Vérification du Déterminisme pour le Langage X10

Tomofumi Yuki (CSU/IRISA) Paul Feautrier (ENSL/INRIA)
Sanjay Rajopadhye (CSU) Vijay Saraswat (IBM)

ENS de Lyon
Paul.Feautrier@ens-lyon.fr

27 mars 2013

O mais c'est que, voyez-vous bien,
je n'ai point sujet d'être mécontent de mes polyèdres
A. Jarry

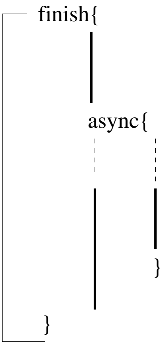
La crise du parallélisme

- La programmation parallèle devient obligatoire ...
- mais reste difficile.
- Deux nouveaux types de bugs :
  - Interblocages
  - Non déterminisme
- On peut en général éliminer l'un des risques, mais pas les deux, sous peine de perdre en expressivité
- X10 élimine les interblocages, mais permet le non-déterminisme, qu'il s'agit de détecter et de signaler.

Le langage X10

- développé à IBM Research (mais un clone à Rice sous la direction de Vivek Sarkar)
- dérivé de Java : langage à objet
- Partitioned Global Address Space (mais cet aspect ne sera pas développé dans cet exposé)
- parallélisme de contrôle par création d'*activités* (lightweight threads), hiérachique et possiblement récursif. L'acte de création d'une activité est une opération de première classe.
- synchronisation de type fork/join, ou par barrières (voir plus loin), ou par sections critiques, ou par "remote method invocation".

Parallelisme async/finish



Syntaxe :

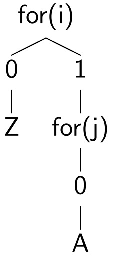
```
S ::= finish S
    | async S
    | {S; S}
    | for(x in exp .. exp} S
    | assignment
```

- async créé une *activité* (lightweight thread)
- analogie avec fork / wait
- la distinction global /local résulte de la visibilité des déclarations

Analyse du flot des données, version séquentielle

```
for(i in 0 .. n-1){
  A[i] = 0.0;    //Z
  for(j in 0 .. n-1)
    A[i] = A[i] + ... //M
}
```

Arbre de Syntaxe Abstraite (AST)



Quelle est la source de la valeur de `A[i]` lue à l'itération (i, j) de M (vecteur de position $[i, 1, j, 0]$) ?

Réponse : la source est l'opération la plus tardive qui écrit dans `A[i]` et qui précède $[i, 1, j, 0]$. Ce peut être une itération de Z ou une itération de M.

Ordre d'exécution

Pour calculer une source, il faut exprimer l'ordre d'exécution des opérations : c'est l'ordre lexicographique des vecteurs de position, noté $\prec$.

Exemple

$$\begin{aligned} < Z, i > \prec < M, i', j' > &\equiv [i, 0] \ll [i', 1, j'] \\ &\equiv i < i' \vee (i = i' \wedge 0 < 1) \\ &\equiv i \leq i'. \end{aligned}$$

- L'ordre $\prec$ est total,
- On compare toujours des symboles de même type : compte-tours ou nombres entiers,
- Il n'y a pas à comparer les longueurs des vecteurs.

Calcul de la source, I

Soit $W : A[f(x)] = \ldots$ une écriture et $R : \ldots = A[g(y)]$ une lecture du même tableau A.

- trouver la valeur maximale de x selon $\prec$ telle que :
- x et y sont des itérations légales : $x \in D_W;\ y \in D_R$.
- W et R accédent au même élément de A : $f(x) = g(y)$.
- $x \prec y$.

Il faut ensuite combiner les diverses possibilités pour W.

Calcul de la source, II

Méthode de résolution :

- Le domaine de x est une union de polyèdres (décomposition de $\ll$)
- Chaque polyèdre dépend du paramètre y
- On doit calculer le maximum lexicographique de chaque polyèdre, à l'aide du logiciel PIP. Le résultat est une expression conditionelle.

On doit ensuite combiner les différents candidats, à l'aide de règles de réécriture du genre :

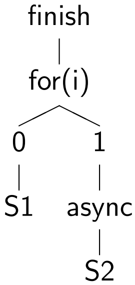
$$\begin{array}{rcl}
\max(\textbf{if } p \textbf{ then } a \textbf{ else } b, c) & = & \textbf{if } p \textbf{ then } \max(a, c) \textbf{ else } \max(b, c) \\
\textbf{if } p \textbf{ then } a \textbf{ else } a & = & a \\
\max(a, \perp) & = & a
\end{array}$$

où $\perp$ représente le "maximum de l'ensemble vide".

Le cas de X10 : ordre d'exécution

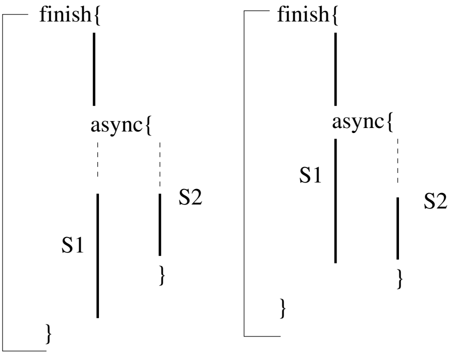
```
finish
  for(i in 0..n-1){
    S1;
    async
      S2;
  }
```

AST



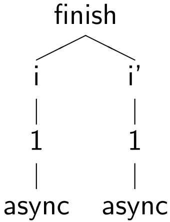
Position vectors : $S1 : [f, i, 0]$ $S2 : [f, i, 1, a]$

Indéterminisme



- L'ordre d'exécution dépend des décisions de l'ordonnanceur ou des performances des cœurs.
- La présence d'un `async` ne peut que retarder l'exécution de son contenu.

Ordre d'exécution de X10



- Exemple : comparer $[f, i, 1, a]$ et $[f, i', 1, a]$
- Ecrire l'ordre lexicographique :

$$\begin{aligned}
[f, i, 1, a] \quad \ll \quad & [f, i', 1, a] \equiv f < f \\
\vee \quad & (f = f \wedge i < i') \\
\vee \quad & (f = f \wedge i = i' \wedge 1 < 1) \\
\vee \quad & (f = f \wedge i = i' \wedge 1 = 1 \wedge a < a)
\end{aligned}$$

- Tous les termes sont faux sauf le second. Mais :
- Il y a un `async` non couvert par un `finish` dans la branche gauche, donc le terme doit être omis.

Les deux opérations sont incomparables et l'ordre d'exécution est partiel.

Analyse du flot des données pour X10

- On ne considère que les programmes X10 polyèdriques
- La définition de l'ensemble des sources potentielles, E reste la même
- mais comme l'ordre d'exécution est partiel, il peut ne pas exister un maximum unique
- on doit utiliser le concept d'extrema :

$$\overline{E} = \{x \in E | \neg \exists y : x \prec y\}$$

- $\overline{E}$ n'est pas obligatoirement un singleton, il peut y avoir plusieurs sources, donc indéterminisme.

Implémentation

En principe :

- Construire l'ensemble E. Pour un programme polyèdrique, c'est une union de polyèdres (ou une formule booléenne affine).
- Dans la définition de $\overline{E}$, éliminer la variable quantifiée y.
- Il existe des outils généraux d'élimination de quantificateurs ...
- mais on peut réduire la complexité en décomposant E en polyèdres et en utilisant des bibliothèques comme PIP ou `isl`.

Il existe un démonstrateur
`http://polyweb.irisa.fr/x10p/index.php`

Hasards

Quand il y a plusieurs sources possibles, on dit que le programme a un *hasard* (*race* en Anglais).
Classification :

- Si l'écriture et la lecture ne sont pas comparable pour $\prec$ c'est presque surement un bug, car la lecture peut accéder à une cellule non initialisée.
- S'il y a ambiguité entre plusieurs écritures, c'est peut être un bug, mais l'ambiguité peut être levée :
  - par une écriture postérieure qui écrase la valeur ambigue
  - par des considérations sémantiques

Dans tous les cas, le dernier mot doit rester au programmeur.

Les horloges de X10

Les horloges (*clocks*) sont une variante plus souple des barrières classiques. Un programme X10 peut exploiter plusieurs barrières. Principe :

- à un instant donné, plusieurs activités peuvent être rattachées à une même horloge
- une activité qui exécute `advance();` se bloque jusqu'à ce que toutes les activités rattachées aient fait de même.
- à ce moment, toute les activités redémarrent.

Il existe deux syntaxes.

Syntaxe explicite

Les horloges sont des objets de première classe, qui peuvent être manipulées par les outils du langage (à quelques exceptions près).

```
Clock c = Clock.make();
...
Clock d = c;
...
d.advance();
```

La vérification de la syntaxe explicite est très difficile, peut exiger une analyse *points-to* imprécise, et ne sera pas abordée aujourd'hui.

Syntaxe implicite

Les horloges n'ont pas de nom, et sont gérées d'aprés le contexte

```
clocked finish{
    ...
    clocked async{
        ...
        advance();
        ...
    }
    ...
    advance();
}
```

- l'activité principale créé une horloge par `clocked finish` à laquelle elle est rattachée
- `clocked async` crée une activité rattachée à l'horloge la plus proche
- les deux activités se synchronisent par `advance();`
- les deux activités atteignent leur accolades finales, l'horloge est détruite.

Dans ce qui suit, on ne considerera que les programmes à une seule horloge. Nous conjecturons que le cas général peut se réduire à ce cas particulier.

Le compteur d'activation

On peut visualiser le fonctionnement d'une horloge de la façon suivante :

- Chaque activité (qui ne peut être rattachée qu'à une seule horloge) entretien un compteur qui est incrémenté de 1 à chaque `advance()` ;
- Les activités rattachées ne peuvent "passer la barrière" que si tous les compteurs sont égaux
- L'implémentation peut évidemment être différente.
- On peut étendre la notion de compteur d'activation à toutes les opérations du programme
- u étant le vecteur de position d'une opération, on note $\phi(u)$ la valeur courante du compteur d'activation.
- Si $\phi(u) < \phi(v)$, alors u est exécutée avant v, même si ces deux opérations ne sont pas dans la même activité.

Ordre d'exécution avec horloges

- Si le programme est polyédrique, la valeur du compteur peut être calculée statiquement par des techniques classiques (E. Ehrhart, M. Brion). C'est une fonction (en général un polynome) du vecteur de position.
- L'ordre d'exécution est la fermeture transitive de l'union de l'ordre $\prec$ et de la relation $\phi(u) < \phi(v)$, qui se simplifie en trois cas :

$$\begin{array}{rcl} u & \prec & v, \\ \phi(u) & < & \phi(v) \\ u \prec u' & , & \phi(u') = \phi(v), \end{array}$$

Dans le dernier cas, v doit être une `advance();`.

Déterminisme

Plutôt que d'étendre une analyse du flot de donnée au cas des horloges, on propose de décider quels sont les hasards que les horloges éliminent.

Soit u et v deux instances qui engendrent un hasard.

- Il existe une relation polyédrique $H(u, v)$ qui exprime l'existence d'un hasard
- Il est impossible que $u \prec v$ ou $v \prec u$, et u et v ne sont pas des `advance()` ;
- Le hasard ne subsiste que si le systeme :

$$\begin{array}{ccc} \phi(u) & = & \phi(v) \\ & H(u, v) & \end{array}$$

est satisfiable

Méthodes de résolution

- Les ϕs peuvent être des polynomes, et les variables sont entières ; donc le problème est probablement indécidable (10ème problème de Hilbert)
- Utiliser des heuristiques : résoudre en rééls (Z3 ou Quepcad), puis appliquer le test du pgcd
- Cas particulier : les ϕ sont linéaires
- Autres heuristiques, soit générales (voir Z3) soit propres à X10 (cas des polynomes identiques).

Un exemple, I

Soit deux activités dont l'une crée un *stream* de valeurs que l'autre exploite. Première solution :

```
finish{                              async{
  async{                               for(i in 0..n-1)
    for(i in 0..n-1)                     ... = ... x ...; //R
      x = ...;  //W                    }
  }                                  }
```

Le programme est incorrect : la lecture et l'écriture peuvent s'exécuter en parallèle.

Un exemple, II

Deuxième solution : on introduit une horloge :

```
clocked finish{
  clocked async{
    for(i in 0..n-1){
      x = ...;      //W
      advance();
    }
  }
                                 clocked async{
                                   for(i in 0..n-1){
                                     advance();
                                     ... = ... x ...; //R
                                   }
                                 }
}
```

On trouve $\phi(W, i) = i$ et $\phi(R, i') = i' + 1$. L'équation $\phi(W, i) = \phi(R, i')$ a pour solution $i = i' + 1$, donc l'ambiguité n'est pas levée.

Un exemple, III

Troisième solution :

```
clocked finish{
  clocked async{
    for(i in 0..n-1){
      advance();
      x = ...;          //W
      advance();
    }
  }
```

```
clocked async{
  for(i in 0..n-1){
    advance();
    advance();
    ... = ... x ...; //R
  }
}
}
```

$\phi(W, i) = 2i + 1$, $\phi(R, i') = 2i' + 2$, l'équation $\phi(W, i) = \phi(R, i')$ n'a pas de solution *entière* (c'est le test du pgcd), le programme est déterministe.

Noter que le test du pgcd s'applique aussi bien à un polynome qu'à une forme linéaire.

Conclusion

- L'analyse `async / finish` est correcte et complète pour un programme polyèdrique
- L'introduction des horloges rend l'analyse incorrecte (il peut y avoir des faux signalements)
- Il en sera de même si on tente d'élargir le modèle polyédrique (par exemple, `advance();` sous condition)

Il faudra traiter les autres traits de X10 : `atomic` et `at`.
Y a-t-il d'autres applications du modèle polyédrique à X10 : recherche de parallélisme supplémentaire, transformations de programmes, tuilage ?