Consignes générales

L'examen dure deux heures. Tous les documents sont autorisés. Tout matériel électronique doté d'une fonction de communication doit être silencieux et rangé hors de vue.

1 Introduction

Le but de cet examen est d'étudier $\mathsf{Fun}_{\mathsf{ex}}$, un petit langage de programmation équipé de l'addition, de fonctions, et d'exceptions. Chacun des trois exercices concerne l'une des facettes de ce langage : son évaluation, son typage, et enfin sa compilation. Le sujet est **volontairement trop long** : il suffit d'avoir fait deux exercices parfaitement pour avoir 20/20 ; vous pouvez commencer un exercice sans avoir terminé le précédent. Les exercices sont indépendants les uns des autres, mais il est recommandé d'avoir lu l'intégralité du sujet avant de commencer à répondre.

Le langage $\mathsf{Fun}_{\mathsf{ex}}$ peut être vu comme un petit sous-ensemble d'Hopix auquel on aurait ajouté des constructions pour lancer et rattraper des exceptions similaires à celles qu'on peut trouver en OCaml, Java ou C++. Les exceptions sont utiles pour interrompre le flot normal de l'exécution, souvent — mais pas toujours — pour signaler une erreur.

La syntaxe abstraite des expressions de $\mathsf{Fun}_{\mathsf{ex}}$ est décrite par la grammaire suivante.

$e, e' ::=$	Expressions
$\mid x$	Variable
$\mid \underline{n}$	Constante entière
$\mid e \mathbin{\underline{+}} e'$	Addition
$\mid \mathsf{fun}\, x.\, e$	Fonction anonyme
$\mid e\ e'$	Application de fonction
$\mid \mathsf{raise}\, K$	Levée d'exception
$\mid \mathsf{try}\, e\, \mathsf{with}\, K \Rightarrow e'$	Rattrapage d'exception

Vous avez déjà rencontré les cinq premières en cours, nous allons donc nous attarder sur les deux dernières, qui manipulent les exceptions. Elles fonctionnent à la manière des constructions similaires en OCaml :

— la construction $\mathsf{raise}\, K$ interrompt l'exécution en levant l'exception K.

— la construction $\mathsf{try}\, e\, \mathsf{with}\, K \Rightarrow e'$ se comporte comme e, sauf si e lève l'exception K, auquel cas elle évalue e'.

La métavariable K désigne les constructeurs d'exceptions. Contrairement à ceux d'OCaml, ceux-ci n'ont pas à être déclarés, et ne peuvent recevoir de paramètres. Dans une syntaxe concrète, un constructeur d'exception serait simplement un identifiant commençant par une majuscule, comme par exemple Error, Foo ou Bar, à la manière des constructeurs de types algébriques d'Hopix ou OCaml.

2 Évaluation (10 points)

Le but de cet exercice est d'écrire une sémantique pour $\mathsf{Fun}_{\mathsf{ex}}$. On va suivre la méthode vue en cours, en définissant par un ensemble de règles le jugement $e; \sigma \Downarrow V$, qui spécifie que l'expression e s'évalue en la valeur V dans l'environnement σ. Les valeurs sont décrites par la grammaire $V ::= n \mid (x.e)\{\sigma\}$, où $(x.e)\{\sigma\}$ est une fermeture. Les règles du jugement d'évaluation sont données ci-dessous.

$\boxed{e; \sigma \Downarrow V}$

$$\textsc{EVar}\ \frac{}{x; \sigma \Downarrow \sigma(x)} \qquad \textsc{EConst}\ \frac{}{\underline{n}; \sigma \Downarrow n} \qquad \textsc{EAdd}\ \frac{e; \sigma \Downarrow n \qquad e'; \sigma \Downarrow n'}{e \mathbin{\underline{+}} e'; \sigma \Downarrow n + n'} \qquad \textsc{EFun}\ \frac{}{\mathsf{fun}\, x.\, e; \sigma \Downarrow (x.e)\{\sigma\}}$$

$$\textsc{EApp}\ \frac{e; \sigma \Downarrow (x_f.e_f)\{\sigma_f\} \qquad e'; \sigma \Downarrow V_a \qquad e_f; \sigma_f[x_f \mapsto V_a] \Downarrow V}{e\ e'; \sigma \Downarrow V} \qquad \textsc{ETryWithL}\ \frac{e; \sigma \Downarrow V}{\mathsf{try}\, e\, \mathsf{with}\, K \Rightarrow e'; \sigma \Downarrow V} \qquad \textsc{ETryWithR}\ \frac{e; \sigma \not\downarrow K \qquad e'; \sigma \Downarrow V}{\mathsf{try}\, e\, \mathsf{with}\, K \Rightarrow e'; \sigma \Downarrow V}$$

Deux points importants : la construction raise K n'a pas de règle d'évaluation ; la règle ETryWithR utilise un nouveau jugement $e; \sigma \not\downarrow K$ pour exprimer que l'évaluation de l'expression e dans l'environnement σ lève l'exception K.

Les règles du jugement sont définies ci-dessous, avec des prémisses manquantes.

$\boxed{e; \sigma \not\downarrow K}$

$$\textsc{RAddL}\ \frac{e; \sigma \not\downarrow K}{e \pm e'; \sigma \not\downarrow K} \qquad \textsc{RAddR}\ \frac{?}{e \pm e'; \sigma \not\downarrow K} \qquad \textsc{RApp1}\ \frac{?}{e\ e'; \sigma \not\downarrow K} \qquad \textsc{RApp2}\ \frac{?}{e\ e'; \sigma \not\downarrow K} \qquad \textsc{RApp3}\ \frac{?}{e\ e'; \sigma \not\downarrow K}$$

$$\textsc{RRaise}\ \frac{}{\mathsf{raise}\, K; \sigma \not\downarrow K} \qquad \textsc{RTryWithL}\ \frac{? \qquad K \neq K'}{\mathsf{try}\, e\, \mathsf{with}\, K \Rightarrow e'; \sigma \not\downarrow K'} \qquad \textsc{RTryWithR}\ \frac{?}{\mathsf{try}\, e\, \mathsf{with}\, K \Rightarrow e'; \sigma \not\downarrow K'}$$

1. (1 point) Donnez la ou les prémisses manquantes de la règle RAddR. Cette règle doit exprimer que si e s'évalue vers une valeur V quelconque mais que e' lève l'exception K, alors $e \pm e'$ lève l'exception K.
2. (3 points) Selon la spécification donnée à la question précédente, quel est l'ensemble des exceptions K telles que $(\mathsf{raise\ Foo}) \pm (\mathsf{raise\ Bar}); \emptyset \not\downarrow K$? Comment changer les règles RAddL et RAddR pour obtenir exactement l'ensemble $\{\mathsf{Bar}\}$, c'est à dire une évaluation de droite à gauche ? Donnez les règles modifiées.
3. (3 points) Donnez la ou les prémisses manquantes des règles RApp1 à RApp3. Elles doivent spécifier une évaluation de gauche à droite.
4. (3 points) Donnez la ou les prémisses manquantes des règles RTryWithL et RTryWithR. Le jugement doit être déterministe, au sens où $e; \sigma \not\downarrow K$ et $e; \sigma \not\downarrow K'$ implique $K = K'$.

3 Typage (10 points)

On souhaite adopter une discipline de typage pour $\mathsf{Fun_{ex}}$. On a vu en cours que le type d'une expression décrit l'ensemble des valeurs en lesquelles son évaluation peut résulter. Pour $\mathsf{Fun_{ex}}$, on va être un peu plus ambitieux : le type d'une expression va également décrire l'ensemble des exceptions qui peuvent être lancées durant son évaluation. La métavariable $\mathcal{E}$ décrit les ensembles finis $\{K_1, \ldots, K_n\}$ d'exceptions. La grammaire des types est la suivante.

$$t, t' ::= \mathsf{Int} \mid t \to_{\mathcal{E}} t'$$

La spécificité de ce langage de type concerne celui des fonctions : $t \to_{\mathcal{E}} t'$, en plus de décrire le type de l'argument et du résultat, spécifie les exceptions qui peuvent être levées lors de l'évaluation du corps de la fonction. Le jugement de typage, quant à lui, prend la forme $\Gamma \vdash e : t \mid \mathcal{E}$, ce qui signifie que l'évaluation de e termine avec une valeur de type t, ou bien lève une exception appartenant à $\mathcal{E}$, en supposant que les variables libres de e aient les types prescrits par Γ.

$\boxed{\Gamma \vdash e : t \mid \mathcal{E}}$

$$\textsc{TVar}\ \frac{\Gamma(x) = t}{\Gamma \vdash x : t \mid \emptyset} \qquad \textsc{TConst}\ \frac{}{\Gamma \vdash \underline{n} : \mathsf{Int} \mid \emptyset} \qquad \textsc{TOp}\ \frac{\Gamma \vdash e : \mathsf{Int} \mid \mathcal{E}_1 \qquad \Gamma \vdash e' : \mathsf{Int} \mid \mathcal{E}_2}{\Gamma \vdash e \pm e' : \mathsf{Int} \mid \mathcal{E}_1 \cup \mathcal{E}_2}$$

$$\textsc{TFun}\ \frac{\Gamma, x : t \vdash e : t' \mid ?}{\Gamma \vdash \mathsf{fun}\, x.\, e : t \to_? t' \mid ?} \qquad \textsc{TApp}\ \frac{\Gamma \vdash e : t \to_? t' \mid ? \qquad \Gamma \vdash e' : t \mid ?}{\Gamma \vdash e\ e' : t' \mid ?} \qquad \textsc{TRaise}\ \frac{}{\Gamma \vdash \mathsf{raise}\, K : t \mid ?}$$

1. (6 points) Complétez les règles TFun, TApp et TRaise en remplaçant les points d'interrogation par des ensembles d'exceptions. Le jugement qui en résulte doit être :
   — *correct*, au sens où si $\Gamma \vdash e : t \mid \mathcal{E}$ alors l'évaluation de e dans un environnement bien typé ne peut pas lever une exception qui n'appartient pas à $\mathcal{E}$;
   — *précis*, au sens où l'ensemble $\mathcal{E}$ doit être le plus petit possible (tout en restant correct).
2. (4 points) Proposez une ou des règles pour $\mathsf{try}\, e\, \mathsf{with}\, K \Rightarrow e'$. Discutez brièvement vos choix.

4 Compilation monadique vers un langage pur (10 points)

Une fonctionnalité comme celle des exceptions peut être éliminée plus ou moins tardivement dans la chaîne de compilation. L'éliminer tôt simplifie le compilateur, au prix de performances moindres. Dans cet exercice, on va décrire une telle compilation simple, en imaginant une traduction de $\mathsf{Fun}_{\mathsf{ex}}$ vers un fragment d'OCaml **sans exceptions**. On va utiliser les types ci-dessous.

```
type ex_constructor = string
type 'a result = Rok of 'a | Rex of ex_constructor
```

Le type `t result` décrit le résultat de l'évaluation d'une expression $\mathsf{Fun}_{\mathsf{ex}}$ de type `t` traduite vers OCaml. Si l'expression $\mathsf{Fun}_{\mathsf{ex}}$ s'évalue sans exception vers la valeur v, alors sa traduction s'évalue vers **`Rok`** `v`. Si son évaluation lève l'exception K, sa traduction s'évalue vers **`Rex`** K.

1. (4 points) Il est utile de disposer de fonctions de manipulation du type `funex_v`.

   ```
   let return : 'a -> 'a result = ?
   let exc : ex_constructor -> 'a result = ?
   let bind : 'a result -> ('a -> 'b result) -> 'b result = ?
   ```

   Complétez le code ci-dessus pour obtenir des fonctions des types polymorphes spécifiés. Un appel à `bind r f` doit passer à `f` le contenu du résultat `r` si celui-ci correspond à une valeur, et renvoyer directement l'exception correspondant à `r` sinon.

2. (6 points) En utilisant les fonctions de la question précédente, complétez la traduction ci-dessous.

$$
\begin{aligned}
(\!|x|\!) &= \texttt{return}\ x \\
(\!|\underline{n}|\!) &= \texttt{return}\ n \\
(\!|e \underline{+} e'|\!) &= \texttt{bind}\ (\!|e|\!)\ \texttt{(fun n -> bind}\ (\!|e'|\!)\ \texttt{(fun n' -> return (n + n')))} \\
(\!|\mathsf{fun}\ x.\ e|\!) &= \texttt{return (fun}\ x\ \texttt{->}\ (\!|e|\!)\texttt{)} \\
(\!|e\ e'|\!) &= ? \\
(\!|\mathsf{raise}\ K|\!) &= \texttt{val\_of\_ex}\ K \\
(\!|\mathsf{try}\ e\ \mathsf{with}\ K \Rightarrow e'|\!) &= ?
\end{aligned}
$$

5 Compilation bas niveau vers x86-64 (10 points)

Ce dernier exercice concerne une implémentation plus efficace des exceptions en générant de l'assembleur x86-64 dédié. L'idée est de maintenir une pile de *gestionnaires d'exception* à l'exécution.

- L'implémentation de $\mathsf{try}\ e\ \mathsf{with}\ K \Rightarrow e'$ doit d'abord empiler un gestionnaire d'exception associant e' à l'exception K, puis exécuter le code traduit de e, puis dépiler le gestionnaire d'exception si e termine normalement.
- L'implémentation de $\mathsf{raise}\ K$ doit détourner le flot d'exécution vers le gestionnaire d'exception correspondant à K (s'il existe).

Les questions suivantes sont volontairement ouvertes. Vous prendrez soin de justifier vos réponses en incluant du code en assembleur x86-64.

1. (4 points) Quelles informations doit contenir un gestionnaire d'exception ? *Indication* : une exception peut être levée dans un appel de fonction et rattrapée par son appelant.
2. (6 points) Décrivez précisément l'opération d'empilement d'un nouveau gestionnaire d'exception, correspondant à l'implémentation de $K \Rightarrow e'$ dans $\mathsf{try}\ e\ \mathsf{with}\ K \Rightarrow e'$, et l'opération de déroutage, correspondant à l'implémentation de $\mathsf{raise}\ K'$.