

Algorithmes, Réseaux, Langages (INF431)

Contrôle des connaissances CC2

partie Langages

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 :

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

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

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 \perp .

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 \perp , 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 \perp . 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 $\llbracket e \rrbracket$. (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 $\llbracket e_1 \rrbracket$ et $\llbracket e_2 \rrbracket$ sont des multi-ensembles de valeurs, alors $\llbracket e_1 \mid e_2 \rrbracket$ est leur union ;
2. si $\llbracket e_1 \rrbracket$ ou $\llbracket e_2 \rrbracket$ est \perp , alors $\llbracket e_1 \mid e_2 \rrbracket$ est \perp .

Une expression de la forme « **fail** » représente un échec. Elle provoque la mort de la branche courante. Pour en définir la sémantique, on pose $\llbracket \text{fail} \rrbracket = \emptyset$. 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 $\llbracket k \rrbracket$, où k est une constante (booléenne ou entière) ? \diamond

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

$$\begin{array}{l} 0 \mid 1 \mid 2 \\ 0 \mid \text{fail} \mid 2 \end{array} \quad \diamond$$

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 $v_1 \llbracket op \rrbracket v_2$ 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 \perp . Par exemple, $2 \llbracket + \rrbracket 2$ vaut 4 et $2 \llbracket < \rrbracket 3$ vaut *true* et *true* $\llbracket + \rrbracket$ *true* vaut \perp .

Une expression de la forme « $e_1 \text{ op } e_2$ » représente une opération primitive ordinaire. Chaque opération primitive est en soi déterministe. Néanmoins, l'expression « $e_1 \text{ op } e_2$ » 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 $\llbracket e_1 \rrbracket$ et $\llbracket e_2 \rrbracket$ sont des multi-ensembles de valeurs, et si pour chaque $v_1 \in \llbracket e_1 \rrbracket$ et $v_2 \in \llbracket e_2 \rrbracket$ on a $v_1 \llbracket op \rrbracket v_2 \neq \perp$, alors $\llbracket e_1 \text{ op } e_2 \rrbracket$ est le multi-ensemble :

$$\{v_1 \llbracket op \rrbracket v_2 \mid v_1 \in \llbracket e_1 \rrbracket, v_2 \in \llbracket e_2 \rrbracket\}$$

2. sinon, $\llbracket e_1 \text{ op } e_2 \rrbracket$ est \perp .

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

$$\begin{array}{l} \text{true} + \text{true} \\ 0 \mid (\text{true} + \text{true}) \end{array} \quad \diamond$$

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

$$\begin{array}{l} 1 + \text{fail} \\ 1 + (0 \mid \text{fail}) \\ 1 + (0 \mid 1) \\ (0 \mid 1) + (0 \mid 1) \end{array} \quad \diamond$$

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. \diamond

```

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`. ◇

On s'intéresse à présent aux variables « `x` » et aux définitions locales « `let x = e1 in e2` ».

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 ?

```
let coin = (0 | 1) in (coin == coin)
let coin1 = (0 | 1) in let coin2 = (0 | 1) in (coin1 == coin2)
```

◇

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`. ◇

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 $\{0, 1, \dots, n - 1\}$. ◇

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. ◇

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. ◇

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 $\Sigma, \Gamma \vdash e \mid$ **fail** : τ . ◇

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 \perp), 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`. ◇

Question 14 Comparer les deux interprètes de MiniliX. ◇