Algorithmes, Réseaux, Langages (INF431)
Contrôle des connaissances CC2
partie Langages
CORRIGÉ

29 juin 2011

Le sujet de ce contrôle de classement est constitué de deux parties, intitulées « Réseaux et concurrence » et « Langages ». **Les deux parties doivent être traitées.** Elles sont indépendantes.

Ceci est la partie « Langages ». **Elle doit être traitée sur des feuilles ROSES.**

Les deux premières sections (« Sémantique » et « Interprétation ») doivent en principe être traitées avant d'aborder les sections suivantes. Les sections suivantes sont indépendantes les unes des autres.

Introduction

On s'intéresse à un fragment du langage MiniliX dont la syntaxe abstraite est donnée dans la figure 1.

On souhaite introduire dans ce langage une notion de *non-déterminisme.*

On veut qu'un programme produise non pas nécessairement une valeur unique, mais un certain nombre de valeurs possibles. Par exemple, on pourrait souhaiter écrire une expression *coin* qui peut produire la valeur 0 et qui peut produire la valeur 1.

On ne veut pas que le choix entre les deux valeurs possibles de l'expression *coin* soit fait de façon définitive pendant l'évaluation de *coin*. On veut que *coin* propose les deux résultats possibles 0 et 1 et que la suite du programme explore les deux possibilités qui en découlent. On ne dira donc pas que « la valeur de l'expression *coin* est 0 ou 1 », mais on préférera dire que *les valeurs (possibles) de l'expression coin sont 0 et 1.*

On veut :

1. que l'exécution d'un programme puisse se sub-diviser en plusieurs branches ;
2. que chaque branche puisse à son tour se sub-diviser en plusieurs branches ;
3. que certaines branches puissent échouer et disparaître ;
4. que les valeurs produites ultimement par les branches survivantes soient considérées comme les résultats du programme.

On ajoute donc au langage de la figure 1 deux constructions supplémentaires :

e	::=	...	(voir la figure 1)
		$e \mid e$	choix non déterministe
		fail	échec

e	::=		*Expression*
		— valeurs et opérations primitives	
		k	constante booléenne ou entière
		$e\,op\,e$	$op \in \{+, -, *, /, ==, !=, <, \leq, >, \geq\}$
		— gestion des variables	
		x	accès à une variable (lecture)
		let $x = e$ **in** e	définition locale
		— fonctions	
		$e(e, \ldots)$	appel de fonction
		— contrôle structuré	
		if e **then** e **else** e	expression conditionnelle
d	::=	**fun** $f(x, \ldots)\,\{e\}$	*Définition de fonction*
p	::=	$d, \ldots\ \{e\}$	*Programme complet*

FIGURE 1 – Syntaxe d'un fragment de MiniliX

Sémantique

Une *valeur* v est soit un booléen soit un entier.

On rappelle qu'un multi-ensemble est un ensemble dans lequel un élément peut apparaître un nombre arbitraire de fois. (Par contraste, dans un ensemble ordinaire, un élément soit apparaît une fois soit n'apparaît pas.) Par exemple, le multi-ensemble $\{4, 0, 4\}$ contient deux fois l'élément 4 et une fois l'élément 0. L'ordre des éléments n'importe pas : les multi-ensembles $\{4, 0, 4\}$ et $\{0, 4, 4\}$ sont égaux. L'union de deux multi-ensembles se construit par simple concaténation, sans élimination des doublons : par exemple, $\{0, 4, 4\} \cup \{0, 1, 1\}$ est égal à $\{0, 4, 4, 0, 1, 1\}$, que l'on peut écrire également $\{0, 0, 1, 1, 4, 4\}$.

Dans la suite, on manipule des multi-ensembles de valeurs. Dans la section « Interprétation », on représentera un multi-ensemble de valeurs par une liste de type `LinkedList<Value>`, dans laquelle on considérera que l'ordre des éléments n'a pas d'importance.

On considère que chaque expression produit une *réponse*. Une *réponse* est soit un multi-ensemble V de valeurs, soit $\bot$.

Si une expression e a pour réponse V, cela signifie informellement que e s'exécute sans erreur, termine, et que les valeurs produites par les différentes branches qui apparaissent lors de l'exécution de e sont exactement les éléments de V. Les éléments de V sont *les valeurs* de l'expression e.

Si une expression e a pour réponse $\bot$, cela signifie informellement que l'exécution de e ne termine pas ou bien provoque une erreur d'exécution. On souligne qu'il suffira qu'une seule branche provoque une erreur d'exécution pour que la réponse soit $\bot$. En d'autres termes, une erreur d'exécution est fatale et met fin à l'exécution du programme tout entier.

La sémantique du langage peut être définie mathématiquement comme une fonction qui à chaque expression e associe une réponse $[\![e]\!]$. (On ne s'intéresse pour le moment qu'à des expressions closes, ce qui nous évite de demander un environnement.)

Une expression de la forme « $e_1 \mid e_2$ » représente un choix non déterministe. Elle provoque la division de la branche courante en deux sous-branches. Pour en définir la sémantique, on pose que :

1. si $[\![e_1]\!]$ et $[\![e_2]\!]$ sont des multi-ensembles de valeurs, alors $[\![e_1 \mid e_2]\!]$ est leur union ;
2. si $[\![e_1]\!]$ ou $[\![e_2]\!]$ est $\bot$, alors $[\![e_1 \mid e_2]\!]$ est $\bot$.

Une expression de la forme « **fail** » représente un échec. Elle provoque la mort de la branche courante. Pour en définir la sémantique, on pose $[\![\textbf{fail}]\!] = \emptyset$. On souligne que « **fail** » n'est pas une erreur d'exécution. Elle met fin à l'exécution de la branche courante uniquement, pas à l'exécution du programme tout entier.

Question 1 Quelle doit être la définition de $[\![k]\!]$, où k est une constante (booléenne ou entière) ?◇

Solution. Une expression constante termine, ne provoque pas d'erreur d'exécution, et produit un seul résultat, à savoir la valeur k. Sa réponse doit donc être le multi-ensemble singleton $\{k\}$.□

Question 2 Quelle est la réponse de chacune des expressions suivantes ?

$$0 \mid 1 \mid 2$$
$$0 \mid \textbf{fail} \mid 2$$ ◇

Solution. On s'appuie sur la réponse à la question précédente et sur la définition de la sémantique des constructions $e_1 \mid e_2$ et **fail**.

L'expression « $0 \mid 1 \mid 2$ » a pour réponse $\{0, 1, 2\}$.

L'expression « $0 \mid$ **fail** $\mid 2$ » a pour réponse $\{0, 2\}$. En effet, la branche **fail** a pour réponse un multi-ensemble vide et ne contribue donc aucune valeur. Elle ne constitue pas une erreur d'exécution et ne « tue » donc pas les autres branches. □

On suppose que la sémantique de chaque opérateur op dans le cadre déterministe ordinaire est connue. En d'autres termes, si v_1 et v_2 sont deux valeurs, on s'autorise à noter $v_1 \,[\![op]\!]\, v_2$ le résultat de l'application de l'opérateur op aux valeurs v_1 et v_2. Ce résultat est soit une valeur, soit une erreur d'exécution $\bot$. Par exemple, $2 \,[\![+]\!]\, 2$ vaut 4 et $2 \,[\![<]\!]\, 3$ vaut *true* et *true* $[\![+]\!]$ *true* vaut $\bot$.

Une expression de la forme « $e_1 \, op \, e_2$ » représente une opération primitive ordinaire. Chaque opération primitive est en soi déterministe. Néanmoins, l'expression « $e_1 \, op \, e_2$ » peut produire plusieurs résultats si l'une des sous-expressions e_1 ou e_2 produit plusieurs résultats. Pour en définir la sémantique, on pose que :

1. si $[\![e_1]\!]$ et $[\![e_2]\!]$ sont des multi-ensembles de valeurs, et si pour chaque $v_1 \in [\![e_1]\!]$ et $v_2 \in [\![e_2]\!]$ on a $v_1 \,[\![op]\!]\, v_2 \neq \bot$, alors $[\![e_1 \, op \, e_2]\!]$ est le multi-ensemble :
$$\{v_1 \,[\![op]\!]\, v_2 \mid v_1 \in [\![e_1]\!], v_2 \in [\![e_2]\!]\}$$
2. sinon, $[\![e_1 \, op \, e_2]\!]$ est $\bot$.

Question 3 Quelle est la réponse de chacune des expressions suivantes ?

$$\textit{true} + \textit{true}$$
$$0 \mid (\textit{true} + \textit{true})$$ ◇

Solution. La sous-expression *true* produit le résultat $\{\textit{true}\}$. L'opération *true* $[\![+]\!]$ *true* produit $\bot$, donc, d'après la définition ci-dessus, la réponse de l'expression *true* + *true* est $\bot$. Cette expression provoque une erreur d'exécution.

L'expression « $0 \mid (\textit{true} + \textit{true})$ » a également pour réponse $\bot$, puisque sa seconde branche produit la réponse $\bot$.

D'après notre définition de la sémantique, si une seule branche rencontre une erreur d'exécution, alors le programme tout entier rencontre une erreur d'exécution. Une erreur d'exécution au sein d'une seule branche est considérée comme fatale pour le programme tout entier. □

Question 4 Quelle est la réponse de chacune des expressions suivantes ?

$$1 + \textbf{fail}$$
$$1 + (0 \mid \textbf{fail})$$
$$1 + (0 \mid 1)$$
$$(0 \mid 1) + (0 \mid 1)$$

◇

Solution. L'expression « $1 + \textbf{fail}$ » a pour réponse $\emptyset$, car **fail** ne produit aucune valeur et l'addition $1 + []$ ne change rien à cela.

L'expression « $1 + (0 \mid \textbf{fail})$ » a pour réponse $\{1\}$. En général, on peut noter que « $e \mid \textbf{fail}$ » a la même réponse que e.

L'expression « $1 + (0 \mid 1)$ » a pour réponse $\{1, 2\}$, car la sous-expression « $0 \mid 1$ » a pour réponse $\{0, 1\}$ et l'addition $1 + []$ vient incrémenter chacune de ces valeurs.

L'expression « $(0 \mid 1) + (0 \mid 1)$ » a pour réponse $\{0, 1, 1, 2\}$. En effet, chacune des deux sous-expressions « $0 \mid 1$ » a pour réponse $\{0, 1\}$. Quatre additions élémentaires sont donc effectuées, dont deux produisent le même résultat. □

Interprétation

On souhaite définir deux classes `EFail` et `EChoice`, sous-classes de `Expression`, qui représentent les constructions « **fail** » et « $e \mid e$ ».

Question 5 Écrire en Java la déclaration des classes `EFail` et `EChoice`. On demande de faire figurer dans chaque déclaration l'en-tête, les champs, et le constructeur. On ne demande pour le moment aucune définition de méthode. ◇

Solution. Parce que l'expression « **fail** » n'a aucune sous-expression, la classe `EFail` n'a aucun champ. Son constructeur est vide et pourrait être omis, car Java fournit par défaut un constructeur vide.

```
class EFail extends Expression {

    EFail ()
    {
    }

}
```

Parce que l'expression « $e_1 \mid e_2$ » a deux sous-expressions e_1 et e_2, la classe `EChoice` a deux champs de type `Expression`, que nous pouvons nommer par exemple `left` et `right`. Ces champs ne sont pas modifiables. Le constructeur les initialise.

```
class EChoice extends Expression {

    final Expression left, right;

    EChoice (Expression left, Expression right)
    {
        this.left = left;
        this.right = right;
    }

}
```

```
class EIntegerConstant extends Expression {
    final int i;
}
class EPrimAdd extends Expression {
    final Expression left, right;
}
class EVarRead extends Expression {
    final String x;
}
class ELet extends Expression {
    final String x;
    final Expression left;
    final Expression right;
}
```

FIGURE 2 – Quelques sous-classes de `Expression` et leurs champs

On souhaite à présent étendre l'interprète de l'amphi 14 pour permettre l'évaluation de programmes MiniliX non déterministes.

On représente un multi-ensemble de valeurs à l'aide d'une liste de type `LinkedList<Value>`. On rappelle qu'on peut créer une liste vide à l'aide de l'expression « `new LinkedList<Value> ()` » et que l'on peut ajouter un élément `x` à une liste `l` à l'aide de l'instruction « `l.add(x)` ».

On rappelle que la classe abstraite `Value` a plusieurs sous-classes, ici `VBoolean` et `VInteger`, qui représentent les différentes sortes de valeurs. On rappelle que la classe abstraite `Value` définit une méthode `asInteger` qui renvoie un entier si l'objet `this` appartient à la sous-classe `VInteger` et provoque une erreur d'exécution dans le cas contraire. La façon dont une erreur d'exécution est signalée ne change pas : la méthode `asInteger` met fin à l'exécution de l'interprète via un appel à `System.exit`.

La structure des environnements ne change pas : à une variable, un environnement associe une valeur. On rappelle que l'environnement vide est `Environment.empty`, que la valeur associée à une variable `x` par un environnement `env` s'écrit `env.search(x).value`, et que l'expression `env.extend(x,v)` produit un nouvel environnement qui à `env` ajoute une liaison de la variable `x` à la valeur `v`.

La signature de la méthode `eval` est modifiée pour indiquer que l'évaluation produit non pas une valeur, mais un multi-ensemble de valeurs :

```
abstract class Expression {

    abstract LinkedList<Value> eval (Environment env);

}
```

On souligne que si `eval` termine (sans boucler et sans provoquer d'erreur) alors son résultat n'est jamais `null`, mais toujours une liste de valeurs bien définie.

On rappelle que les classes `EIntegerConstant` et `EPrimAdd` sont les sous-classes de `Expression` qui représentent les constructions « k » et « $e + e$ ». La figure 2 rappelle quels sont les champs de ces classes.

Question 6 En suivant la définition de la notion de *réponse* d'une expression, implémenter la méthode `eval` successivement pour chacune des classes `EIntegerConstant`, `EFail`, `EChoice`, `EPrimAdd`. ◇

Solution. La méthode `eval` de la classe `EIntegerConstant` produit un multi-ensemble singleton :

```
class EIntegerConstant extends Expression {

    final int i;

    LinkedList<Value> eval (Environment env)
    {
        LinkedList<Value> results = new LinkedList<Value> ();
        results.add(new VInteger (i));
        return results;
    }

}
```

La méthode `eval` de la classe `EFail` produit un multi-ensemble vide :

```
class EFail extends Expression {

    LinkedList<Value> eval (Environment env)
    {
        return new LinkedList<Value> ();
    }

}
```

La méthode `eval` de la classe `EChoice` effectue un appel récursif pour évaluer chacune des deux branches. (L'ordre dans lequel ces appels récursifs sont effectués n'a pas d'importance.) Chacun de ces appels produit un multi-ensemble (une liste) de valeurs, et il reste à en calculer l'union (la concaténation).

```
class EChoice extends Expression {

    final Expression left, right;

    LinkedList<Value> eval (Environment env)
    {
        LinkedList<Value> results = new LinkedList<Value> ();
        for (Value v : left.eval(env))
            results.add(v);
        for (Value v : right.eval(env))
            results.add(v);
        return results;
    }

}
```

Notons que, au lieu d'utiliser une boucle et d'ajouter les éléments dans `results` un par un, on peut également utiliser la méthode `addAll`.

La méthode `eval` de la classe `EPrimAdd` effectue d'abord un appel récursif pour évaluer chacune des deux sous-expressions. (L'ordre dans lequel ces appels récursifs sont effectués n'a pas d'importance.)

Chacun de ces appels produit un multi-ensemble (une liste) de valeurs, et il reste à calculer toutes les sommes d'une valeur `v1` prise dans le premier et d'une valeur `v2` prise dans le second. Si l'un de ces calculs produit une erreur d'exécution, celle-ci est fatale. (La méthode `asInteger` peut mettre fin à l'exécution de l'interprète.) Sinon, on renvoie le multi-ensemble de toutes ces sommes.

```
class EPrimAdd extends Expression {
```

```
        final Expression left, right;

        LinkedList<Value> eval (Environment env)
        {
            LinkedList<Value> results = new LinkedList<Value> ();
            LinkedList<Value> results1 = left.eval(env);
            LinkedList<Value> results2 = right.eval(env);
            for (Value v1 : results1) {
                for (Value v2 : results2) {
                    int i1 = v1.asInteger();
                    int i2 = v2.asInteger();
                    results.add(new VInteger (i1 + i2));
                }
            }
            return results;
        }

    }
```

Il faut noter que la tentation de gagner en concision en ne nommant pas les résultats intermédiaires `results1` et `results2` était fausse. En effet, si l'on écrivait :

```
            for (Value v1 : left.eval(env)) {
                for (Value v2 : right.eval(env)) {
                    int i1 = v1.asInteger();
                    int i2 = v2.asInteger();
                    results.add(new VInteger (i1 + i2));
                }
            }
```

alors l'évaluation de la sous-expression `right` pouvait avoir lieu zéro fois, une fois, ou plusieurs fois. Or, évaluer cette sous-expression plusieurs fois est inefficace ; et, pire, l'évaluer zéro fois est incorrect, car l'interprète peut alors réussir dans une situation où la sémantique dicte qu'il devrait provoquer une erreur d'exécution. □

On s'intéresse à présent aux variables « x » et aux définitions locales « **let** $x = e_1$ **in** e_2 ».

Les deux questions suivantes concernent la sémantique de ces expressions. La première étudie un exemple, la seconde le cas général, en Java. On n'a pas donné la définition mathématique de la réponse de ces expressions, et on ne vous demande pas de la donner.

Question 7 Selon vous, quelle devrait être la réponse des expressions suivantes ?

$$\textbf{let}\, coin = (0 \mid 1)\, \textbf{in}\, (coin == coin)$$
$$\textbf{let}\, coin_1 = (0 \mid 1)\, \textbf{in let}\, coin_2 = (0 \mid 1)\, \textbf{in}\, (coin_1 == coin_2)$$ ◇

Solution. L'évaluation de la sous-expression « 0 | 1 » provoque la création de deux branches. Il semble naturel que, dans l'une des branches, la variable *coin* soit liée à la valeur 0 et que dans l'autre elle soit liée à la valeur 1. Dans chacune de ces branches, la comparaison *coin* == *coin* renvoie la valeur *true*. Le multi-ensemble des valeurs finales sera donc {*true*, *true*}.

On aurait pu tenter d'imaginer que la variable *coin* soit liée à l'expression « 0 | 1 » et que par conséquent l'expression « **let** *coin* = (0 | 1) **in** (*coin* == *coin*) » soit équivalente à « (0 | 1) == (0 | 1) », qui produit la réponse {*true*, *true*, *false*, *false*}. Cependant, cela contredirait le sujet, qui spécifie que l'environnement associe à chaque variable une valeur, et non pas une expression ou un multi-ensemble de valeurs.

À nouveau, l'évaluation de la première sous-expression « 0 | 1 » provoque la création de deux branches. La variable $coin_1$ est liée à la valeur 0 dans l'une et 1 dans l'autre. Puis, dans chacune de

ces branches, l'évaluation de la seconde sous-expression « 0 | 1 » provoque à nouveau la création de deux branches. Nous avons alors quatre branches, dans lesquelles les variables $coin_1$ et $coin_2$ prennent respectivement les valeurs 0 et 0, 0 et 1, 1 et 0, 1 et 1. Dans deux de ces branches, la comparaison $coin_1 == coin_2$ produit *true*, et dans les deux autres branches, elle produit *false*. Le multi-ensemble des valeurs booléennes finales est donc {*true*, *true*, *false*, *false*}. □

On rappelle que les constructions « x » et « **let** $x = e$ **in** e » sont représentées par les classes `EVarRead` et `ELet`, sous-classes de `Expression`. La figure 2 rappelle quels sont les champs de ces classes.

Question 8 En accord avec votre réponse à la question précédente, implémenter la méthode `eval` pour les classes `EVarRead` et `ELet`. ◇

Solution. Il est clair que l'évaluation d'une variable `x` doit produire un multi-ensemble singleton dont l'unique valeur est celle associée à `x` par l'environnement.

```
class EVarRead extends Expression {

    final String x;

    LinkedList<Value> eval (Environment env)
    {
        LinkedList<Value> results = new LinkedList<Value> ();
        results.add(env.search(x).value);
        return results;
    }

}
```

Pour évaluer l'expression « **let** $x = e_1$ **in** e_2 », on évalue d'abord e_1, comme dans le cas déterministe ordinaire. Cette évaluation produit un multi-ensemble de valeurs. Alors, pour chaque valeur v_1 prise dans ce multi-ensemble, on étend l'environnement avec une liaison de x à v_1, et on évalue e_2. La réponse finale est l'union des réponses fournies par ces évaluations de e_2.

```
class ELet extends Expression {

    final String x;
    final Expression left;
    final Expression right;

    LinkedList<Value> eval (Environment env)
    {
        LinkedList<Value> results = new LinkedList<Value> ();
        for (Value v1 : left.eval(env))
            results.addAll(right.eval(env.extend(x, v1)));
        return results;
    }

}
```

Programmation en MiniliX non déterministe

Question 9 Définir dans le langage MiniliX non déterministe une fonction *random* à un argument (entier) et un résultat (entier) de sorte que, pour tout entier naturel n, l'expression *random*(n) ait pour réponse le multi-ensemble $\{0, 1, \ldots, n-1\}$. ◇

Solution. On pose :

$$\textbf{fun}\ random(x)\ \{\ \textbf{if}\ x \leq 0\ \textbf{then fail else}\ x-1 \mid random(x-1)\ \}$$

Si on le souhaite, on peut vérifier par récurrence sur n que la réponse de l'expression $random(n)$ est bien le multi-ensemble $\{0, 1, \ldots, n-1\}$.

Notons qu'on pourrait écrire « $x == 0$ » au lieu de « $x \leq 0$ », puisque l'énoncé ne précise pas quel doit être le comportement de $random(n)$ lorsque n est négatif.

On pourrait également écrire $random(x-1) \mid x-1$ au lieu de $x-1 \mid random(x-1)$. La sémantique du langage est définie en termes de multi-ensembles, ce qui indique que l'ordre dans lequel les valeurs possibles sont produites n'a pas d'importance. □

Question 10 À l'aide de la fonction *random*, définir une fonction *sqrt* à un argument (entier) et un résultat (entier) de sorte que, pour tout entier naturel n, l'expression $sqrt(n)$ ait pour réponse le singleton $\{\sqrt{n}\}$ si n est un carré parfait et le multi-ensemble vide $\emptyset$ dans le cas contraire. On ne recherche pas l'efficacité. ◇

Solution. On pose :

$$\textbf{fun}\ sqrt(x)\ \{\ \textbf{let}\ r = random(x+1)\ \textbf{in if}\ r * r == x\ \textbf{then}\ r\ \textbf{else fail}\ \}$$

Lors d'un appel à $sqrt(n)$, l'évaluation de la sous-expression $random(n+1)$ produit tous les entiers compris entre 0 et n inclus, donc parmi eux l'entier $\sqrt{n}$, s'il existe. Autrement dit, elle provoque l'apparition de $n+1$ branches d'exécution, et dans chaque branche, la variable r prend une valeur différente, comprise entre 0 et n. Parmi ces candidats r, il faut conserver celui que nous recherchons et éliminer les autres. On reconnaît le candidat recherché grâce au test $r * r == n$. Si ce test réussit, on renvoie r ; sinon, on échoue. Ainsi, parmi les $n+1$ branches créées par l'appel à $random(n+1)$, une branche au plus survit, et si une branche survit, sa valeur est $\sqrt{n}$. □

Typage

On souhaite maintenant étendre le système de types de MiniliX (amphi 17) afin d'autoriser les deux constructions « **fail** » et « $e \mid e$ ».

La forme du jugement de typage ne change pas : elle reste « $\Sigma, \Gamma \vdash e : \tau$ ». Son interprétation informelle change légèrement, parce qu'une expression produit en général plusieurs valeurs. On peut considérer qu'un tel jugement signifie maintenant : « sous les hypothèses Σ et Γ, l'expression e ne provoque aucune erreur d'exécution, et chacune des valeurs qu'elle produit admet le type τ. »

Aucune des règles de typage existantes n'est modifiée.

Question 11 Indiquer quelles règles de typage il faut ajouter pour décrire les expressions « **fail** » et « $e_1 \mid e_2$ ». On rappelle que chaque règle de typage indique à quelle(s) condition(s) une expression est bien typée et quel est alors son type. ◇

Solution. L'expression **fail** ne constitue pas une erreur d'exécution : elle met fin à la branche courante, mais pas au programme tout entier. Il ne faut donc pas chercher à l'interdire : au contraire, il faut considérer qu'elle est toujours bien typée. De plus, on peut considérer qu'elle admet n'importe quel type τ. Ceci peut paraître surprenant, mais est en fait naturel : puisque « **fail** » ne produit aucune valeur (sa réponse est le multi-ensemble vide), il est vrai que toutes les valeurs produites par « **fail** » ont le type τ. La règle de typage est donc :

$$\Sigma, \Gamma \vdash \textbf{fail} : \tau$$

Cette règle est un axiome (elle n'a aucune prémisse).

Pour garantir que l'expression « $e_1 \mid e_2$ » ne produit pas d'erreur d'exécution, il suffit de garantir que ni e_1 ni e_2 ne peut produire une erreur d'exécution. Pour que « $e_1 \mid e_2$ » soit bien typée, on exige donc que e_1 et e_2 soient bien typées. De plus, les valeurs produites par « $e_1 \mid e_2$ » sont l'union de celles produites par e_1 et de celles produites par e_2. Si l'on souhaite que toutes les valeurs produites par « $e_1 \mid e_2$ » admettent un même type τ, il faut exiger que toutes les valeurs produites par e_1 admettent le type τ et que toutes les valeurs produites par e_2 admettent le type τ. La règle de typage est donc :

$$\frac{\Sigma, \Gamma \vdash e_1 : \tau \qquad \Sigma, \Gamma \vdash e_2 : \tau}{\Sigma, \Gamma \vdash e_1 \mid e_2 : \tau}$$

□

Question 12 En principe, les règles de typage proposées lors de la question précédente devraient permettre de montrer que, si une expression e admet un type τ, alors l'expression « $e \mid$ **fail** » admet également le type τ. Montrer donc comment, à partir du jugement $\Sigma, \Gamma \vdash e : \tau$ et par application des règles de typage, on peut obtenir le jugement $\Sigma, \Gamma \vdash e \mid \textbf{fail} : \tau$. ◇

Solution. On construit la dérivation de typage suivante :

$$\frac{\Sigma, \Gamma \vdash e : \tau \qquad \Sigma, \Gamma \vdash \textbf{fail} : \tau}{\Sigma, \Gamma \vdash e \mid \textbf{fail} : \tau}$$

La prémisse de gauche est notre hypothèse. La prémisse de droite est obtenue grâce à l'axiome de typage qui concerne **fail**. On applique enfin la règle de typage associée à la construction « $e \mid e$ » pour obtenir la conclusion attendue. □

Interprétation en style CPS

On souhaite enfin étendre l'interprète en style CPS (« continuation-passing style ») de l'amphi 16 pour permettre l'évaluation de programmes MiniliX non déterministes.

Au lieu de calculer le multi-ensemble (la liste) de tous les résultats, on voudrait se comporter de façon paresseuse et ne calculer les résultats qu'au fur et à mesure qu'ils sont exigés.

Par exemple, si on doit évaluer l'expression « $0 \mid e$ », on peut annoncer que le premier résultat est 0 sans même évaluer e. On n'évaluera e que si le « reste du programme » ne se satisfait pas du résultat 0. Par exemple, si le programme est « **let** $x = (0 \mid e)$ **in if** $x == 0$ **then fail else** *true* », on tente d'abord de lier x à la valeur 0, mais cela nous mène un peu plus loin à un échec (**fail**). On doit alors revenir en arrière, obtenir une (la première) valeur de l'expression e, lier x à cette valeur, et essayer à nouveau d'évaluer « **if** $x == 0$ **then fail else** *true* ».

Dans le programme ci-dessus, au moment où la méthode `evalCPS` est appelée pour évaluer la sous-expression « $0 \mid e$ », le reste du programme « **let** $x = []$ **in if** $x == 0$ **then fail else** *true* » est représenté par la continuation `c`. « Essayer » une valeur v pour la variable x revient donc à appeler la continuation `c` en lui passant la valeur v. Et rien ne nous empêche d'essayer successivement plusieurs valeurs différentes.

Signalons un détail technique. La sémantique de notre langage indique que, si une branche ne termine pas (c'est-à-dire : produit la réponse $\bot$), alors le programme tout entier ne termine pas. Pour respecter cette sémantique, avant de produire un premier résultat, il faudrait s'assurer que toutes les branches terminent, donc calculer tous les résultats, ce qui ici n'est pas souhaitable, puisque nous voulons être paresseux. Notre interprète CPS ne respectera donc pas exactement la sémantique : il sera plus « optimiste » et dans certains cas produira une valeur dans un cas où l'interprète des questions 6 et 8 aurait bouclé ou bien provoqué une erreur d'exécution.

De plus, la sémantique de notre langage indique que les expressions « $e_1 \mid e_2$ » et « $e_2 \mid e_1$ » sont équivalentes, parce que l'ordre des résultats n'a pas d'importance. Ici, si on veut produire

d'abord un « premier » résultat, il faut que nous définissions ce que nous entendons par là. On convient que l'interprète CPS évaluera d'abord e_1, et, en cas d'échec, évaluera e_2. Les expressions « $e_1 \mid e_2$ » et « $e_2 \mid e_1$ » pourront donc mener à des « premiers » résultats différents.

On se donne une exception `FAIL` et on adopte la convention que la méthode `evalCPS` de la classe `Expression`, comme la méthode `run` de la classe `Continuation`, pourront lancer l'exception `FAIL` en cas d'échec. Leurs signatures respectives sont donc :

```
abstract class Expression {
    abstract Value evalCPS (Environment env, Continuation c) throws FAIL;
}
abstract class Continuation {
    abstract Value run (Value v) throws FAIL;
}
```

Pour toutes les sous-classes pré-existantes de la classe `Expression`, l'implémentation de la méthode `evalCPS` est inchangée. Il ne reste donc qu'à :

Question 13 Implémenter la méthode `evalCPS` des classes `EFail` et `EChoice`. ◇

Solution. La première de ces deux méthodes est triviale : on lance une exception. Ceci signifie que cette branche ne peut pas continuer ; il faut revenir en arrière.

```
    class EFail extends Expression {

        Value evalCPS (Environment env, Continuation c) throws FAIL
        {
            throw new FAIL ();
        }

    }
```

Jusqu'où faut-il revenir en arrière ? Jusqu'au dernier choix que l'on a effectué, et que l'on aurait pu effectuer différemment. Ceci va s'écrire très simplement dans la méthode `evalCPS` de la classe `EChoice` :

```
    class EChoice extends Expression {

        final Expression left, right;

        Value evalCPS (Environment env, Continuation c) throws FAIL
        {
            try {
                return left.evalCPS(env, c);
            } catch (FAIL f) {
                return right.evalCPS(env, c);
            }
        }

    }
```

On essaie d'abord d'emprunter la branche de gauche, mais si cela échoue (soit parce que l'évaluation de l'expression `left` elle-même échoue, soit parce que cette évaluation produit une valeur qui conduit ensuite à un échec de la continuation `c`) alors on rattrape l'exception qui signale cet échec et on essaie la branche de droite. (Si celle-ci échoue à son tour, l'exception s'échappera.)

Cet interprète n'est plus « en style CPS », parce que l'appel `left.evalCPS(env,c)` est situé à l'intérieur d'une construction `try` et, pour cette raison, n'est pas terminal. On pourrait obtenir un véritable interprète en style CPS pour notre langage non déterministe en transformant à nouveau l'interprète et en remplaçant l'exception `FAIL` par une continuation d'échec. Mais c'est une autre histoire... □

Question 14 Comparer les deux interprètes de MiniliX. ◇

Solution. Le premier interprète est inefficace parce qu'il alloue des listes de résultats dont la manipulation est coûteuse. En particulier, la méthode `eval` de la classe `EChoice` copie des listes. De plus, ce premier interprète est strict (glouton) : par construction, il calcule *tous* les résultats.

Le second interprète est plus efficace parce qu'il n'utilise aucune liste de résultats. Il utilise la pile (explicite) de continuations de l'interprète CPS étudié en cours, plus une pile (implicite) de gestionnaires d'exceptions. Il est paresseux : il peut calculer un résultat sans les calculer tous. Dans certains cas, calculer tous les résultats est exponentiellement plus coûteux que n'en calculer qu'un. □