Reactive Programming and FairThreads

Frédéric Boussinot

MIMOSA Project, Inria-Sophia

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

April 2007

Summary

1. Reactive programming objectives
2. The FairThreads model and the FunLoft language
3. Cellular automata
4. Use of multicore machines

Reactive programming objectives

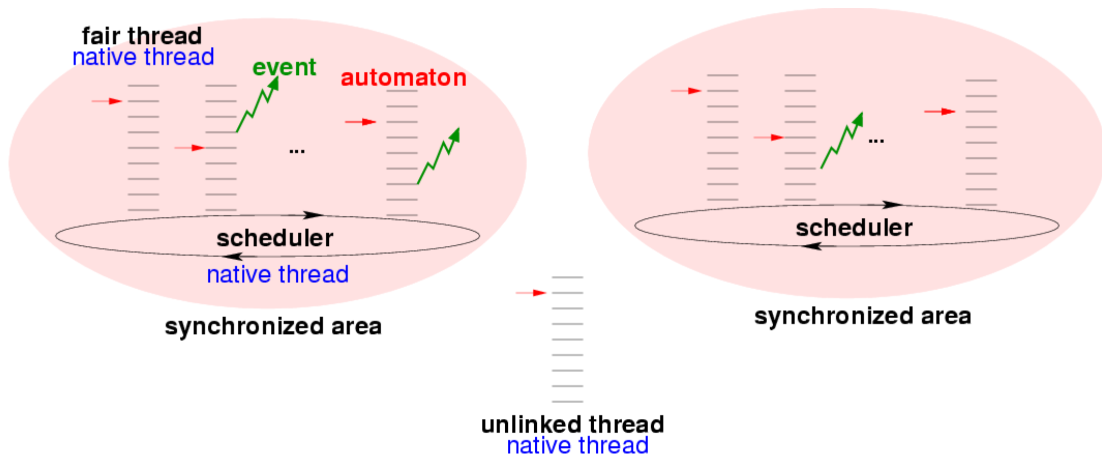
- Concurrent programming with clear and precise semantics (compared to Pthreads, for example)
- Static analysis to ensure general properties such as safety, absence of memory leaks, or absence of data-races
- Efficient implementations (large number of components, multicore)

Application domains:

- Simulations of systems made of large numbers of interacting entities (Physics, games)
- Embedded systems
- Migration-based systems

The FairThreads Model

- Threads linked to a scheduler are run cooperatively and share the same instants
- Several schedulers run asynchronously - Thread migration



- Implementations: Java (restriction to a unique scheduler), Scheme (with specialised service threads), library of FairThreads in C, LOFT.

Work in Progress: FunLoft

- Inductive data types - First order functions
  - Termination detection of recursively defined functions.
    Consequence: termination of instants ("reactivity")
- Restriction on the flow of data (stratification) carried by references and events.
  Consequence: bounded system size = absence of memory leaks
- Separation of references (using a type and effect system):
  - Schedulers own references shared by threads linked to them
  - Threads own private references only accessible by them
  - Consequence: atomicity of the cooperative model extended to unlinked threads and to multi-schedulers = absence of data-races

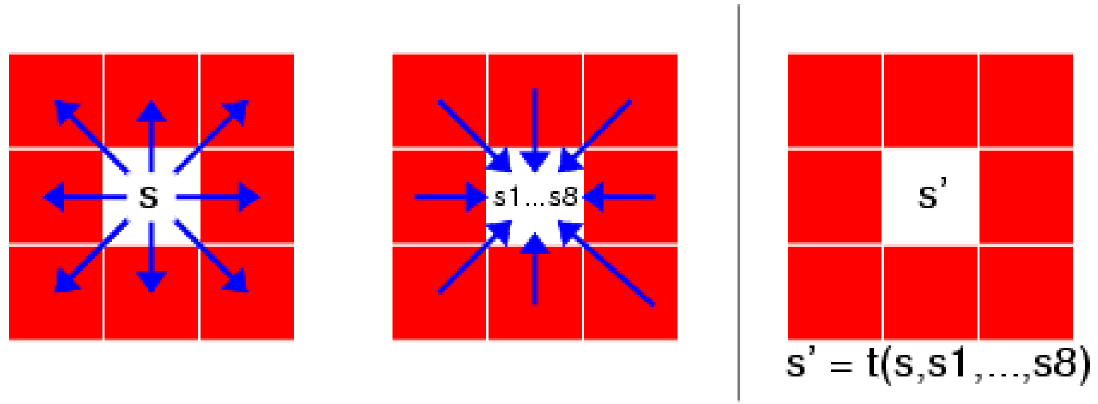
FunLoft Abstract Syntax

$$
\begin{array}{lcl}
p & ::= & x \mid C(p, \ldots, p) \\
e & ::= & x \mid C(e, \ldots, e) \mid f(e, \ldots, e) \\
& & \mid \texttt{match}\ x\ \texttt{with}\ p -> e \mid \ldots \mid p -> e \\
& & \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{ref}\ e \mid\ !e \mid e{:=}e \\
& & \mid \texttt{cooperate} \mid \texttt{thread}\ f(e, \ldots, e) \mid \texttt{join}\ e \mid \texttt{stop}\ e \\
& & \mid \texttt{unlink}\ e \mid \texttt{link}\ s\ \texttt{do}\ e \\
& & \mid \texttt{event} \mid \texttt{generate}\ e\ \texttt{with}\ e \mid \texttt{await}\ e \mid \texttt{get\_all\_values}\ e\ \texttt{in}\ e \\
& & \mid \texttt{loop}\ e \mid \texttt{while}\ e\ \texttt{do}\ e
\end{array}
$$

- functions defined by recursion at top-level
- schedulers defined at top-level
- function/module (functions terminate instantly, modules not)

Cellular automata

From the 50's (von Neumann, Ulam): grid of cells, fixed neighbourhood for each cell, finite number of possible states for each cell and transition rules defined locally



- Parallelism + discrete time + determinism
- *Game of Life* (Conway) :
  dead cell + 3 living neighbours → living;
  living cell + neighbours ≠ 2,3 → dead

Coding a Cell in FunLoft

```
let module linked_cell (x,y,me,state,neighbours) =
  let count = ref 0 in
  let living = ref state in
    begin
      generate ready;
      await starting_event;
      loop begin
        cell_display (x,y,!living,color);
        if !living then awake (neighbours) else await me;
        count := 0;
        for_all_values me with _ -> count++;
        gol_strategy (living,!count);
      end
    end
```

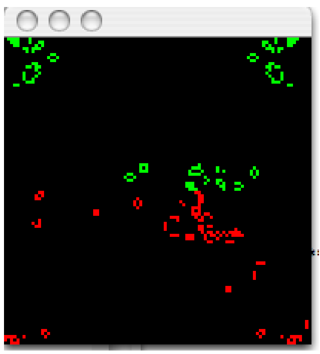
Multicore Programming

- How can a single application benefit from a multicore architecture? Solution: multithreading
- Benchmark: *Game Of Life (GOL)* divided into several synchronised areas: one native thread per area. Strong synchronisation. Global determinism.
- At language level: Synchronised schedulers
  - no sharing of memory (to avoid data races)
  - events: shared among synchronised schedulers
  - syntax:

```
let s1 = scheduler
and s2 = scheduler
```

Multithreaded GOL

- Main differences with a unique scheduler solution:
  - Drawing orders sent to the graphical thread
  - No global array of cells
  - Synchronised start of cells
- Difficult to get full benefit from multicore:
  - multi-threaded malloc
  - multi-threaded GC (H. Boehm's GC)
- Demo (10K cells, 500 instants, 1K cycles)



one scheduler	two schedulers
real 0m26.367s	real 0m20.944s
user 0m24.991s	user 0m26.548s
sys 0m0.381s	sys 0m0.626s

Conclusion

FunLoft provides:

- concurrent programming with clear semantics
- static analyses to prevent from data-races and memory leaks
- efficient implementation: large number of components
- syntax for multithreaded applications on multicore architectures

FunLoft is experimental:

- formalisation yet to achieve: type inference, join primitive, synchronised schedulers
- rough implementation: Loft-C, pthreads, Boehm's GC