : \texttt{int}; c : \texttt{int}\} \to \texttt{int}$ |
| $\texttt{int}$ | $\texttt{int} \to \texttt{int}$ | ÉCHEC |

---

**Question 12** Donner une définition *algorithmique* de l'opération $\vee$.

---

**Correction :** On procède récursivement sur la structure des types :

$$\begin{aligned} \tau \vee \tau &= \tau \\ \{f_i : \tau_i\} \vee \{g_j : \tau'_j\} &= \{h_k : \tau_k \vee \tau'_k\} \text{ avec } \{h_k\} = \{f_i\} \cap \{g_j\} \\ \tau_1 \to \tau_2 \vee \tau'_1 \to \tau'_2 &= \tau_1 \wedge \tau'_1 \to \tau_2 \vee \tau'_2 \end{aligned}$$

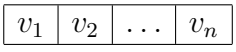
où l'opération $\wedge$ est définie de façon duale :

$$\begin{aligned} \tau \wedge \tau &= \tau \\ \{f_i : \tau_i\} \wedge \{g_j : \tau'_j\} &= \{h_k : \tau_k \wedge \tau'_k\} \text{ avec } \{h_k\} = \{f_i\} \cup \{g_j\} \\ \tau_1 \to \tau_2 \wedge \tau'_1 \to \tau'_2 &= \tau_1 \vee \tau'_1 \to \tau_2 \wedge \tau'_2 \end{aligned}$$

On note que ces définitions récursives sont bien fondées (les types décroissent strictement). Elles sont partielles *i.e.* on échoue dans les cas non couverts par les définitions ci-dessus.

---

**Compilation.** On se propose maintenant de compiler notre petit langage. Une valeur de la forme $\{f_1 = v_1; \ldots; f_n = v_n\}$ est l'adresse d'un bloc mémoire alloué sur le tas, de la forme




où les valeurs $v_1, \ldots, v_n$ des champs $f_1, \ldots, f_n$ sont rangées en mémoire dans l'ordre alphabétique des noms des champs (en supposant donc ici $f_1 < f_2 < \cdots < f_n$). Cette représentation des valeurs pose un problème évident. Dans une application comme

```
(fun (x:{b:int}) -> x.b) {a=1;b=2;c=3}
```

la valeur construite pour le paramètre effectif contient trois champs, et le champ $b$ y est rangé en deuxième position, mais le corps de la fonction a été compilé en supposant que $x$ contient un seul champ, rangé à la première position.

Pour y remédier, on va transformer le programme avant de le compiler, pour modifier toute valeur dont le type « change » à cause du sous-typage. Ainsi, dans l'exemple ci-dessus, on va transformer

la valeur `{a=1;b=2;c=3}` en la valeur `{b=2}` avant de la passer à la fonction. Pour cela, on introduit une fonction $C(e, \tau)$ qui transforme une expression $e$ de type $\tau'$ en une expression de type $\tau$ dès lors que $\tau' \sqsubseteq \tau$. Avec cette fonction, on va modifier le programme $e$ en un programme $T(e)$ de la façon suivante :

$$\begin{array}{rcl} T(e_2\ e_1) & = & T(e_2)\ (C(T(e_1), \tau_1)) \\ T(\texttt{ifzero}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) & = & \texttt{ifzero}\ T(e_1)\ \texttt{then}\ C(T(e_2), \tau_2 \vee \tau_3)\ \texttt{else}\ C(T(e_3), \tau_2 \vee \tau_3) \end{array}$$

où $\tau_1$ d'une part et $\tau_2, \tau_3$ d'autre part sont les types impliqués dans le typage de ces deux constructions. Partout ailleurs, $T$ est un morphisme, c'est-à-dire

$$\begin{array}{rcl} T(n) & = & n \\ T(x) & = & x \\ T(\texttt{fun}\ (x : \tau) \to e) & = & \texttt{fun}\ (x : \tau) \to T(e) \\ T(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & \texttt{let}\ x = T(e_1)\ \texttt{in}\ T(e_2) \\ T(e.f) & = & T(e).f \\ T(\{f_1 = e_1; \ldots; f_n = e_n\}) & = & \{f_1 = T(e_1); \ldots; f_n = T(e_n)\} \end{array}$$

**Question 13** Donner une définition de la fonction $C$.

---

**Correction :** La définition de $C(e, \tau)$ se fait par récurrence sur le type $\tau$.

$$\begin{array}{rcl} C(e, \tau) & = & e \text{ si } \mathit{type}(e) = \tau \\ C(e, \{f_1 : \tau_1; \ldots; f_n : \tau_n\}) & = & \texttt{let}\ v = e\ \texttt{in}\ \{f_{i_1} = C(v.f_{i_1}, \tau_1); \ldots; f_n = C(f_n, \tau_n)\} \\ & & \text{si } \mathit{type}(e) = \{f_1 : \tau'_1; \ldots; g_{i_m} : \tau'_m\} \\ C(e, \tau_1 \to \tau_2) & = & \texttt{fun}\ (x : \tau_1) \to \texttt{let}\ y = C(x, \tau'_1)\ \texttt{in}\ C(e\ y, \tau_2) \\ & & \text{si } \mathit{type}(e) = \tau'_1 \to \tau'_2 \end{array}$$

---

**Compilation vers l'assembleur x86-64.** (Un aide-mémoire est donné en annexe.) On adopte le schéma de compilation suivant. Toutes les valeurs occupent un mot mémoire de 64 bits. Une valeur entière $n$ est directement représentée par un entier 64 bits. Une valeur de la forme $\texttt{fun}\ (x : \tau) \to e$ est l'adresse d'un bloc mémoire alloué sur le tas représentant une fermeture, c'est-à-dire un bloc de $n + 1$ mots de la forme

| code | $v_1$ | ... | $v_n$ |
|---|---|---|---|

où *code* est l'adresse du code correspondant à la fonction et $v_1, \ldots, v_n$ sont les valeurs des variables libres $x_1, \ldots, x_n$ de $\texttt{fun}\ (x : \tau) \to e$, dans un ordre arbitraire choisi par le compilateur. Enfin, une valeur de type enregistrement est l'adresse d'un bloc mémoire contenant les valeurs de ses champs, comme expliqué plus haut. Dans le code d'une fermeture, l'unique argument est passé dans `%rdi`, la fermeture est passée dans `%rsi` et le résultat est renvoyé dans `%rax`.

**Question 14** Expliquer pourquoi il n'y a pas de risque de confondre, à l'exécution,

- un entier avec une adresse ;
- l'adresse d'une fermeture avec l'adresse d'un enregistrement.

---

**Correction :** La clé est la *préservation du typage*, qui nous garantit que les réductions (de la sémantique) n'impliquent que des expressions bien typées. Or, la seule valeur que l'on peut construire dans le type `int` est une constante $n$. De même, dans toute application $v_1\ v_2$, la valeur $v_1$ a un type de fonction. Or, la seule valeur que l'on peut construire dans ce type est une fermeture. Enfin, toute valeur d'un type enregistrement ne peut être construite que par la construction $\{\ldots\}$.

---

**Question 15** On considère un programme de la forme

```
let x = ... in
let y = fun (z: int) -> { a = x; b = z } in
...
```

1. À quoi ressemble la fermeture représentant la valeur de `y` ?
2. Donner un code assembleur pour cette fermeture. (On ne demande pas en revanche le code qui construit la fermeture, ni le code qui l'applique.)

---

**Correction :**

1. Il y a une seule variable libre, `x`. Dès lors, la fermeture a la forme

   ```
   +------+---+
   | code | x |
   +------+---+
   ```

2. Il faut appeler `malloc` pour allouer l'enregistrement, ce qui nécessite de sauvegarder `%rdi` et `%rsi` avant cela.

   ```
   funy: push %rdi     # z
         push 8(%rsi)  # x récupéré dans la fermeture
         mov  $16, %rdi
         call malloc
         pop  %rcx
         mov  %rcx, (%rax)   # a = x
         pop  %rcx
         mov  %rcx, 8(%rax)  # b = z
         ret
   ```

---

**Question 16** On souhaite ajouter une primitive `add` à notre langage, sous la forme d'une fonction prédéfinie de type `int` $\to$ (`int` $\to$ `int`). On peut alors écrire une expression comme `(add 40) 2`, où la première application (`add 40`) construit une fermeture, qui est ensuite appliquée à `2`. La fonction `add` elle-même est représentée par une fermeture dont l'environnement est vide. Donner le code assembleur de ces deux fermetures.

---

**Correction :**

```
add0: push %rdi          # on sauvegarde x
      mov  $16, %rdi     # on alloue la fermeture
      call malloc
      mov  $add1, (%rax)
      pop  %rcx
      mov  %rcx, 8(%rax) # et on y stocke x
      ret
add1: mov 8(%rsi), %rax  # on récupère x dans la fermeture
      add %rdi, %rax     # et on lui ajoute y
      ret
.data
add:  .quad add0         # la fermeture de la fonction add
```

---

**Ajout de références.** On ajoute enfin une dernière chose à notre langage, à savoir des références à la OCaml, c'est-à-dire les trois constructions :

$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & \texttt{ref}\ e & \textit{création} \\ & | & \texttt{!}e & \textit{accès} \\ & | & e\ \texttt{:=}\ e & \textit{affectation} \end{array}$$

La sémantique est la même que celle d'OCaml : une référence est un bloc alloué sur le tas, contenant une unique valeur, à laquelle on accède avec `!` et que l'on modifie avec `:=`. Les références nous permettent notamment de définir des fonctions *récursives*, en utilisant la technique dite du *nœud de Landin* :

1. créer une référence $r$ contenant une fonction arbitraire ;
2. affecter à $r$ une fonction qui utilise $\texttt{!}r$ pour effectuer des appels récursifs.

**Question 17** Donner une expression de type `int → int` qui, appliquée à un entier $n \geq 0$, renvoie le terme $F_n$ de la suite de Fibonacci. (À toutes fins utiles, on rappelle que la suite de Fibonacci $(F_n)$ est définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_n + F_{n+1}$ pour $n \geq 0$.) On pourra utiliser librement la fonction `add` introduite à la question 16 et remarquer qu'une constante $n$ peut être négative.

---

**Correction :** Une solution qui suit la définition récursive de $(F_n)$ :

```
let fib =
  let f = ref (fun n:int -> 42) in
  let _ = f := fun n:int ->
                 ifzero n then 0
                 else ifzero (add n -1) then 1
                 else add (!f (add n -1)) (!f (add n -2)) in
  !f in
...
```

Une autre solution, avec un nombre linéaire d'additions cette fois :

```
let fib =
  let f = ref (fun s:{a:int;b:int;k:int} -> 42) in
```

```
let _ = f := fun s:{a:int;b:int;k:int} ->
               ifzero s.k then s.a
               else !f { a=s.b; b=add s.a s.b; k=add s.k -1} in
fun n:int -> !f {a=0; b=1; k=n} in
```

---

**Typage des références.** On introduit un nouveau constructeur de types pour les références :

$$\begin{array}{rcll} \tau & ::= & \ldots & \\ & | & \texttt{ref}\ \tau & \textit{type d'une référence} \end{array}$$

On se donne les règles de typage suivantes :

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ref}\ e : \texttt{ref}\ \tau} \qquad \frac{\Gamma \vdash e : \texttt{ref}\ \tau}{\Gamma \vdash\ !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{ref}\ \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \sqsubseteq \tau_1}{\Gamma \vdash e_1 := e_2 : \{\}}$$

**Question 18** On pourrait être tenté d'étendre la définition de $\sqsubseteq$ en ajoutant la règle suivante :

$$\frac{\tau' \sqsubseteq \tau}{\texttt{ref}\ \tau' \sqsubseteq \texttt{ref}\ \tau}$$

1. Montrer que cette règle n'est pas sûre.
2. Proposer quelque chose de correct. On ne demande pas de justification.

---

**Correction :**

1. Avec cette règle, on parviendrait à remplacer un enregistrement contenu dans une référence par un enregistrement contenant *moins* de champs :

```
let r = ref {a=0} in                    // r : ref {a}
let y = fun (x:ref {}) -> x := {} in    // y : ref {} -> {}
let _ = y r in                          // OK car r : ref {a} <= ref {}
!r . a                                  // bien typé car r: ref {a}
                                        // mais BOUM car r n'a plus de champ a
```

2. Il suffit de tout simplement *ne pas* ajouter de règle à la définition de $\sqsubseteq$. (On dit que le constructeur `ref` est *invariant*.)

---

**Langage objet.** Avec le langage initial et les références que nous venons d'ajouter, on a maintenant un langage qui permet de modéliser des *objets* au sens de la programmation orientée objets. Ainsi, on peut définir un objet « compteur » $c$ de la façon suivante :

$$\begin{array}{l} \texttt{let}\ c = \\ \quad \texttt{let}\ s = \{n = \texttt{ref}\ 0\}\ \texttt{in} \\ \quad \{\ \ \textit{get} = \texttt{fun}\ (x : \{\}) \to\ !(s.n); \\ \qquad \textit{inc} = \texttt{fun}\ (x : \{\}) \to s.n := (\texttt{add}\ !(s.n))\ 1\ \} \\ \texttt{in}\ \ldots \end{array}$$

(On utilise ici la fonction `add` introduite à la question 16.) Il s'agit d'un enregistrement contenant des *méthodes* (ici *get* et *inc*) agissant sur un *état* interne à l'objet (l'enregistrement $s$, contenant ici un unique attribut $n$). Pour appeler une méthode, il suffit de faire

$$c.\textit{get}\ \{\}$$

c'est-à-dire d'extraire la valeur correspondant à un champ (ici *get*) et de l'appliquer (ici à un enregistrement vide). Voici un exemple plus développé, avec à gauche du code Java et à droite sa traduction dans notre langage :

```
class A {
  int n = 0;
  int  get() { return n; }
  void inc() { n += 1. }
}

class B extends A {
  int i = 42;
  void reset() { n=i; }
}




void inc2(A a) {
  a.inc(); a.inc();
}
... inc2(new B());...
```

```
let methA = fun (s:{n:ref int}) ->
  { get = fun (_:{}) -> !(s.n);
    inc = fun (_:{}) -> s.n := add !(s.n) 1 } in
let newA = fun (_:{}) ->
  let s = { n = ref 0 } in methA s in

let methB = fun (s:{n:ref int; i:ref int}) ->
  let super = methA s in
  { get = super.get;
    inc = super.inc;
    reset = fun (_:{}) -> s.n := !(s.i) } in
let newB = fun (_:{}) ->
  let s = { n = ref 0; i = ref 42 } in methB s in

let inc2 = fun (a:{get:{}->int;inc:{}->{}}) ->
  let _ = a.inc {} in a.inc {} in

... inc2 (newB {}) ...
```

**Question 19** Indiquer à quels endroits du code ci-dessus le sous-typage a été utilisé.

---

**Correction :** Le sous-typage est utilisé
- dans `methB` pour l'application `methA s`, car `s` a un sous-type de `{n:ref int}` ;
- pour l'application `inc2 (newB {})`, car `inc2` attend un objet avec uniquement les méthodes `get` et `inc`, mais on lui passe là un objet avec trois méthodes.

---

**Question 20** On souhaite maintenant faire la même chose pour les deux classes suivantes :

```
class C {
  int n = 0;
  int  get() { inc(); return n; }
  void inc() { n += 1; }
}
```

```
class D extends C {
  void inc() { n += 2; }
}
```

Ce cas de figure est plus subtile que le précédent, car la méthode `get` de la classe `D`, héritée de `C`, doit appeler la méthode `inc` *de la classe `D`* (appel dynamique). Proposer un schéma de traduction.

---

**Correction :** La méthode `get` doit pouvoir appeler une méthode `inc` définie plus loin (possiblement dans une autre classe). De manière générale, les méthodes deviennent mutuellement récursives. On a vu plus haut comme utiliser un nœud de Landin pour définir une fonction récursive et on peut s'en servir ici pour définir l'objet récursivement. Il faut cependant faire attention à la stratégie d'évaluation qui est stricte, et ne construire récursivement qu'une *fonction* qui renvoie l'objet.

Dans le code ci-dessous, on s'autorise à écrire `fix` pour un opérateur de point fixe que l'on aurait défini (au cas par cas) avec un nœud de Landin.

```
let methC = fun (s:{n:ref int}) ->
  fun (self:{}->{get:{}->int;inc:{}->{}}) ->
    fun (_:{}) ->
      { get = fun (_:{}) -> let _ = (self {}).inc {} in !(s.n);
        inc = fun (_:{}) -> s.n := add !(s.n) 1 }
in
let newC = fun (_:{}) ->
  let s = { n = ref 0 } in
  fix (methC s) {}
in
```

On procède de même pour la classe `D`, avec un appel à `methC` pour hériter des méthodes de la classe `C` :

```
let methD = fun (s:{n:ref int}) ->
  fun (self:{}->{get:{}->int;inc:{}->{}}) ->
    fun (_:{}) ->
      let super = methC s self {} in
      { get = super.get;
        inc = fun (_:{}) -> s.n := add !(s.n) 2 }
in
let newD = fun (_:{}) ->
  let s = { n = ref 0 } in
  fix (methD s) {}
in
...
```

---

$$
\begin{array}{rcll}
e & ::= & n & \textit{constante } n \in \mathbb{Z} \\
& | & x & \textit{variable} \\
& | & \texttt{fun}\ (x : \tau) \to e & \textit{fonction} \\
& | & e\ e & \textit{application} \\
& | & \texttt{let}\ x = e\ \texttt{in}\ e & \textit{variable locale} \\
& | & e.f & \textit{accès au champ} \\
& | & \{f = e; \ldots; f = e\} & \textit{création enregistrement} \\
\\
\tau & ::= & \texttt{int} & \textit{type des entiers} \\
& | & \tau \ \to\ \tau & \textit{type des fonctions} \\
& | & \{f : \tau; \ldots; f : \tau\} & \textit{type des enregistrements}
\end{array}
$$

FIGURE 2 – Syntaxe abstraite.

**valeurs**

$$
\begin{array}{rcl}
v & ::= & n \\
& | & \texttt{fun}\ (x : \tau) \to e \\
& | & \{f = v; \ldots; f = v\}
\end{array}
$$

**réductions de tête**

$$
\begin{array}{rcl}
(\texttt{fun}\ (x : \tau) \to e)\ v & \xrightarrow{\epsilon} & e[x \leftarrow v] \\
\texttt{let}\ x = v\ \texttt{in}\ e & \xrightarrow{\epsilon} & e[x \leftarrow v] \\
\{f_1 = v_1; \ldots; f_n = v_n\}.f_i & \xrightarrow{\epsilon} & v_i
\end{array}
$$

**contextes de réduction**

$$
\begin{array}{rcl}
E & ::= & \square \\
& | & E\ e \\
& | & v\ E \\
& | & \texttt{let}\ x = E\ \texttt{in}\ e \\
& | & E.f \\
& | & \{f = v; \ldots; f = v; f = E; f = e; \ldots; f = e\}
\end{array}
$$

**réduction**

$$
\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \to E(e_2)}
$$

FIGURE 3 – Sémantique opérationnelle à petits pas.

$$
\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\texttt{fun}\ (x : \tau_1) \to e) : \tau_1 \to \tau_2}
$$

$$
\frac{\Gamma \vdash e_2 : \tau_1 \ \to\ \tau_2 \quad \Gamma \vdash e_1 : \tau \quad \tau \sqsubseteq \tau_1}{\Gamma \vdash e_2\ e_1 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}
$$

$$
\frac{\Gamma \vdash e : \{\ldots; f : \tau; \ldots\}}{\Gamma \vdash e.f : \tau} \qquad \frac{f_i \text{ distincts} \qquad \forall i,\, \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{f_1 = e_1; \ldots; f_n = e_n\} : \{f_1 : \tau_1; \ldots; f_n : \tau_n\}}
$$

$$
\frac{}{\tau \sqsubseteq \tau} \qquad \frac{\tau_1' \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \tau_2'}{\tau_1 \ \to\ \tau_2 \sqsubseteq \tau_1' \ \to\ \tau_2'} \qquad \frac{\forall 1 \le j \le m,\, \exists 1 \le i \le n,\, f_j' = f_i \text{ et } \tau_i \sqsubseteq \tau_j'}{\{f_1 : \tau_1; \ldots; f_n : \tau_n\} \sqsubseteq \{f_1' : \tau_1; \ldots; f_m' : \tau_m'\}}
$$

FIGURE 4 – Règles de typage.

# Annexe : aide-mémoire x86-64

On donne ici un fragment du jeu d'instructions x86-64. Vous êtes libre d'utiliser tout autre élément de l'assembleur x86-64. Dans ce qui suit, $r_i$ désigne un registre, $n$ une constante entière et $L$ une étiquette.

| | |
|---|---|
| `mov` $r_2$`,` $r_1$ | copie le registre $r_2$ dans le registre $r_1$ |
| `mov` \$$n$`,` $r_1$ | charge la constante $n$ dans le registre $r_1$ |
| `mov` $L$`,` $r_1$ | charge la valeur à l'adresse $L$ dans le registre $r_1$ |
| `mov` \$$L$`,` $r_1$ | charge l'adresse de l'étiquette $L$ dans le registre $r_1$ |
| `add` $r_2$`,` $r_1$ | calcule la somme de $r_1$ et $r_2$ dans $r_1$ (on a de même `sub` et `imul`) |
| `mov` $n$`(`$r_2$`),` $r_1$ | charge dans $r_1$ la valeur contenue en mémoire à l'adresse $r_2 + n$ |
| `mov` $r_1$`,` $n$`(`$r_2$`)` | écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans $r_1$ |
| `push` $r_1$ | empile la valeur contenue dans $r_1$ |
| `pop` $r_1$ | dépile une valeur dans le registre $r_1$ |
| `cmp` $r_2$`,` $r_1$ | positionne les drapeaux en fonction de la valeur de $r_1 - r_2$ |
| `test` $r_2$`,` $r_1$ | positionne les drapeaux en fonction de la valeur de $r_1 \& r_2$ |
| `je` $L$ | saute à l'adresse désignée par l'étiquette $L$ en cas d'égalité (on a de même `jne`, `jg`, `jge`, `jl` et `jle`) |
| `jmp` $L$ | saute à l'adresse désignée par l'étiquette $L$ |
| `call` $L$ | saute à l'adresse désignée par l'étiquette $L$, après avoir empilé l'adresse de retour |
| `ret` | dépile une adresse et y effectue un saut |

On alloue de la mémoire sur le tas avec un appel à `malloc`, qui attend un nombre d'octets dans `%rdi` et renvoie l'adresse du bloc alloué dans `%rax`.