Examen final d'ILP
1ère session
Éléments de réponse

Christian Queinnec

15 décembre 2009

1 Généralités

Lisez bien l'énoncé et les exemples. Présentez correctement vos programmes en particulier l'indentation (souvent trompeuse), les tabulations (détruisant l'ascii-art) et la marge droite (pas plus de 80 caractères).

Si l'on vous demande un schéma de compilation et du code implantant ce schéma alors, les deux doivent être cohérents : répondre deux choses différentes n'augmente nullement vos chances.

Soyez terminologiquement précis : une variable n'est pas un argument (une fonction est définie avec des variables, elle est invoquée avec des arguments).

Si vous engendrez quelque chose comme `ILP_f` alors, il faut que cette chose (variable ou fonction) soit définie dans la bibliothèque d'exécution et au moins commenté afin que soit compréhensible ce que cela vaut ou effectue.

Qu'il n'y ait pas de vrais entiers en Javascript, n'empêche nullement ILP4 d'en avoir. Il y a juste la sémantique du reste et du modulo qui s'en ressente et qui devrait donc faire l'objet de fonctions appropriées dans la bibliothèque d'exécution.

Le langage à compiler était ILP4 et le compilateur suggéré ne recevait en entrée que des IAST4. En aucun cas, un recours aux IAST2, aux délégués ne pouvait être raisonnablement envisagé. Un code tel que

```
((CEAST*) iast.getX())
```

est ultra-dangereux car il requiert une preuve qu'`iast` est bien une instance de `CEAST*`, preuve qui ne peut s'acquérir que si vous maitrisez la construction de ces objets ce qui n'était pas le cas ici. En plus, il n'y en avait aucun besoin comme le montre le corrigé.

La syntaxe de Java et de Javascript sont l'une et l'autre basée sur la syntaxe de C et donc sont largement similaires. La compilation vers Javascript est donc pratiquement la même que celle pour C sauf que :

- La destination (**`void`**) n'existe pas en Javascript
- Javascript dispose de **`try`**/**`catch`**/**`finally`** en natif
- Javascript dispose des booléens **`true`** et **`false`**
- Suivant l'évaluateur, on peut avoir `print()` ou `document.write()`, il faut donc dissimuler cette différence dans la bibliothèque d'exécution.

En revanche, on ne peut utiliser aucune méthode `compile*` d'ILP4 car elles engendrent du C alors que l'on souhaite du Javascript et qu'il suffit qu'un **`try`**/**`catch`**/**`finally`** ou une macro `ILP_*` apparaisse pour que l'on ne puisse introduire du C au sein d'un programme Javascript. Un rare endroit où cela pouvait néanmoins être utilisé était la compilation des constantes sauf si la destination était (**`void`**) car ce n'est pas du Javascript !

2 Détails

2.1 Schéma de compilation

Attention aux boucles infinies dans les schémas de récursion ! Comme, par exemple, dans l'exemple fantaisiste suivant :

```
                        ----> d
                        qqch


{
  var tmp;

  ----> tmp
  qqch                  // boucle infinie!

  return tmp;
}
```

Lorsque le terme à réécrire doit être compilé avec une direction d, il est étonnant (et devrait faire l'objet d'un commentaire) que cette destination n'apparaisse pas dans le terme réécrit ! C'est le cas, par exemple, dans l'exemple fantaisiste ci-dessus. C'est analogue à écrire, en mathématiques, une proposition superfétatoirement quantifiée :
$\forall d, 1 = 1$

2.2 Visiteur

Un visiteur est une alternance de `visit` et `accept`. Ainsi pour effectuer une récursion, peut-on écrire :

```
public Object visit(IAST* iast, Object odata) {
   iast.get*().accept(this, otherdata);
}
```

Attention aussi aux boucles infinies telle que :

```
public Object visit(IAST* iast, Object odata) {
   iast.accept(this, odata);
}
```

Pour réussir l'examen, il suffisait de s'inspirer des méthodes `compile()` d'ILP4. De la lecture d'une telle méthode, on en déduisait le schéma de compilation qu'il suffisait alors de transcrire en Java. Les schémas de compilation donnés en cours sont ceux d'ILP2. Sont mentionnées, en ILP4, les modifications nécessaires (passage systématique par des variables temporaires intermédiaires pour toute expression non triviale) pour les adapter à ILP4.

Lorsque l'on avait, pour C, un appel récursif au compilateur tel que

```
getX().compile(buffer, newlexenv, newcommon, newdestination);
```

il faut le réécrire en :

```
Data newData = new Data(buffer, newlexenv, newcommon, newdestination);
iast.getX().accept(this, // le visiteur
                   newData );
```

2.3 Simplicité

Soyez simple. La compilation d'une constante ne nécessite pas de passer par une variable intermédiaire. Ainsi, est totalement surfait d'écrire :

```
                        --------> d
                        constante

{ var tmp;

  tmp = constante;

  d tmp;
}
```

2.4 Temps d'évaluation

Une des difficultés de la compilation est de bien gérer les divers temps d'évaluation. Ainsi, le schéma suivant pour l'alternative est-il bigrement faux :

```
                        ---------------------------------------------> d
                        alternative(condition, consequence, alternant)

var tmp1;
var tmp2;
var tmp3;

--------> tmp1
condition;

----------> tmp2
consequence;

--------> tmp3
alternant;

if ( tmp1 ) {
  d tmp2;
} else {
  d tmp3;
}
```

En effet, cela revient à toujours évaluer la branche `then` et **`else`** de l'alternative alors que l'on ne doit évaluer qu'une seule de ces branches selon la valeur de la condition.

Une erreur semblable serait d'évaluer le corps d'une fonction lors de sa définition avant même qu'elle soit appelée !

2.5 Autres considération

Une difficulté était le schéma de compilation de **`try`**/**`catch`**/**`finally`**. Si la destination du corps de **`try`** ne peut être que celle du tout, les destinations de **`catch`** et **`finally`** ne peuvent être que la poubelle. Pour vous en convaincre, imaginez que la destination est **`return`**.