Programmation réactive

Frédéric Boussinot
projet MIMOSA, Inria-Sophia

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

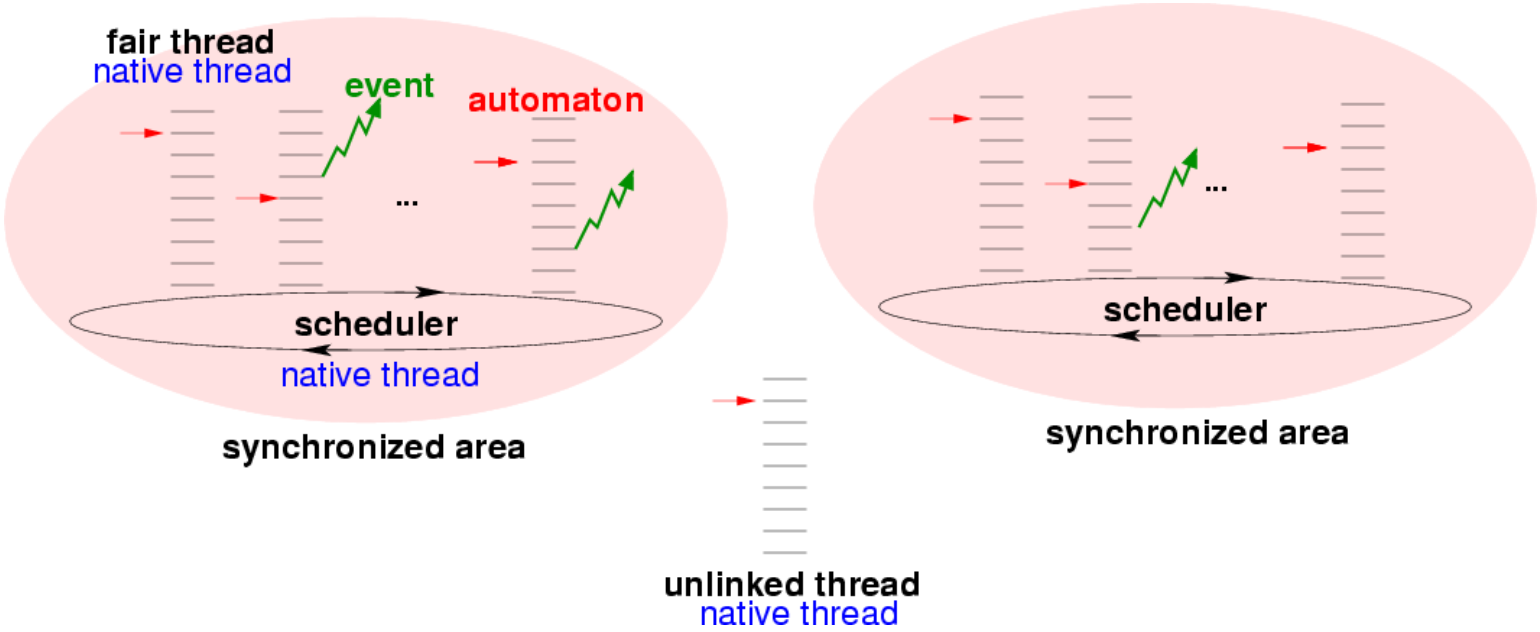
février 2005

Plan

- FairThreads, LOFT
- Analyse statique de terminaison
- Utilisation d'un langage "*sûr*"

FairThreads en C

- Threads coopératifs + instants + événements diffusés
- Scheduling avec un ordre fixe d'exécution



- LOFT : *Language over Fair Threads*

Objectifs dans ALIDECS

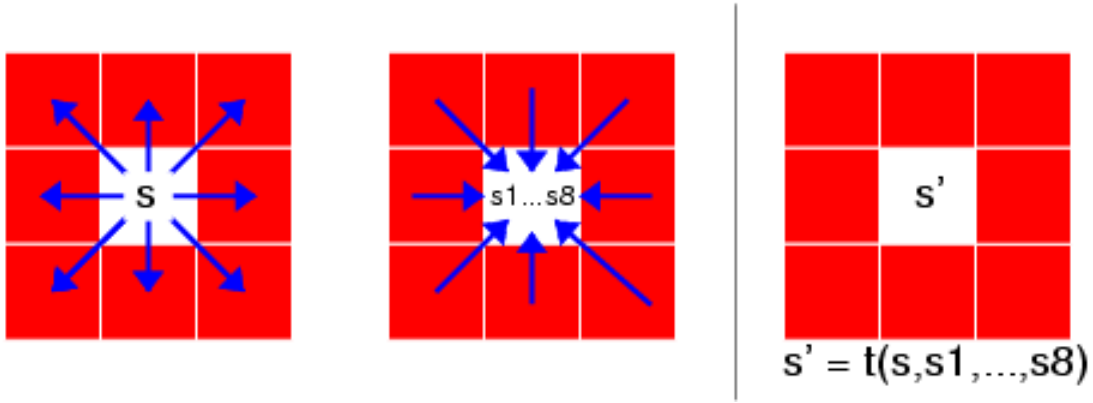
- Aspect langage
  - Prog. concurrente par threads : besoin de sémantique
  - FairThreads = C + librairie : API insuffisante = besoin de syntaxe
  - Langage LOFT : besoin de sécurité

  1. Assurer la coopération (absence de bouclage instantané)
     - Atomes (restriction de C + typage ? F. Dabrowski; Contrôle dynamique ?)
     - Passage des instants : automates (?), analyse statique
  2. Absence d'erreur à l'exécution : langage sûr

- Aspect réutilisation de code : threads POSIX implémentés en réactif

Automates cellulaires

Remontent aux années 50 (von Neumann, Ulam) : grille de cellules, voisinage invariant de chaque cellule, ensemble fini d'états des cellules et règles de transitions locales



- Parallélisme + discrétisation du temps + déterminisme
- *Jeu de la vie* (Conway) : modèle Turing-complet avec règles symétriques

Code d'une cellule d'automate cellulaire

```
module cell (int x, int y, event_t activation, int status, int state)
local neighborhood_t neighborhood, info_t info,
          image_t image, int counter, int more;
  initialize_cell (self ());
  while (1) do
      if (local(status) == QUIESCENT) then
          await (local(activation));
      else
          activate_neighborhood (self ());
      end
      {
          local(more) = 1;
          local(counter) = 0;
          clear (local(neighborhood));
      }
```

```
      while (local(more)) do
        get_value (local(activation),local(counter),(void**)&local(info));
        {
            if (return_code () == OK) {
               set_neighbor (local(neighborhood),local(info));
               local(counter)++;
            } else local(more) = 0;
        }
      end
      cell_behavior (self ());
  end
end module
```

Coopération = capture de toutes les valeurs

Analyse statique

Changements syntaxiques : `loop`, `get_all_values`

```
module cell (int x, int y, event_t activation, int status, int state)
local neighborhood_t neighborhood, image_t image;
   initialize_cell (self ());
   loop
        if (local(status) == QUIESCENT) then await (local(activation));
        else activate_neighborhood (self ()); end
        clear (local(neighborhood));
        get_all_values (local(activation),callback);
        cell_behavior (self ());
   end
end module
```

Rejet des boucles instantanées par analyse statique

Cyclone = variante *Safe* de C

- Développé à Cornell et ATT (Dan Grossman, Trevor Jim, Greg Morrisett)
- C - Cyclone - ML
- Cyclone = C
  - \+ contrôle des accès aux pointeurs
  - \- arithmétique de pointeurs
  - \+ GC
  - \+ contrôle des bornes des tableaux : exceptions
  - \+ polymorphisme
  - \+ inférence de types (`let`)
- Pas de construction dynamique de fonctions (pas de lambda)

Scheduling en Cyclone

```
struct _scheduler_t
{
   item_list_t<thread_t> linked;
   thread_t current;
   long instant;
   ...
};

typedef void *@region(`H) arg_t;
typedef char (*automaton_t) (arg_t);

struct _thread_t
{
   automaton_t  automaton;
   arg_t        locals;
   char          state;
   ...
};
```

Scheduling en Cyclone - 2

```
struct _t_data_t {...}
char t_automaton (t_data_t _local_vars) {...}

extern "C" automaton_t t_to_automaton (char (*f)(t_data_t));

thread_t t_create_in (scheduler_t _sched)
{
   let _locals = new _t_data_t {0,0};
   let aut = t_to_automaton(t_automaton);
   let _thread = make_thread (_sched,aut,NULL,(arg_t)_locals);
   link_to_scheduler (_sched,_thread);
   return _thread;
}
```

Pb: liste d'éléments de types différents ?

Conclusion

- Implémentation de la détection des boucles instantanées
- Sureté de LOFT + Cyclone reste à prouver...
- NB. Solution en ML en utilisant la création dynamique de fonctions :

$$\lambda(x : \text{`}a\ ref).\lambda(f : \text{`}a \to \text{`}a) \to \lambda(() : unit).x := (f\ !x)$$