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

7 juin 2013

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

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 :

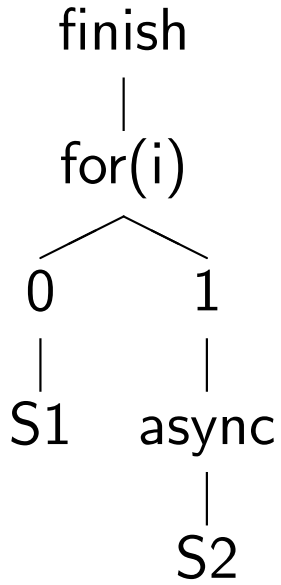
```
S ::= finish S
    | async S
    | {S; S}
    | for(x in exp .. exp} S
    | assignment
```

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

Le cas de X10 : ordre d'exécution

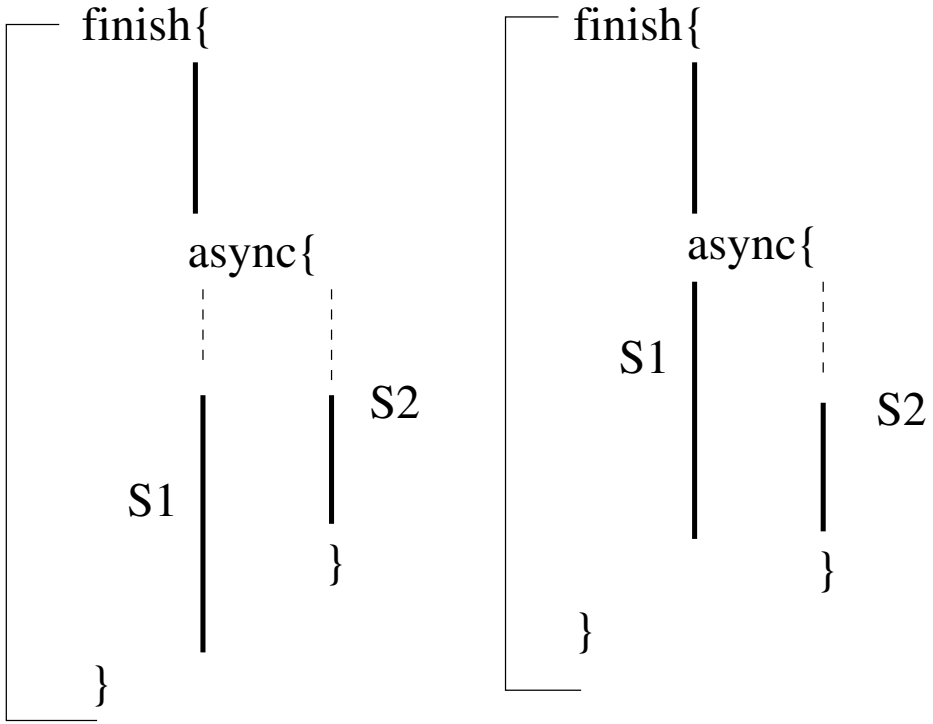
```
finish
  for(i in 0..n-1){
    S1;
    async
      S2;
  }
```

AST



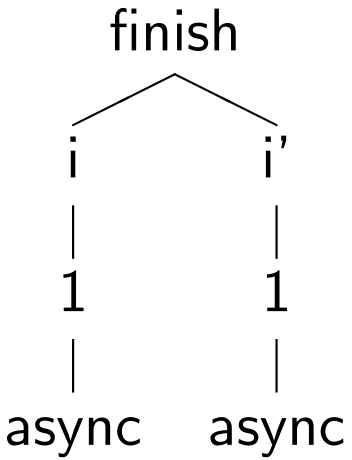
Position vectors : $S1 : [f, i, 0]$ $S2 : [f, i, 1, a]$

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.

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 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();`.

FIXME : more

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, III

Troisième solution :

```
clocked finish{
  int x;
  clocked async{
    for(i in 0..n-1){
      advance();
      x = F();          //W
      advance();
    }
  }
                                  clocked async{
                                    for(i in 0..n-1){
                                      advance();
                                      advance();
                                      G(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, mais il n'a plus de parallélisme.
Noter que le test du pgcd s'applique aussi bien à un polynome qu'à une forme linéaire.

Eliminating clocks

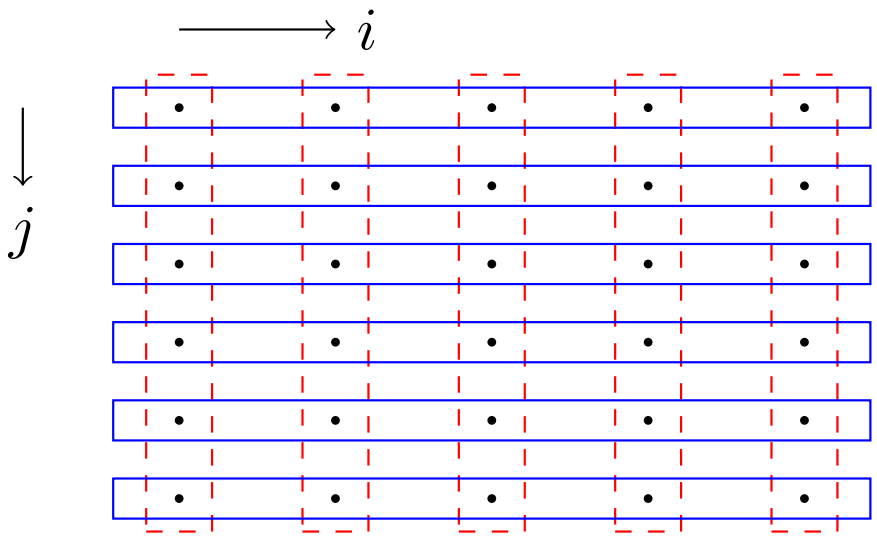
Motivation

- Curiosity : X10 has two kinds of barriers
  1. `advance()`
  2. end of `finish`

  Are they interchangeable ?
- Efficiency :
  - creating activities is cheap, clock management is expensive
  - reducing overhead $\rightarrow$ target finer grain activities
- Enabling scheduler optimizations :
  - task-based parallelism
  - end-of-life synchronization only

Idea

- Rewrite the program in order of increasing "dates"



- Replace successive "phases" of "`S...;advance()`" sequences with `finish` blocks ($\rightarrow$ use only end-of-`finish` barriers)

Assumptions

- Only one clock at a time (clocked `finish`/`async`)
- As many clocked `async`s and `advance`s as necessary
- Nested `finish` allowed

General solution

- Iterate on time steps

```
for ( d=0 ; <some work is left> ; d++ )
    finish {
        execute a copy of the program where
        - "advance" becomes "++φ"
        - "S;" becomes "if (d == φ) S;"
    }
```

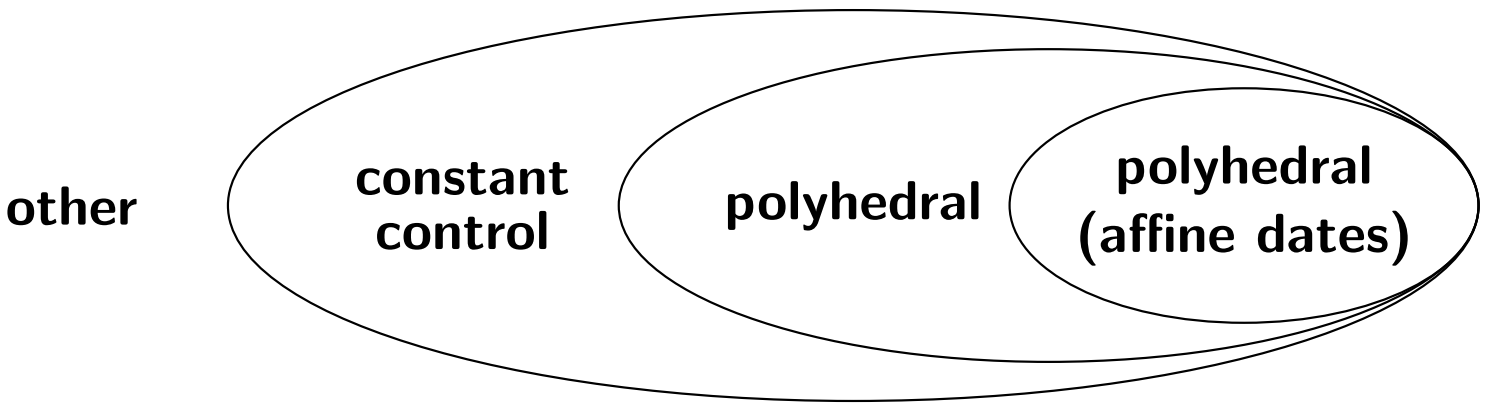
- Requires :
  - maintaining clock values (`advance` $\rightarrow$ ++ϕ)
    $\rightarrow$ repeatedly reproducing control
  - changing state at approriate times (`S` $\rightarrow$ `if (d ==` ϕ`) S`)
    $\rightarrow$ conditionally reproducing effects
- Valid only on *constant control* programs
  - known start state
  - no side-effect in control (loop and branch conditions)

Correctness

- On constant control programs :
- All instructions are "executed" (including `advances`)
- Take any two instructions instances S and T :
  - if $\phi(S) < \phi(T)$, they are executed during distinct iterations of the outermost loop (on `d`, which is monotonically increasing)
  - if $\phi(S) = \phi(T)$, they are executed during the same iteration of the outermost loop, but in original program order

Variations

- The general solution may not be the most efficient...
- but there are sub-classes of program that can be optimized
- The space we consider is :



- Polyhedral programs are made of :
  - loops with affine bounds
  - branches on affine conditions
  - arbitrary instructions

Constant control

```
finish {
  async {

    for(i in 0..N-1) {
      S0(i);
      if (a0[i] > 0) advance;
  } }
  async {

    for(i in 0..N-1) {
      S1(i);
      if (a1[i] > 0) advance;
} } }
```

- Input dependent synchronization

Constant control

```
for ( d=0 ; ??? ; d++ ) {

  finish {
    async {

      for(i in 0..N-1) {
        S0(i);
        if (a0[i] > 0) advance;
    } }
    async {

      for(i in 0..N-1) {
        S1(i);
        if (a1[i] > 0) advance;
  } } }
}
```

- Input dependent synchronization
- Wrap a loop around the block

Constant control

```
for ( d=0 ; ??? ; d++ ) {
  int φ = 0;
  finish {
    async {
      int φ0 = φ;
      for(i in 0..N-1) {
        S0(i);
        if (a0[i] > 0) advance ++φ0;
    } }
    async {
      int φ1 = φ;
      for(i in 0..N-1) {
        S1(i);
        if (a1[i] > 0) advance ++φ1;
  } } }
}
```

- Input dependent synchronization
- Wrap a loop around the block
- Maintain date (local dates capture enclosing date on startup)

Constant control

```
for ( d=0 ; ??? ; d++ ) {
  int φ = 0;
  finish {
    async {
      int φ0 = φ;
      for(i in 0..N-1) {
        S0(i) if (d == φ0) S0(i);
        if (a0[i] > 0) ++φ0;
    } }
    async {
      int φ1 = φ;
      for(i in 0..N-1) {
        S1(i) if (d == φ1) S1(i);
        if (a1[i] > 0) ++φ1;
  } } }
}
```

- Input dependent synchronization
- Wrap a loop around the block
- Maintain date (local dates capture enclosing date on startup)
- Guard statements

Constant control

```
for ( d=0 ; ??? ; d++ ) {
  int φ = 0;
  finish {
    async {
      int φ0 = φ;
      for(i in 0..N-1) {
        if (d == φ0) S0(i);
        if (a0[i] > 0) ++φ0;
    } }
    async {
      int φ1 = φ;
      for(i in 0..N-1) {
        if (d == φ1) S1(i);
        if (a1[i] > 0) ++φ1;
  } } }
}
```

- Input dependent synchronization
- Wrap a loop around the block
- Maintain date (local dates capture enclosing date on startup)
- Guard statements
- Flag any action (either `Si` or `++`ϕ), go on as long there is some

Polyhedral programs I

- Nested loops with affine bounds, arbitrary instructions
- Dates are polynomials over counters and parameters
- Example :

```
for ( i in 0..(N-1) ) {
  async {
    for ( j in 0..(M-1) ) {
      S0(i,j);
      for ( k in 0..(i-1) ) {
        S1(i,j,k);
        advance();
      }
    }
  }
  advance();
}
```

Polyhedral programs I

- Nested loops with affine bounds, arbitrary instructions
- Dates are polynomials over counters and parameters
- Example :

```
for ( i in 0..(N-1) ) {
  async { // φ = i
    for ( j in 0..(M-1) ) {
      S0(i,j); // φ = i + (j(j-1)/2
      for ( k in 0..(i-1) ) {
        S1(i,j,k); // φ =  i + j(j-1)/2 + k
        advance();
      }
    }
  }
  advance();
}
```

Polyhedral programs II

Polyhedral programs with affine dates I

- When all times are affine, instructions+dates can be manipulated as (abstract) polyhedra
- Example :

```
finish {
  for ( i in 0..(N-1) ) {
    async
      for ( j in i..(N-1) ) {
        S0(i,j);
        advance();
      }
    advance
    async
      for ( j in 0..(i-1) ) {
        S1(i,j);
        advance();
      }
  }
}
```

Polyhedral programs with affine dates I

- When all times are affine, instructions+dates can be manipulated as (abstract) polyhedra
- Example :

```
finish {
  for ( i in 0..(N-1) ) {
    async // φ = i
      for ( j in i..(N-1) ) {
        S0(i,j); // φ = i+j-i = j
        advance();
      }
    advance
    async // φ = i+1
      for ( j in 0..(i-1) ) {
        S1(i,j); // φ = i+1+j
        advance();
      }
  }
}
```

Polyhedral programs with affine dates I

- When all times are affine, instructions+dates can be manipulated as (abstract) polyhedra
- Example :

```
finish {
  for ( i in 0..(N-1) ) {
    async // φ = i
      for ( j in i..(N-1) ) {
        S0(i,j); // φ = i+j-i = j
        advance();
      }
    advance
    async // φ = i+1
      for ( j in 0..(i-1) ) {
        S1(i,j); // φ = i+1+j
        advance();
      }
  }
}
```

```
S0 := [N] ->
      {[i,j,t,0]:
       0<=i<N and
       i<=j<N and
       t=j };
S1 := [N] ->
      {[i,j,t,1]:
       0<=i<N and
       0<=j<i and
       t=i+j+1 };

M := {[i,j,t,p]
       -> [t,i,j,p]};
R := M(S0+S1);
codegen R;
```

Polyhedral programs with affine dates II

- Final result provided by CLoog (+ `finish` around d-iterations, + async around statements) :

```
for (int d = 0; d <= N - 1; d += 1)
  finish
    for (int c1 = 0; c1 <= d; c1 += 1) {
      if (d >= c1 + 1 && 2 * c1 >= d)
        async S1(c1, d - c1 - 1);
      async S0(c1, d);
    }
for (int d = N; d <= 2 * N - 2; d += 1)
  finish
    for (int c1 = d - d / 2; c1 <= N - 1; c1 += 1)
      async S1(c1, d - c1 - 1);
```

- Note : 2D $\rightarrow$ 2D

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 ?