École Normale Supérieure

Langages de programmation et compilation

examen 2019–2020

Jean-Christophe Filliâtre

24 janvier 2020

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.
L'épreuve dure 3 heures.

Dans tout ce sujet, on considère un petit langage de programmation qui est une variante du langage mini-ML étudié en cours. La syntaxe abstraite de ce langage est donnée figure 1. Cette syntaxe distingue des expressions déjà évaluées, dites *atomiques* et notées a, et des expressions arbitraires représentant des calculs (potentiels), notées e. Dans ce langage, les valeurs sont réduites à des entiers relatifs, notés n, et des fonctions récursives. Une fonction récursive est notée `rec` f x `=` e où f est une variable dénotant la fonction et x une variable dénotant son argument.

Une sémantique opérationnelle à petits pas est donnée pour ce langage, figure 2, sous la forme d'un jugement $e \to e'$ défini par des règles d'inférence.

Question 1 Donner la suite des réductions pour chacune des trois expressions suivantes :

1. `let double = rec _ n = add n n in double 21`
2. `let f = rec f n = f n in f 0`
3. `let f = rec _ n = ifz x then 34 else 55 in let x = 0 in f x`

Correction : Pour la première expression, on obtient le résultat attendu :

```
    let double = rec _ n = add n n in double 21
--> (rec _ n = add n n) 21
--> add 21 21
--> 42
```

Pour la deuxième expression, le calcul ne termine pas, avec la réduction infinie suivante :

```
    let f = rec f n = f n in f 0
--> (rec f n = f n) 0
--> (rec f n = f n) 0
--> ...
```

Enfin, la troisième expression contient une variable libre `x`. On parvient cependant à une valeur mais seulement à cause d'une capture de variable :

```
    let f = rec _ n = ifz x then 34 else 55 in let x = 0 in f x
--> let x = 0 in (rec _ n = ifz x then 34 else 55) x
--> (rec _ n = ifz 0 then 34 else 55) 0
--> ifz 0 then 34 else 55
--> 34
```

Si on renomme la variable `x` liée par le `let`, alors le calcul bloquera sur une expression `ifz x ...` irréductible.

Question 2 Donner une expression e telle que, pour tout entier naturel n, l'évaluation de l'expression `let` $f = e$ `in` $f\ n$ aboutisse à la valeur du n-ième terme de la suite de Fibonacci, c'est-à-dire `let` $f = e$ `in` $f\ n \to^\star F_n$ avec F_n ainsi défini :

$$\begin{cases} F_0 & = & 0 \\ F_1 & = & 1 \\ F_{n+2} & = & F_n + F_{n+1} \quad \text{pour } n \geq 0. \end{cases}$$

La complexité de cette évaluation, définie comme le nombre total de petits pas, doit être $O(n)$.

Correction :

```
let fib = rec fib a = rec _ b = rec _ n =
          ifz n then a else let fibb = fib b in
                            let next = add a b in
                            let fib2 = fibb next in
                            let n_1 = add n -1 in
                            fib2 n_1 in
let fib = fib 0 in
fib 1
```

Analyse syntaxique. On souhaite réaliser l'analyse syntaxique de notre petit langage avec l'outil Menhir. La figure 3 contient un fichier d'entrée pour Menhir contenant la grammaire de notre langage. Les actions sémantiques ne nous intéressent pas ici et sont omises (`{...}`).

Question 3 Donner les valeurs de NULL, FIRST et FOLLOW pour les deux non-terminaux `expr` et `atom`.

Correction : On trouve par calcul de point fixe :

$$\text{NULL}(\texttt{atom}) = \text{NULL}(\texttt{expr}) = \texttt{false}$$

puis

$$\begin{array}{rcl} \text{FIRST}(\texttt{atom}) & = & \{\, \texttt{IDENT}, \texttt{CONST}, \texttt{REC} \,\} \\ \text{FIRST}(\texttt{expr}) & = & \{\, \texttt{IDENT}, \texttt{CONST}, \texttt{REC}, \texttt{ADD}, \texttt{LET}, \texttt{IFZ} \,\} \end{array}$$

et enfin

$$\begin{array}{rcl} \text{FOLLOW}(\texttt{atom}) & = & \{\, \texttt{IDENT}, \texttt{CONST}, \texttt{REC}, \texttt{THEN}, \texttt{IN}, \texttt{ELSE}, \texttt{EOF} \,\} \\ \text{FOLLOW}(\texttt{expr}) & = & \{\, \texttt{IDENT}, \texttt{CONST}, \texttt{REC}, \texttt{THEN}, \texttt{IN}, \texttt{ELSE}, \texttt{EOF} \,\} \end{array}$$

Question 4 Lorsque l'outil Menhir est lancé sur le fichier donné en figure 3, il déclare trois conflits de type lecture/réduction (`shift/reduce`). Les identifier, les expliquer, faire le choix de favoriser lecture ou réduction et modifier en conséquence le fichier Menhir.

Correction : Les trois conflits apparaissent dans le même état, à savoir

```
expr --> atom .
         atom . atom
```

Ils se produisent à la lecture de `IDENT`, `CONST` ou `REC`. Dans les trois cas, la lecture est possible (ces trois terminaux sont dans FIRST(`atom`) et la réduction également (ces trois terminaux sont également dans FOLLOW(`atom`)). Ces trois conflits traduisent l'ambiguïté entre

```
(rec x x = a) a
```

et

```
rec x x = (a a)
```

(on a ajouté ici des parenthèses pour montrer la différence entre les deux).

Si on veut favoriser la lecture, c'est-à-dire la seconde forme, il faut donner une priorité plus grande aux trois terminaux `IDENT`, `CONST` et `REC` qu'à la règle `expr --> atom atom`. On peut le faire ainsi avec Menhir :

```
...
%nonassoc prec_atom
%nonassoc REC IDENT CONST
...
expr:
  | a=atom %prec prec_atom {...}
...
```

Le reste est inchangé.

Sucre syntaxique. Telle quelle, la grammaire de notre langage n'est pas très agréable, surtout lorsqu'on souhaite définir ou utiliser des fonctions à plusieurs arguments. Ainsi, pour définir une fonction à deux arguments, on doit écrire `rec` f x_1 `=` `rec` _ x_2 `=` e et pour l'appliquer à deux arguments on doit écrire `let` $g = f\ a_1$ `in` $g\ a_2$.

Question 5 Proposer une modification de la syntaxe *concrète* pour autoriser une définition de fonction de la forme `rec` $f\ x_1\ \ldots\ x_n$ `=` e, c'est-à-dire avec $n \geq 1$ paramètres formels, ainsi qu'une application de la forme $a_0\ a_1\ a_2\ \ldots\ a_n$, c'est-à-dire avec $n \geq 1$ paramètres effectifs. Indiquer les modifications à apporter à la grammaire Menhir, ainsi que les actions sémantiques des règles qui ont été modifiées. On suppose que la syntaxe abstraite OCaml est de la forme suivante :

```
type atom = Var of string | Rec of string * string * expr | ...
and  expr = Atom of atom | App of atom * atom | Let of string * expr * expr | ...
```

On prendra soin de construire des arbres de syntaxe abstraite bien typés, en respectant la différence entre les types `atom` et `expr`.

Correction : Pour l'application, l'idée est de transformer $a_1\ a_2\ \ldots\ a_n$ en

$$\texttt{let \$} = (\texttt{let \$} = (\ldots)\ \texttt{in \$}\ a_{n-1})\ \texttt{in \$}\ a_n$$

où \$ est une variable fraîche (par exemple en dehors des conventions lexicales du langage). Une seule variable suffit, car il n'y a pas de capture possible, la variable \$ ne pouvant apparaître dans les atomes a_i. Il suffit de se donner une petite fonction telle que

```
let mkapp e a = Let ("$", e, App (Var "$", a))
```

et de l'appliquer récursivement sur toute la liste, par exemple avec `List.fold_left` :

```
expr:
  | al=atoms { match al with
               | [a] -> Atom a
               | a1 :: a2 :: al -> List.fold_left mkapp (App (a1, a2)) al
               | [] -> assert false }
```

Ici, la liste d'atomes est reconnue avec un nouveau non-terminal `atoms`, où on a pris soin de conserver l'indication de priorité (réponse à la question précédente) :

```
atoms:
  | a=atom  %prec prec_atom { [a]     }
  | a=atom al=atoms         { a :: al }
```

En ce qui concerne la définition d'une fonction, on transforme `rec` $f\ x_1 \ldots x_n$ `=` e en

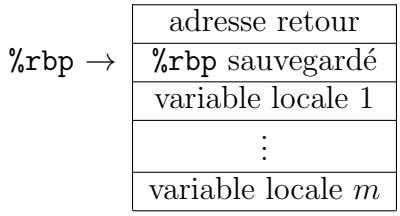
$$\texttt{rec}\ f\ x_1\ \texttt{=}\ (\texttt{rec}\ \_\ x_2\ \texttt{=}\ (\ldots \texttt{rec}\ \_\ x_n\ \texttt{=}\ e))$$

On modifie la grammaire pour reconnaître une liste d'identificateurs (au lieu d'un seul) puis on applique la transformation ci-dessus, par exemple avec `List.fold_right` :

```
  | REC f=IDENT xs=IDENT+ EQUAL e=expr
    { match xs with
      | x :: xs ->
          Rec (f, x, List.fold_right (fun x e -> Atom (Rec ("_", x, e))) xs e)
      | [] -> assert false }
```

Compilation vers x86-64. On se propose maintenant de compiler notre petit langage vers l'assembleur x86-64. (Un aide-mémoire est donné en annexe.) La compilation d'une expression e est un code assembleur noté $C(e)$ dont l'exécution a pour effet de stocker la valeur de l'expression e dans le registre `%rax`.

Notre langage étant un langage fonctionnel par nature, sa compilation met en œuvre des fermetures, comme expliqué en cours. On adopte le schéma de compilation suivant. Une valeur est soit un entier 64 bits signé, soit un pointeur vers un bloc alloué sur le tas contenant une fermeture. Les valeurs ont donc toutes la même taille (64 bits). Pour toute expression `rec` $f\ x$ `=` e contenue dans le programme, on introduit une fonction assembleur f, recevant l'argument x dans le registre `%rdi` et la fermeture dans le registre `%rsi`. Le tableau d'activation de la fonction a la forme ci-contre, les adresses croissant vers le haut. Dans ce tableau, on trouve les m variables locales à l'expression e. Les



$$
\begin{array}{rcll}
a & ::= & x & \textit{variable} \\
& | & n & \textit{constante } (n \in \mathbb{Z}) \\
& | & \texttt{rec } x\ x\ \texttt{=}\ e & \textit{fonction récursive} \\
\\
e & ::= & a & \textit{atome} \\
& | & a\ a & \textit{application} \\
& | & \texttt{add } a\ a & \textit{addition} \\
& | & \texttt{let } x = e\ \texttt{in } e & \textit{variable locale} \\
& | & \texttt{ifz } a\ \texttt{then } e\ \texttt{else } e & \textit{conditionnelle}
\end{array}
$$

FIGURE 1 – Syntaxe abstraite.

$$
\frac{}{(\texttt{rec } f\ x\ \texttt{=}\ e)\ a \to e[x \leftarrow a][f \leftarrow \texttt{rec } f\ x\ \texttt{=}\ e]} \qquad \frac{n = n_1 + n_2}{\texttt{add } n_1\ n_2 \to n}
$$

$$
\frac{}{\texttt{let } x = a\ \texttt{in } e_2 \to e_2[x \leftarrow a]} \qquad \frac{e_1 \to e_1'}{\texttt{let } x = e_1\ \texttt{in } e_2 \to \texttt{let } x = e_1'\ \texttt{in } e_2}
$$

$$
\frac{}{\texttt{ifz } 0\ \texttt{then } e_1\ \texttt{else } e_2 \to e_1} \qquad \frac{n \neq 0}{\texttt{ifz } n\ \texttt{then } e_1\ \texttt{else } e_2 \to e_2}
$$

FIGURE 2 – Sémantique opérationnelle à petits pas.

```
%token <string> IDENT
%token <int> CONST
%token LET IN REC IFZ THEN ELSE ADD EQUAL EOF
%start one_expr
%type  <unit> one_expr
%%
one_expr:
  | expr EOF                      {...}
expr:
  | atom                          {...}
  | atom atom                     {...}
  | ADD atom atom                 {...}
  | LET IDENT EQUAL expr IN expr  {...}
  | IFZ atom THEN expr ELSE expr  {...}
atom:
  | IDENT                         {...}
  | CONST                         {...}
  | REC IDENT IDENT EQUAL expr    {...}
```

FIGURE 3 – Fichier Menhir.

variables libres de e, en revanche, se trouvent dans la fermeture, à l'exception de la variable x qui se trouve dans `%rdi`. On note que si la fonction f est effectivement récursive, f apparaît dans les variables libres de e et peut donc être retrouvée dans la fermeture, comme toute autre variable libre de e. Le résultat de la fonction (c'est-à-dire la valeur de l'expression e constituant le corps de la fonction) est placé dans le registre `%rax`.

Question 6 On considère l'exécution du code assembleur obtenu pour le programme suivant :

```
let f = rec f x = ifz x then 42 else let x = add x -1 in f x in f 1729
```

Indiquer combien de fermetures seront allouées sur le tas pendant cette exécution.

Correction : Une seule. En effet, l'évaluation d'une seule construction `rec` est rencontrée pendant l'exécution, pour initialiser la variable (locale) `f`.

Question 7 Donner un code assembleur pour la fonction `f` résultant de la compilation de l'expression

```
rec f x = ifz x then 42 else let x = add x -1 in f x
```

Indiquer si l'appel à `f` est terminal. Le cas échéant, l'optimiser.

Correction :

```
f: testq %rdi, %rdi
   jnz   1f
   mov   $42, %rax
   ret
1: decq  %rdi
   jmp   f    # appel terminal
```

L'appel à `f` est terminal et optimisé par un simple saut. En toute généralité, on chercherait la valeur de `f` dans la fermeture (ici, ce serait `8(%rsi)` car il y a une seule variable capturée dans la fermeture, à savoir `f` elle-même), pour ensuite effectuer l'appel, c'est-à-dire

```
movq 8(%rsi), %rsi
jmp  *(%rsi) # appel terminal optimisé
```

Mais dans ce cas précis on sait qu'on fait un appel récursif.

Question 8 Donner la définition de $C(e)$ pour chacune des constructions du langage.

Correction :

```
C(n) = mov $n, %rax
C(x) = mov %rdi, %rax    # si x = argument
C(x) = mov o(%rbp), %rax # si x = variable locale
C(x) = mov o(%rsi), %rax # si x dans la fermeture
```

```
C(rec f x = e) = soit {y1,...,yn} = fv(e)\{x}
  mov $8(n+1), %rdi
  call malloc
  mov  %rax, %rcx
  mov  $f, (%rcx)
  C(y1) mov %rax, 8(%rcx) ... C(yn) mov %rax, 8n(%rcx)
  mov %rcx, %rax
C(let x = e1 in e2) =
  C(e1)
  mov %rax, o(%rbp)
  C(e2)
C(a1 a2)
  C(a1)
  mov %rax %rdx
  C(a2)
  mov %rax %rdi
  mov %rdx %rsi
  call *(%rsi)
C(add a1 a2)
  C(a1)
  mov %rax %rdx
  C(a2)
  add %rdx %rax
C(ifz a then e1 else e2) =
  C(a)
  testq %rax, %rax
  jnz 1f
  C(e1)
  jmp 2f
1:C(e2)
2:
```

Note : la compilation d'un atome (resp. expression) utilise le registre `%rcx` (resp. `%rdx`) comme temporaire.

Effets algébriques. On augmente maintenant notre petit langage avec de nouvelles constructions, appelées *effets algébriques*, dont la syntaxe est donnée figure 4. Un effet, noté E, est un simple identifiant. La construction `let` x `=` `do` E a `in` e déclenche l'effet E, en lui associant la valeur a, et lie la variable x dans l'expression e. La construction `handle` e `with` h évalue l'expression e, en traitant les effets résultants, le cas échéant, à l'aide du gestionnaire h. Ce dernier contient des quadruplets de la forme E v k `=>` e', où les variables v et k sont liées dans l'expression e'.

Des règles de sémantique opérationnelle pour ces nouvelles constructions sont données figure 5, où la notation $E \notin h$ signifie qu'il n'existe aucun quadruplet pour E dans le gestionnaire h. Ces nouvelles règles s'ajoutent aux règles précédentes (de la figure 2), qui sont inchangées. La sémantique de ces deux constructions est assez subtile et on prendra le temps de bien lire et de bien comprendre ces nouvelles constructions. En particulier, on notera comment, dans un gestionnaire E x k `=>` e, la fonction k permet de *reprendre* le calcul à l'endroit où l'effet a été déclenché, la valeur passée en argument à k se retrouvant liée à la variable y introduite par la construction `let` y `=` `do` à l'endroit

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{let}\ x = \texttt{do}\ E\ a\ \texttt{in}\ e & \textit{déclenchement} \\
 & | & \texttt{handle}\ e\ \texttt{with}\ h & \textit{traitement} \\
h & ::= & \{\, E\ x\ x\ \texttt{=>}\ e\texttt{;}\ \ldots\texttt{;}\, E\ x\ x\ \texttt{=>}\ e\,\} & \textit{gestionnaire}
\end{array}
$$

FIGURE 4 – Syntaxe abstraite augmentée.

$$
\frac{}{\texttt{handle}\ a\ \texttt{with}\ h \to a} \qquad \frac{e_1 \to e_2}{\texttt{handle}\ e_1\ \texttt{with}\ h \to \texttt{handle}\ e_2\ \texttt{with}\ h}
$$

$$
\frac{\text{(on suppose que } y \text{ n'est pas libre dans } e_2\text{, quitte à la renommer)}}{\texttt{let}\ x = \texttt{let}\ y = \texttt{do}\ E\ a\ \texttt{in}\ e_1\ \texttt{in}\ e_2 \to \texttt{let}\ y = \texttt{do}\ E\ a\ \texttt{in let}\ x = e_1\ \texttt{in}\ e_2}
$$

$$
\frac{E \notin h}{\texttt{handle let}\ y = \texttt{do}\ E\ a\ \texttt{in}\ e_2\ \texttt{with}\ h \to \texttt{let}\ y = \texttt{do}\ E\ a\ \texttt{in handle}\ e_2\ \texttt{with}\ h}
$$

$$
\frac{(E\ x\ k\ \texttt{=>}\ e_1) \in h}{\texttt{handle let}\ y = \texttt{do}\ E\ a\ \texttt{in}\ e_2\ \texttt{with}\ h \to e_1[x \leftarrow a][k \leftarrow \texttt{rec}\ \_\ y\ \texttt{=}\ \texttt{handle}\ e_2\ \texttt{with}\ h]}
$$

FIGURE 5 – Sémantique opérationnelle à petits pas (nouvelles règles).

du déclenchement.

Question 9 Donner la suite des réductions pour chacune des deux expressions suivantes :

1. `handle let x = do Stop 41 in x with { Stop _ k => k 42 }`
2. `handle let x = do E 21 in add x x with { E n k => let n = k n in k n }`

Correction :

1.
```
    handle let x = do Stop 41 in x with { Stop _ k => k 42 }
--> (rec _ x = handle x with ...) 42
--> handle 42 with ...
--> 42
```

2.
```
    handle let x = do E 21 in add x x with { E n k => let n = k n in k n }
--> let n = (rec _ x = handle add x x with ...) 21 in
    (rec _ x = handle add x x with ...) n
--> let n = handle add 21 21 with ... in
    (rec _ x = handle add x x with ...) n
--> let n = handle 42 with ... in
    (rec _ x = handle add x x with ...) n
--> let n = 42 in
    (rec _ x = handle add x x with ...) n
--> (rec _ x = handle add x x with ...) 42
--> handle add 42 42 with ...
```

```
--> handle 84 with ...
--> 84
```

Question 10 Compléter l'expression suivante

```
handle
   let _ = do Write n1 in let x = do Read 0 in let x = add x n2 in
   let _ = do Write x in let y = do Read 0 in y
with { ... }
```

avec un gestionnaire (la partie indiquée `...`) de telle façon que l'évaluation de l'expression donne toujours l'entier $n_1 + n_2$. Le gestionnaire ne doit pas dépendre des constantes n_1 et n_2.

Correction :

```
Write v k => handle k 0 with { Read _ k => k v }
```

On note que le gestionnaire de `Write` est bien réinstallé autour de la continuation, ce qui permet d'utiliser `Write`/`Read` autant de fois que possible.
Pour faire encore mieux, on peut installer un gestionnaire pour `Read` à côté de celui pour `Write`, dans le cas où le premier `Read` serait déclenché avant le premier `Write`, et par exemple signaler une erreur (avec encore un autre effet si on le souhaite !).

Question 11 Expliquer comment encoder la notion usuelle d'exceptions à l'aide d'effets algébriques.

Correction : Une exception E devient un effet E. Pour la levée de l'exception, il suffit d'utiliser une continuation arbitraire (elle ne sera pas utilisée) :

```
    raise (E v)
--> let _ = do E v in 0
```

Pour la gestion des exceptions, il suffit d'ignorer les continuations :

```
    try e with E1 x1 -> e1 | ... | En xn -> en
--> handle e with { E1 x1 _ => e1; ... ; En xn _ => en }
```

Typage. On s'intéresse maintenant au typage de notre langage, avec pour objectif d'obtenir la sûreté du typage, *i.e.*, l'évaluation d'une expression bien typée aboutit à une valeur (constante entière ou fonction) ou ne termine pas. En particulier, elle ne doit pas aboutir à un déclenchement d'effet qui ne serait pas traité par un gestionnaire.

On se donne des types τ pour les valeurs et des types κ pour les calculs, définis de la manière suivante :

$$\begin{array}{rcll} \tau & ::= & \texttt{int} & \textit{type d'un entier} \\ & | & \tau \to \kappa & \textit{type d'une fonction} \\ \kappa & ::= & \tau\langle S\rangle & \textit{type d'un calcul} \\ S & ::= & \{E, \ldots, E\} & \textit{ensemble fini d'effets} \end{array}$$

Un type de calcul $\tau\langle S\rangle$ est une paire formée d'un type de valeur (le type de la valeur renvoyée par ce calcul) et d'un ensemble d'effets *susceptibles* d'être déclenchés par ce calcul. Un environnement de typage Γ donne un type de valeur τ à toute variable apparaissant dans le programme. On a deux jugements de typage : un jugement $\Gamma \vdash a : \tau$ pour les atomes et un jugement $\Gamma \vdash e : \kappa$ pour les expressions. Les règles de typage sont données figure 6. Par simplicité, on suppose que tous les effets attendent un argument de type `int` et "renvoient" une valeur de type `int`.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma, f : \tau \ \to \ \kappa, x : \tau \vdash e : \kappa}{\Gamma \vdash \texttt{rec}\ f\ x = e : \tau \ \to \ \kappa}$$

$$\frac{\Gamma \vdash a : \tau}{\Gamma \vdash a : \tau\langle S\rangle} \qquad \frac{\Gamma \vdash a_1 : \tau \ \to \ \kappa \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash a_1\ a_2 : \kappa} \qquad \frac{\Gamma \vdash a_1 : \texttt{int} \quad \Gamma \vdash a_2 : \texttt{int}}{\Gamma \vdash \texttt{add}\ a_1\ a_2 : \texttt{int}\langle S\rangle}$$

$$\frac{\Gamma \vdash e_1 : \tau_1\langle S\rangle \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2\langle S\rangle}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2\langle S\rangle} \qquad \frac{\Gamma \vdash a : \texttt{int} \quad \Gamma \vdash e_1 : \kappa \quad \Gamma \vdash e_2 : \kappa}{\Gamma \vdash \texttt{ifz}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \kappa}$$

$$\frac{\Gamma \vdash a : \texttt{int} \quad \Gamma, x : \texttt{int} \vdash e : \tau\langle S\rangle \quad E \in S}{\Gamma \vdash \texttt{let}\ x = \texttt{do}\ E\ a\ \texttt{in}\ e : \tau\langle S\rangle}$$

$$\frac{\Gamma \vdash e : \tau\langle S\rangle \quad \forall i. \Gamma, x_i^1 : \texttt{int}, x_i^2 : \texttt{int} \ \to \ \tau\langle S'\rangle \vdash e_i : \tau\langle S'\rangle \quad S \backslash \{ E_1, \ldots, E_n \} \subseteq S'}{\Gamma \vdash \texttt{handle}\ e\ \texttt{with}\ \{\, E_1\ x_1^1\ x_1^2\ \texttt{=>}\ e_1\texttt{;}\ \ldots\ \texttt{;} E_n\ x_n^1\ x_n^2\ \texttt{=>}\ e_n\,\} : \tau\langle S'\rangle}$$

FIGURE 6 – Règles de typage.

Question 12 Pour chacune des trois expressions suivantes, indiquer si elle est bien typée (dans un environnement vide) :

1. `let x = do E 42 in x`
2. `rec f x = f x`
3. `handle 1 with { A x k => let _ = do B 2 in 3 }`

Le cas échéant, donner un type possible et une dérivation de typage. Dans le cas contraire, justifier.

Correction : Les trois expressions sont bien typées :

```
          |- 42:int   x:int |- x:int<E>
        ---------------------------------
         |- let x = do E 42 in x : int<E>


  ... |- f:int -> int<>    ... |- x:int
 ----------------------------------------
   f: int -> int<>, x:int |- f x:int<>
 ----------------------------------------
       |- rec f x = f x : int -> int<>
```

En posant `h` $\stackrel{\text{def}}{=}$ `{ A x k => let _ = do B 2 in 3 }`, on a

```
                                      ...,_:int|-3:int
                                    --------------------------
                      ...|-2:int      ...,_:int|-3:int<B>
                   ---------------------------------------------------
  |- 1:int    x:int, k:int->int<B> |- let_=do B 2 in 3 : int<B>
 ------------------------------------------------------------------
     |- handle 1 with h : int<B>
```

c'est-à-dire qu'on a pris `S'` $= \{\, \texttt{B} \,\}$ et `S` $= \emptyset$ pour la dernière règle.

Question 13 Montrer la propriété de progrès : pour une expression e *close*, si $\emptyset \vdash e : \tau\langle\emptyset\rangle$, alors e est une valeur (un atome qui n'est pas une variable) ou il existe une expression e' telle que $e \to e'$.

Correction : On montre un résultat plus général : si $\emptyset \vdash e : \tau\langle S\rangle$ et que e n'est pas de la forme `let` $x =$ `do` E a `in` e_0 alors e est une valeur (un atome qui n'est pas une variable) ou il existe une expression e' telle que $e \to e'$.

On procède par récurrence structurelle sur la dérivation de typage et par cas sur la dernière règle utilisée.

```
e = x
  impossible car e est close
e = n
  c'est une valeur
e = rec _
  c'est une valeur
e = a1 a2
  on a a1:tau->kappa donc a1 = rec _ et il y a réduction
e = add a1 a2
  on a a1:int et a2:int donc a1 et a2 ne peuvent être que des
  entiers et il y a donc réduction
e = let x = e1 in e2
  par HR e1 est une valeur ou se réduit
  dans les deux cas il y a réduction
e = ifz a then e1 else e2
  on a a:int donc a est un entier, et il y a donc réduction
e = let x = do E a in e'
  impossible par hypothèse
e = handle e1 with h
  si e1 est de la forme let x = do ... alors il y a réduction
  sinon, par HR e est une valeur ou se réduit
  dans les deux cas il y a réduction
```

Le résultat voulu s'en déduit trivialement, car $\emptyset \vdash e : \tau\langle\emptyset\rangle$ ne permet pas à e d'être de la forme `let` $x =$ `do` E a `in` e_0.

Question 14 Montrer la propriété de préservation du typage : si $\Gamma \vdash e : \kappa$ et $e \to e'$ alors $\Gamma \vdash e' : \kappa$. On admettra la propriété de préservation du typage par substitution : si $\Gamma, x : \tau \vdash e : \kappa$ et $\Gamma \vdash a : \tau$, alors $\Gamma \vdash e[x \leftarrow a] : \kappa$.

Correction : On procède par récurrence structurelle sur la dérivation de typage et par cas sur la dernière règle utilisée.

```
e = a
  impossible
e = a1 a2
  on a donc a1=(rec f x = e1)
  donc G,f:tau->kappa,x:tau|-e1:tau->kappa et G|-a2:tau
  et le résultat se déduit de la préservation du typage par substitution
```

```
e = add a1 a2
  on a a1:int et a2:int et une réduction vers a1+a2, de type int
e = let x = e1 in e2
  si e1->e'1 alors par HR e'1 conserve le même type et let x=e'1 in e2 aussi
  si e1 est une valeur alors le résultat se déduit de la
    préservation du typage par substitution
e = ifz a then e1 else e2
  on a a:int et donc une réduction vers e1 ou e2, du même type
e = let x = do E a in e'
  impossible (ne se réduit pas)
e = handle e1 with h
  si e1->e'1 alors par HR e'1 conserve le même type et handle e'1... aussi

  si e = handle a with ... -> a
     |- a : tau<S> pour n'importe quel S, donc celui de e en particulier

  si e = handle let y = do E a in e2 with h
      -> let y = do E a in handle e2 with h  (E n'est pas dans h)

     on a la dérivation
         |- a:int  y:int |- e2:tau<S>  E in S
         ------------------------------------
           |- let y = do E a in e2 : tau<S>        S\{E1...En} dans S'
         ------------------------------------------------------------
            |- handle let ... with h : tau<S'>

    or E est dans S' car E n'est pas dans h, d'où la dérivation

                        y:int |- e2:tau<S> ...  S\{E1...En} dans S'
                       ------------------------------------------
         |- a:int       y:int |- handle e2 with h:tau<S'>            E in S
         ------------------------------------------------------------------
            |- let y= do E a in handle e2 with h : tau<S'>

  si e = handle let y = do E a in e2 with h
      -> eh[x<-a][k<-rec _y=handle e2 with h]  (E x k => eh est dans h)

      alors ... |- eh : tau<S'> et le résultat s'en déduit par substitution
```

Question 15 En déduire la propriété de sûreté du typage, que l'on énoncera précisément.

Correction : Si $\emptyset \vdash e : \tau\langle\emptyset\rangle$ et si $e \to^\star e'$ avec e' irréductible, alors e' est une valeur.
Par induction, la préservation du typage nous donne que e' est bien typée, de type $\tau\langle\emptyset\rangle$. Dès lors, la propriété de progrès nous dit que e' est une valeur, car c'est un atome qui ne peut être une variable car e' est close (typée dans l'environnement vide).

Compilation. On se pose enfin la question de la compilation de notre langage avec effets algébriques. On se propose de le traduire vers le langage OCaml. L'idée est d'utiliser une exception OCaml pour réaliser le déclenchement d'un effet. Cette exception prend en argument le nom de l'effet (de type `string`), la valeur passée en argument à l'effet, ainsi que la suite du calcul sous la forme d'une fonction :

```
exception Do of string * int * (int -> int)
```

Pour simplifier les choses, on suppose ici que la suite du calcul (l'expression e dans la construction `let` $x =$ `do` E a `in` e) est toujours de type `int`.

Question 16 Donner le schéma de la traduction des constructions `let` $x =$ `do` E a `in` e et `handle` e `with` h.

Correction : En notant `T(e)` la traduction de l'expression `e`, on pose

```
T(let x = do E a in e) =
  raise (Do (E, a, (fun x -> T(e))))
```

et

```
T(handle e with { E1 x1 k1 => e1 | ... }) =
    let rec handle f x = try f x with
      | Do (E1, x1, k1) -> let k1 x1 = handle k1 x1 in T(e1)
      | ... in
    handle (fun _ -> T(e)) 0
```

La dernière valeur 0 n'est pas significative.

En réalité, c'est un peu plus compliqué que cela. En effet, on peut se retrouver avec une construction `do` à l'intérieur d'une construction `let`, c'est-à-dire

```
let x =
  let y = do E a in e1
in e2
```

et le calcul de `e2` ne doit pas être perdu, c'est-à-dire qu'il doit être intégré à la continuation passé en argument de `raise` lorsque `do` sera rencontré. Il faudrait donc adopter un style de programmation par continuation (CPS) pour l'ensemble de la traduction, afin de toujours disposer de la continuation courante.

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

Énoncé en français.