ALIDECS
Programmation réactive

Frédéric Boussinot
projet MIMOSA, Inria-Sophia

http://www.inria.fr/mimosa/rp

juin 2006

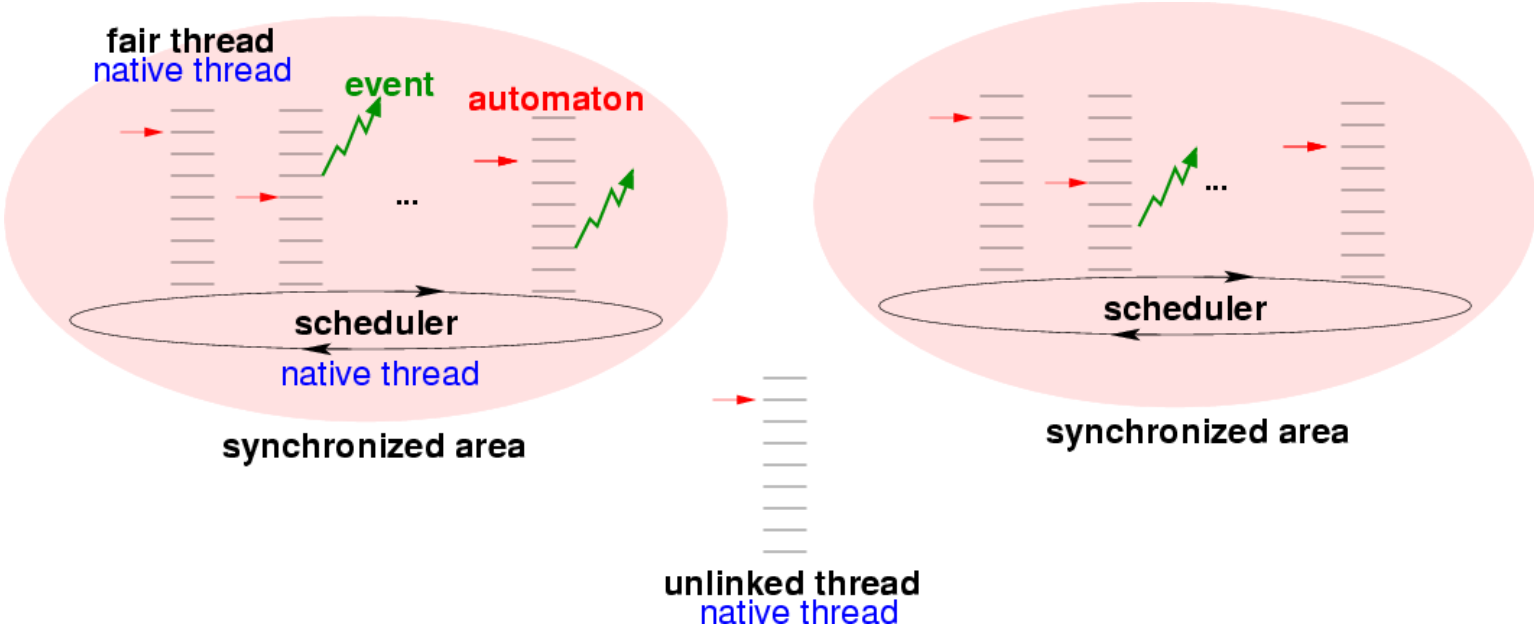
Travail en commun avec Frédéric Dabrowski

Plan

- Rappel : le modèle des FairThreads
- La réactivité dans ce modèle
- Proposition de langage
- Analyses statiques assurant la réactivité
- Conclusion

Le modèle des FairThreads

- Threads coopératifs + instants + événements diffusés
- Possibilité d'exécution asynchrone (non-coopérative)



1. Assurer la coopération
2. Préserver l'atomicité entre deux coopérations
3. Assurer la réactivité

Objectifs

- Assurer la coopération = passage des instants
  - Terminaison des atomes (appels de fonctions)
  - Absence de boucle instantanée
- Préserver l'atomicité entre 2 coopérations
  - Étanchéité mémoire : distinction public/privé
- Assurer la réactivité
  - Absence d'erreur à l'exécution
  - Borne sur le nombre de threads actifs simultanément
  - Borne asymptotique sur la taille des données créées (mémoire bornée)
- Bornes polynomiales sur l'utilisation des ressources (thèse de F. Dabrowski)

Atomicité entre 2 coopérations

- Types inductifs - Fonctions premier ordre
- Références - Pas d'événement
- Un seul scheduler + instruction `unlink`

$$
\begin{array}{lcll}
p & ::= & x \mid c(p,\ldots,p) & (patterns) \\
e & ::= & x \mid c(e,\ldots,e) \mid f(e,\ldots,e) & \\
 & & \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{ref}_{\rho}(e) \mid \texttt{get}(e) \mid \texttt{set}(e,e) & \\
 & & \mid \texttt{thread}\{f(e,\ldots,e)\} \mid \texttt{unlink}\{e\} \mid l & (expressions)
\end{array}
$$

- Coopération : `unlink` $\{()\}$ - Séquence : `let`
- Régions associées aux références (approximation)
- Référence accédée sous un `unlink` : région $\rightarrow$ privé. Référence globale : région $\rightarrow$ public

Système de types et d'effets

Le système de types vérifie que :

1. Aucun accès (lecture ou écriture) à une référence publique n'a lieu dans une instruction `unlink` (y compris à travers un appel de fonction)
2. Aucune référence privée n'est pointée (directement ou indirectement) par une référence publique

Programme correct : atomicité entre deux coopérations + étanchéité entre les mémoires privées des threads

Cooperative Threads and Preemptive Computations,
http://hal.inria.fr/inria-00078780

Proposition de langage : FunLoft

On ajoute au langage précédent :

- Types polymorphes + Inférence de types
- Tableaux de références (taille statique)
- Récursivité bien fondée sur les types inductifs
- Événements (`await`, `generate`, `for_all_values`)
- Distinction fonction/module (efficacité : automates)
- Pas de module récursif, mais `loop` + `repeat`
- Attente de terminaison de threads créés (`join` e)
- `stop`, `suspend`, `resume`

Vérifications

- Terminaison des fonctions (diminution de la taille des paramètres + ordre lexicographique) :

```
type 'a list = Nil | Cons of 'a * 'a list
let length (l) = match l with Nil -> 0
| Cons (h,t) -> 1+length (t) end

let fact (n) = if n = 0 then 1 else n*fact (n-1)
```

- Absence de boucle instantanée (contrôle + événements) :

```
let module ok () =
loop begin print_string ("cycle "); cooperate end

let module bug1 () = loop print_string ("cycle ")

let module bug2 (e) = for_all e with x -> generate (e,...)
```

Vérifications - 2

- Absence d'erreur à l'exécution : test de la division par 0.
  Tableaux cycliques : `a[i] = a[i mod size(a)]`
- Exécution asynchrone en mémoire privée uniquement :

```
let module bug3 (r) = unlink r:=1

let module bug4 (r) = let x=r in unlink x:=1

let module ok (r) = let x=ref !r in unlink x:=1
```

- Pas de thread créé dans une boucle sans `join` :

```
let module bug5 (r) = loop begin thread m (); cooperate end

let module ok5 (r) =
loop begin join thread m (); cooperate end
```

Vérifications - 3

- Contrôle de la taille des données créées : stratification des références de types inductifs : $r1 :=!r2$ entraîne $level(r1) < level(r2)$ - Absence de cycle

```
let module bug6 () =
  let r = ref Nil in
     loop begin r := Cons (0,!r); cooperate end
```

- La stratification doit prendre en compte les événements

```
let module bug7 () =
  let e = event in
  let v = ref Z in
     begin
        generate (e,S(!v));
        for_all_values e with x -> v:=x
     end
```

Bornes polynomiales

```
let explode1 (l) = match l with Nil -> ()
| Cons (h,t) -> begin explode1 (t); explode1 (t) end
```

Le calcul de `explode1` prend un temps exponentiel

```
let explode2 (l) = match l with Nil -> ()
| Cons (h,t) -> begin thread m (h); explode2 (t); explode2 (t)
end
```

Avec un argument de taille n, `explode2` crée 2^n threads.

- Idée : contrôler la taille polynomiale des données + contrôler la linéarité de l'usage des paramètres lors des appels (pas de duplication de paramètre dans les appels récursifs)

Conclusion

1. Séparation mémoire assurant l'atomicité entre 2 coopérations
2. Terminaison des fonctions ⇒ borne sur les ressources utilisées lors du calcul des fonctions
3. Stratification ⇒ borne sur la taille des données utilisées pendant plusieurs instants
4. Synchronisation des threads créés cycliquement (`join`) ⇒ borne sur le nombre de threads pouvant s'exécuter simultanément

(2)+(3)+(4) : borne sur la mémoire globale utilisée et sur la durée des instants = réactivité

- Prototype en cours de réalisation (traduction en Loft+C, GC de H. Boehm)
- Automates cellulaires. Proies/prédateurs