FunLoft

Frédéric Boussinot
MIMOSA Project, EMP/INRIA Sophia-Antipolis, France

ANR-2008-DEFI-PARTOUT

January 9, 2009

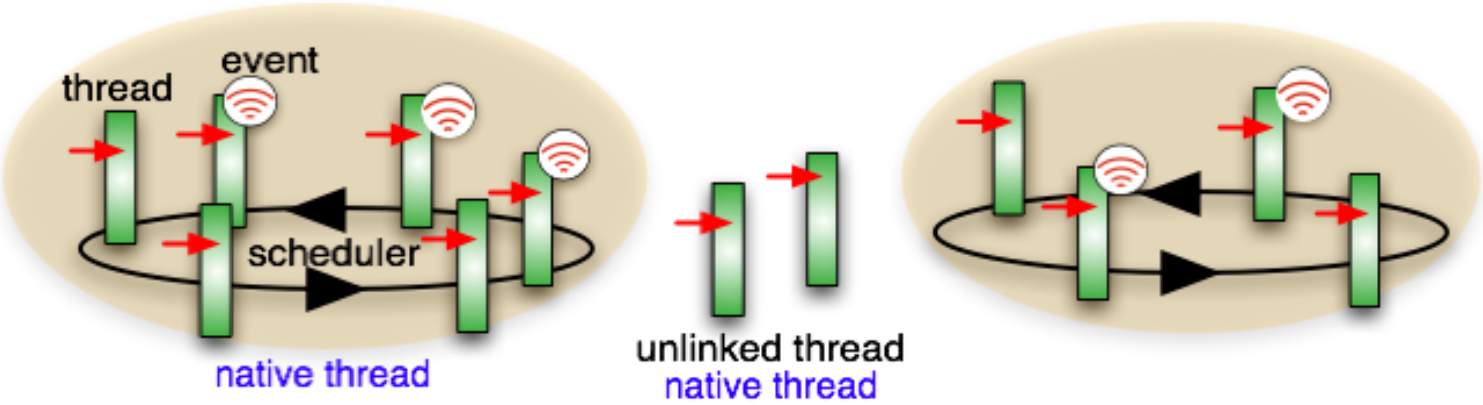
Plan

1. Mixing preemption & cooperation: the FairThreads model
2. Safe reactive programming: the FunLoft language
3. Colliding particles example
4. Implementation on multicore architectures
5. RGC
6. Conclusion

FairThreads

Model of threads with shared memory

- Threads linked to a scheduler are run cooperatively and share the same instants. Synchronisation and communication through broadcast signals
- Several schedulers run asynchronously. Thread migration
- Unlinked threads run in a preemptive way



FairThreads - 2

- Implementations: Java (restriction to a unique scheduler, 2002), Scheme (with specialised service threads, 2004), library of FairThreads in C (2005), LOFT (2006)
- Graphical simulations (cellular automata)

Positive points

- Absence of data-races between threads linked to the same scheduler (data-race = interference = lack of atomicity, ex: !x+!x $\neq$ 2*!x)
- Semantics of cooperation is simpler and clearer than semantics of preemption: the more atomicity is preserved, the less interleavings of instructions have to be considered
- Schedulers and unlinked threads can be run in parallel, on a multicore machine

Many Problems...

- Data-races still possible between:
  - linked and unlinked threads
  - threads linked to different schedulers
  - unlinked threads
- Lack of reactivity: non-cooperative thread linked to a scheduler
- Possibility of memory leaks:
  - data with uncontrolled growing size
  - uncontrolled creation of new threads

Actually, all are standard problems in concurrency and resource control

Plan

1. Mixing preemption & cooperation: the FairThreads model
2. **Safe reactive programming: the FunLoft language**
3. Colliding particles example
4. Implementation on multicore architectures
5. RGC
6. Conclusion

FunLoft

- Inductive data types - First order functions
  - detection of non-terminating (recursive) functions and of instantaneous loops. Consequence: termination of instants ("reactivity")
- Restriction on the flow of data carried by references and events (*stratification*). 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 Basic Syntax

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

- Distinction function/module
  - functions always terminate instantly; not mandatory for modules
  - functions can be recursively defined, modules cannot
- Schedulers, functions, and modules defined at top-level only

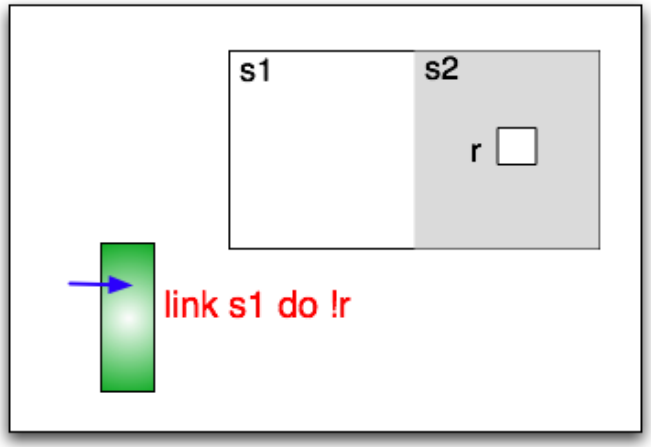
Static Analyses: Separation of the Memory

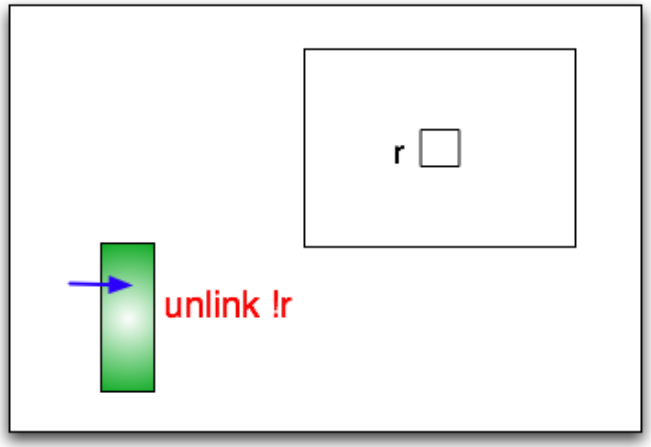
- Status public/private associated to references
  - τ `ref`$_s$: type of a public reference created in scheduler s
  - τ `ref_` : type of a private reference
- Memory separation property:
  - A public reference created in the scheduler s can only be accessed by the threads linked to s
  - A private reference can only be accessed by one unique thread
- Access effect = set of scheduler names

$$\frac{\Gamma \vdash e{:}\tau \ \texttt{ref}_s, F}{\Gamma \vdash !e{:}\tau, F \cup \{s\}} \qquad \frac{\Gamma \vdash e{:}\tau \ \texttt{ref\_}, F}{\Gamma \vdash !e{:}\tau, F}$$

Separation of the Memory - 2

- Checks:
  1. When linked to a scheduler, a thread should not access a public reference of an other scheduler
  2. When unlinked a thread should not access a public reference
- Forbidden situations:





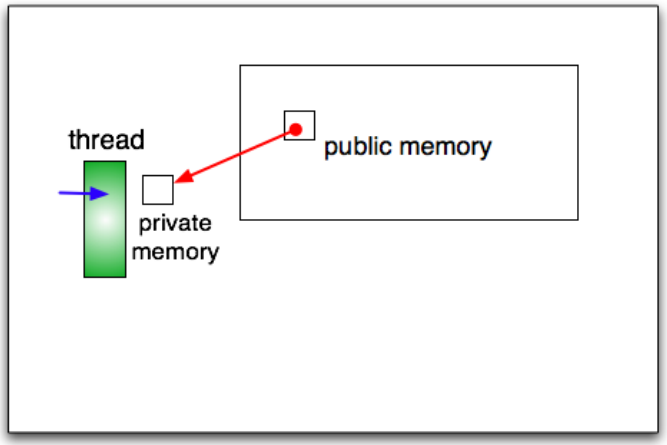
$$\frac{\Gamma \vdash e{:}F \quad F \subseteq \{s\}}{\Gamma \vdash \texttt{link}\ s\ \texttt{do}\ e{:}\emptyset} \qquad \frac{\Gamma \vdash e{:}\emptyset}{\Gamma \vdash \texttt{unlink}\ e{:}\emptyset}$$

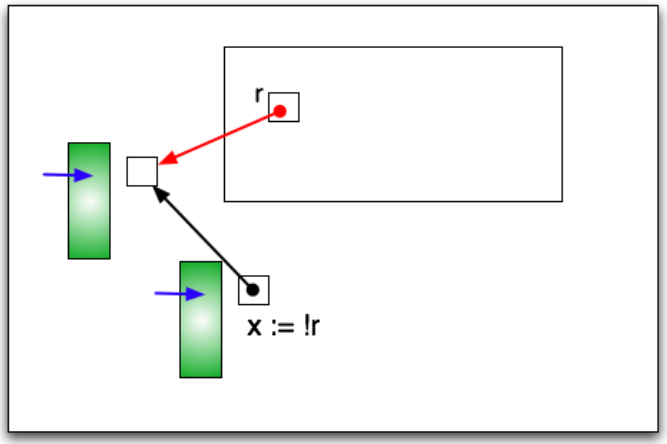
Separation of the Memory - 3

- One must also prevent a thread to access a private reference of another thread
- Check 3: parameters of a new thread should not be private

$$\frac{f{:}\overline{\tau}\rightarrow()/F \qquad \Gamma\vdash e_i{:}\tau_i,F_i \qquad \tau_i{=}\tau_i'\texttt{ref}_{\alpha_i}\Rightarrow\alpha_i\neq\_}{\Gamma\vdash\texttt{thread}\ f(\overline{e}){:}\cup F_i}$$

- Forbidden: private reference pointed to by a public reference





Separation of the Memory - 4

- Check 4: a reference and its initializing value should have same status

$$\frac{\Gamma\vdash e{:}\tau,F \;\; \tau{=}\tau'\texttt{ref}_\alpha \Rightarrow \alpha \neq \_}{\Gamma\vdash \texttt{ref}_s e{:}\tau \;\; \texttt{ref}_s,F} \qquad \frac{\Gamma\vdash e{:}\tau,F \;\; \tau{=}\tau'\texttt{ref}_\alpha \Rightarrow \alpha = \_}{\Gamma\vdash \texttt{ref}\_ e{:}\tau \;\; \texttt{ref}\_,F}$$

- Proof: Memory separation is preserved by rewriting in the formal operational semantics (extended with explicit ownership of private references)

Static Analyses: Memory Leaks

References should not be used as "accumulators"

```
let r = ref Nil_list
let f () = !r
let module m () =
   loop begin r := Cons_list (0,f()); cooperate end
```

- Stratification of references : region associated to each reference creation $r : `a\ list\ ref_k$
- Types with read/write effect:
  $f : unit \rightarrow `a\ list\ [read : k, write :]$
- $e_1 := e_2$ adds the arrow $k_1 \leftarrow k_2$ in the information flow graph, for all k_1 written by e_1 and all k_2 read by e_2.
- Absence of cycles in the graph is checked; in m, $k \leftarrow k$

Inference with Constraints

Types with effects and constraints

```
let f (r1,r2) = r1:=!r2
```

- $f : \text{`}a\ ref_k\ *\ \text{`}b\ ref_l\ \rightarrow unit\ [read : \text{`}b\ ref_l, write : \text{`}a\ ref_k]$
  $(\text{`}a\ ref_k \leftarrow \text{`}b\ ref_l)$

```
let nok () = let r = ref Nil_list in f (r,r)
```

- $\text{`}a\ list\ ref_k \leftarrow \text{`}a\ list\ ref_k\ \Rightarrow\ k \leftarrow k\ \Rightarrow\ error$

```
let ok () = let r = ref 0 in f (r,r)
```

- $int\ ref_k \leftarrow int\ ref_k\ \Rightarrow\ ok$

Constraints are collected during the construction of the most general unifier, and checked when complete

Termination of Recursive Functions

$type\ 'a\ list\ =\ Nil\_list \mid Cons\_list\ of\ 'a\ *\ 'a\ list$

- Strict sub-term order: $Cons\_list\ (head, tail) \succ tail$
- Lexicographic extension:
  $f\ (a, Cons\_list\ (h, tail), t) \succ f\ (a, tail, Cons\_list\ (h, t))$
- Analyses of chains of calls for arguments of inductive types

$$
\begin{array}{l}
let\ process\_all\_collisions\ (me, list)\ = \\
\quad match\ list\ with \\
\quad\quad Nil\_list\ ->\ () \\
\quad \mid Cons\_list\ (other, tail)\ ->\ \\
\quad\quad\ begin\ collision\ (me, other);\ process\_all\_collisions\ (me, tail)\ end \\
\quad end
\end{array}
$$

$list = Cons\_list(other, tail) \Rightarrow list \succ tail \Rightarrow (me, list) \succ (me, tail)$

Several other Static Analyses

- No instantaneous loops
- No uncontrolled thread creation in loops
  `loop begin thread m (); cooperate end`
- No thread creation while unlinked (`unlink thread m ()`)
- Events used in correct context
  - Generated values should also be stratified
  - No reference embedded in generated value
  - No event shared by distinct schedulers
  - No use of events while unlinked

Result: a well-typed program runs in bounded memory, without data-races, and instants always terminate

References

Basic reactive model:

- *A Synchronous pi-Calculus*, R. Amadio, Journal of Information and Computation 205, 9 (2007) 1470-1490.

Memory separation only, 1 scheduler, no events:

- *Cooperative Threads and Preemptive Computations*, Dabrowski, F. and Boussinot, F., Proceedings of TV'06, Seattle, 2006.

Model without distinction module/function nor join (memory separation proved) + polynomial resource control:

- *Programmation Réactive Synchrone, Langage et Contrôle des Ressources*, F. Dabrowski's Thesis, Paris 7, June 2007.

Ongoing work:

- *Formalisation of FunLoft*, F. Boussinot, I. Castellani, F. Dabrowski.

Plan

1. Mixing preemption & cooperation: the FairThreads model
2. Safe reactive programming: the FunLoft language
3. **Colliding particles example**
4. Implementation on multicore architectures
5. RGC
6. Conclusion

Example of Code: Colliding Particles

Type of particles:

```
type particle_t = Particle of
      float ref      * // x coord
      float ref      * // y coord
      float ref      * // x speed
      float ref      * // y speed
      color_t          // color
```

Module defining the particle behaviour:

```
let module particle_behavior (collide_event,color) =
  let s = new_particle (color) in
    begin
      thread bounce_behavior (s);
      thread collide_behavior (s,collide_event);
      thread draw_behavior (s);
    end
```

Particle s is shared by the three threads

Collision Behaviour

```
type 'a list = Nil_list | Cons_list of 'a * 'a list

let process_all_collisions (me,list) =
  match list with
      Nil_list -> ()
    | Cons_list (other,tail) ->
        begin collision (me,other); process_all_collisions (me,tail) end
  end

let module collide_behavior (me,collide_event) =
  let r = ref Nil_list in
    loop begin
      generate collide_event with particle2coord (me);
      get_all_values collide_event in r;
      process_all_collisions (me,!r);
      inertia (me);
    end
```

Function process_all_collisions proved to terminate. The loop in collide_behavior proved to be not instantaneous

Global System

```
let module main () =
   let draw_event = event in
   let collide_event = event in
    begin
      thread graphics (maxx,maxy,BLACK);
      thread draw_processor (draw_event,size);
      repeat particle_number do
          thread particle_behavior (collide_event,draw_event,GREEN);
    end
```

The program is ok: no possibility of data-races because shared particle data structures are only accessed by threads linked to the same scheduler

Plan

1. Mixing preemption & cooperation: the FairThreads model
2. Safe reactive programming: the FunLoft language
3. **Implementation on multicore architectures**
4. RGC
5. Conclusion

Multicore Programming

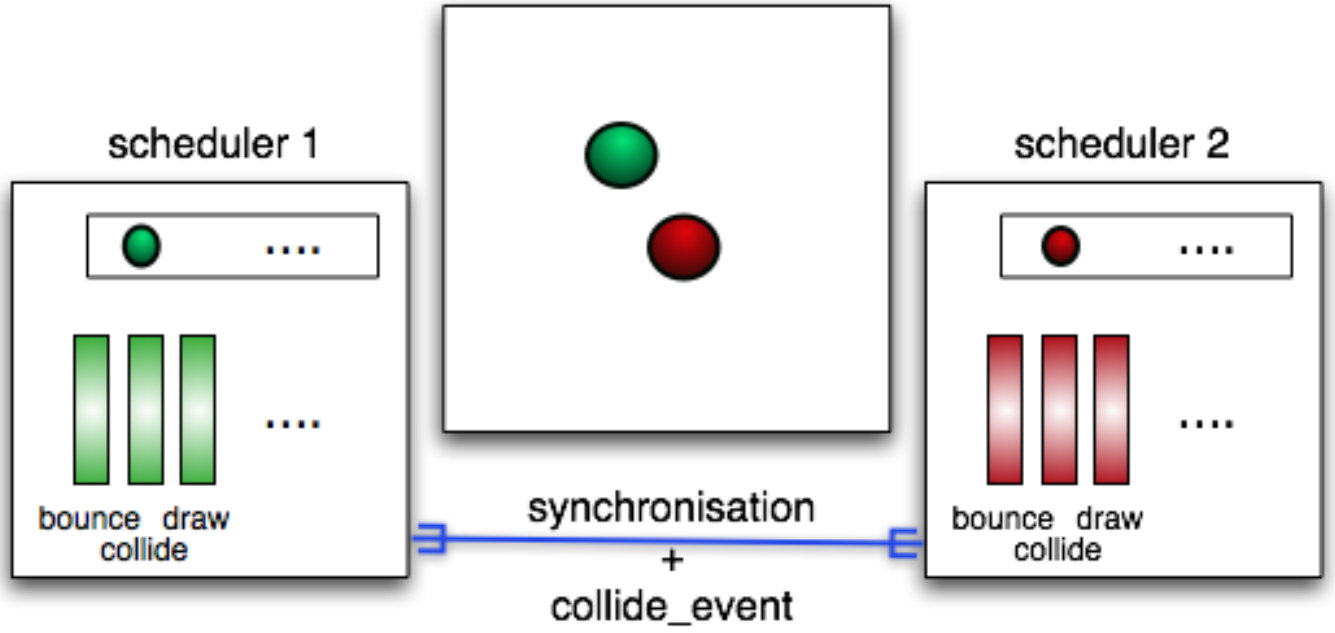
- How can a single application benefit from a multicore architecture? Answer: multithreading!
- General problem: how to get maximum of concurrency + absence of data-races + maximum of parallelism
- Specific problem: **how to adapt the colliding particles simulation to multicore machines?**

Idea: 2 schedulers, each one simulating half of the particles

- Problem 1: strong synchronisation between schedulers needed (to animate particles uniformly).
- Problem 2: collide event shared between the 2 schedulers (forbidden because the schedulers are asynchronous).

Proposal: Synchronised Schedulers

- Strong synchronisation between schedulers (common ends of instants), but parallelism during instants
- No sharing of memory (to avoid data races)
- Events shared among synchronised schedulers

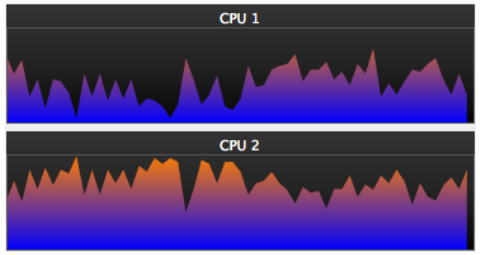


Multithreaded Colliding Particles

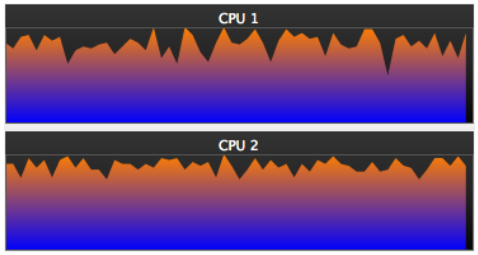
```
let s1 = scheduler and s2 = scheduler

let module main () =
  let draw_event = event in
  let collide_event = event in
    begin
      link s1 do begin
         thread graphics (maxx,maxy,BLACK);
         thread draw_processor (draw_event,size);
         repeat particle_number/2 do
            thread particle_behavior (collide_event,draw_event,GREEN);
      end;
      link s2 do
         repeat particle_number/2 do
            thread particle_behavior (collide_event,draw_event,RED);
    end
```

Demo

- CPU usage (left: 1 scheduler, right: 2 schedulers)



100% CPU



150% CPU

- Time to simulate 500 particles during 100 instants

	1 sched	2 scheds
real	**0m21.832s**	**0m14.189s**
user	0m21.102s	0m21.369s
sys	0m0.220s	0m0.379s

Plan

1. Mixing preemption & cooperation: the FairThreads model
2. Safe reactive programming: the FunLoft language
3. Implementation on multicore architectures
4. **RGC**
5. Conclusion

Reactive GC (basic ideas)

- Main objective: responsiveness
- Based on the notion of instant
- Concurrent, parallel, on-the-fly, generational
- No "stop-the-world" phase (as with tracing-based GCs)
- Unable to handle general cyclic data structures

RGC Algorithm

- References have a status protected/unprotected (initially unprotected)
- Turns protected when needs to survive at the next instant (ex: created by a thread)
- Unprotected objects can be safely collected at the end of the current instant (ex: created by function) by the scheduler to which they belong
- Protection level associated to protected objects. Collection when the protection level falls to 0 (as with reference counting based GCs)

RGC Algorithm

- Generational aspect: short-living = unprotected = one instant, long-living = protected = several instants
- Concurrent: collection by one scheduler, allocation by another
- Parallel: collection in real parallelism by 2 schedulers run on 2 different cores (protection needs locks, in the general case)
- On-the-fly: does not require full halt of the system ("stop-the-scheduler" instead of "stop-the-world")

Conclusion

FunLoft is experimental!

- Lack of realistic bounds (polynomial?)
- Over-restricted detection of termination of functions
- No distribution, no objects, etc...

FunLoft provides:

- Concurrent programming with clear semantics
- Static analyses to prevent data-races and memory leaks, and to ensure reactivity
- Efficient implementation: large number of components
- Support for multithreaded applications on multicore machines

Compiler available at www.inria.fr/mimosa/rp/FunLoft