Safe Reactive Programming: the FunLoft Proposal

Frédéric Boussinot
MIMOSA Project, Inria Sophia-Antipolis
(Joint work with Frédéric Dabrowski)

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

With support from ALIDECS

SYNCHRON 2007

Introduction

Why not a General Purpose Synchronous Language?

- Modularity: how to reuse code?
- Dynamicity: how to deal with non-static aspects? (ex: memory allocation)
- Asynchrony: how to deal with asynchronous aspects? (ex: blocking IOs)
- Safe programming? (ex: how to prove the termination of instants)
- Efficiency? (ex: how to benefit from multiprocessors)

Plan

1. Modularity & dynamicity: the causality issue
2. Mixing synchrony & asynchrony: the FairThreads model
3. Safe reactive programming: the FunLoft language
4. Efficiency: implementation on multicore architectures
5. Conclusion

Modularity (Compositionality)

- Question: how to define the specification associated with a given code, allowing this code to be used in various contexts?
- Problem with causality: specifications are over-complex
- Example: parallel combination of `P` and `Q`
  - `P = present s1 else emit s2 end`, `Q = present s2 then emit s1 end`. `P || Q` has no solution (causality error)
  - `(pause;P) || Q` is correct (constructive causality), but `(pause;P) || (pause;Q)` is not.
  - `(pause;pause;P) || (pause;Q)` is correct, but ...
- Information needed for putting `P` in parallel with `Q` without causality errors is as complex as the semantics (automaton) of `P`. No hope! *"The map is as large as the Empire..."*

Dynamicity

- Dynamic creation of new parallel components arise in many contexts:
  - Interpretor: interpretation of a new entry
  - Embedded system: new versions, adding of new functionalities
  - Agent system: migrating agent reaching a new site
  - Simulation: creation of new elements to simulate
- In all these contexts, it is difficultly acceptable that the creation of a new component could raise causality issues

For both modularity and dynamicity concerns, causality issues are a major drawback

An Alternative to Causality Issues

Delay to the next instant reaction to signal absence

1. `present s then P else Q end`: if `s` is present, `P` is immediately executed; if `s` is absent, `Q` is executed at the next instant.

2. If solutions with `s` present and `s` absent both exists, choose the one with absence.

Example: in `present s else emit s end`, `s` is emitted at the next instant if it is absent

Intuition: to be sure that a signal is absent you have to wait until the end of instant. Implementation: when waiting for `s`, execution suspends until `s` is emitted, or the instant terminates

Delayed Reaction to Absence

Causality errors are ruled out

Compositionality becomes achievable

New parallel components can be added at run time

- SL, SugarCubes, ReactiveML, FairThreads, ... are based on the delayed reaction to absence
- Limitations of expressivity:
  1. No strong preemption (strong abort), only weak one
  2. Values of signals not immediately available
- Pragmatics: not really severe restrictions... (anyway, to be compared to the introduction of `pause` statements to solve causality problems)

Comparison with the standard approach (Esterel) still to be done for real-life programs

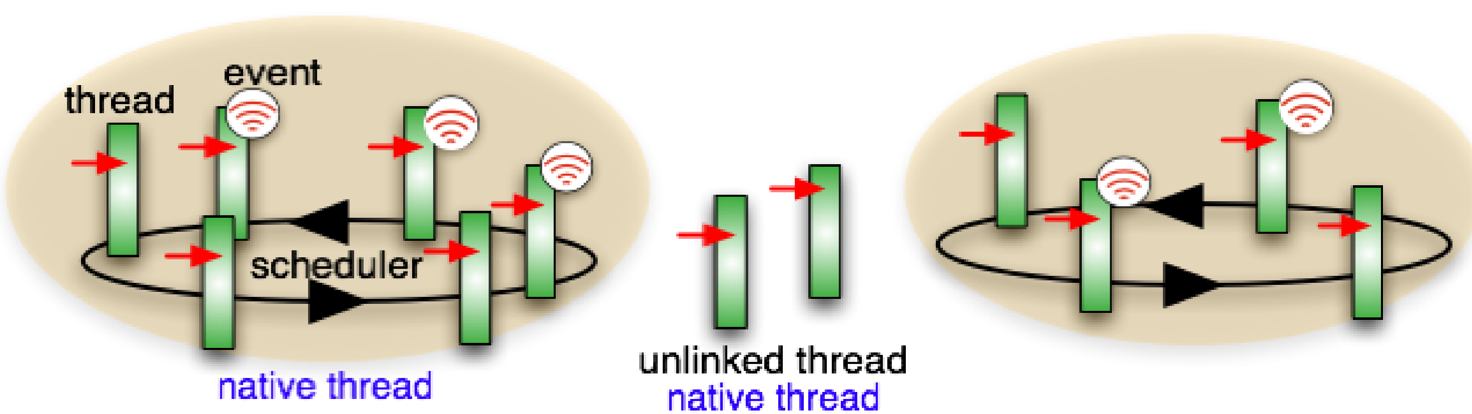
Plan

1. Modularity & dynamicity: the causality issue
2. **Mixing synchrony & asynchrony: the FairThreads model**
3. Safe reactive programming: the FunLoft language
4. Efficiency: implementation on multicore architectures
5. 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

- GALS aspect of FairThreads: schedulers corresponds to locally synchronous areas; systems made of several schedulers are globally asynchronous
- 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)

Many Problems

- Data-races (= interference = lack of atomicity, ex: !x+!x $\neq$ 2*!x) between linked and unlinked threads
- Data-races between threads linked to different schedulers
- Data-races between unlinked threads
- Non-cooperative thread linked to a scheduler (lack of reactivity)
- Uncontrolled creation of new threads
- Data with uncontrolled growing size (memory leaks)
- Buffering of communication between schedulers

Actually, all are standard problems in concurrency and resource control!

Also Problems in Synchronous Languages

These problems also exist for Synchronous Languages, at host language level

```
module m :
   var x := Nil : list in
      loop
         x := f(x);
         pause
      end
   end
end module
```

- Memory leaks: `list f(list x) {return Cons(0,x);}`
- Lack of reactivity: `list f(list x) {return f(x);}`
- Data-races in the context of GALS:

  `list f(list x) {return Cons(global,Cons(global,x));}`

Plan

1. Modularity & dynamicity: the causality issue
2. Mixing synchrony & asynchrony: the FairThreads model
3. **Safe reactive programming: the FunLoft language**
4. Efficiency: implementation on multicore architectures
5. Conclusion

FunLoft

- Inductive data types - First order functions
  - Termination detection of recursively defined functions.
    Consequence: termination of instants ("reactivity")
- Restriction on the flow of data carried by references and events (*stratification*)
  Consequence: bounded system size $\Rightarrow$ 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 $\Rightarrow$ absence of data-races

FunLoft Basic Syntax

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

- 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

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
```

Note: 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

The 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

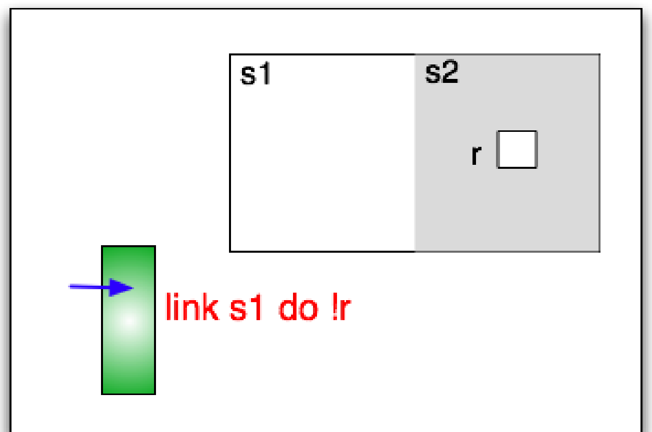
Static Analyses: Separation of the Memory

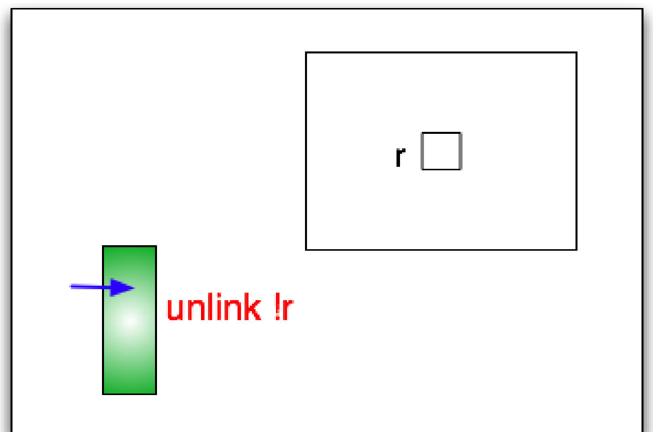
- Status public/private associated to references
  - τ `ref`$_s$: type of a public reference created in scheduler s
  - τ `ref`$_\_$: type of a private reference
- Memory separation property:
  - A public reference created in the scheduler s can only be accessed by the threads linked to s
  - A private 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:





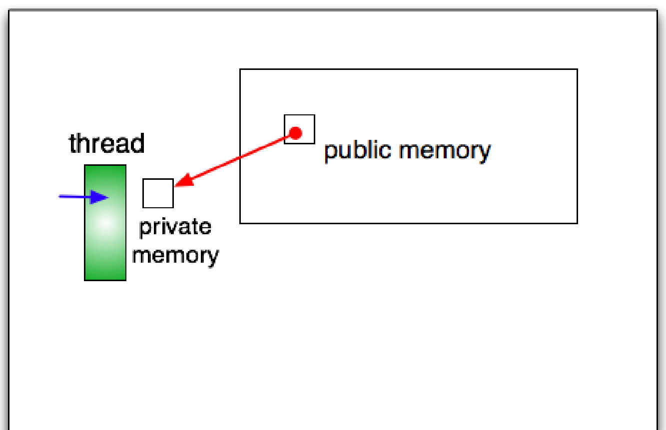
$$\frac{\Gamma \vdash e : F \qquad F \subseteq \{s\}}{\Gamma \vdash \texttt{link}\ s\ \texttt{do}\ e : \emptyset} \qquad \frac{\Gamma \vdash e : \emptyset}{\Gamma \vdash \texttt{unlink}\ e : \emptyset}$$

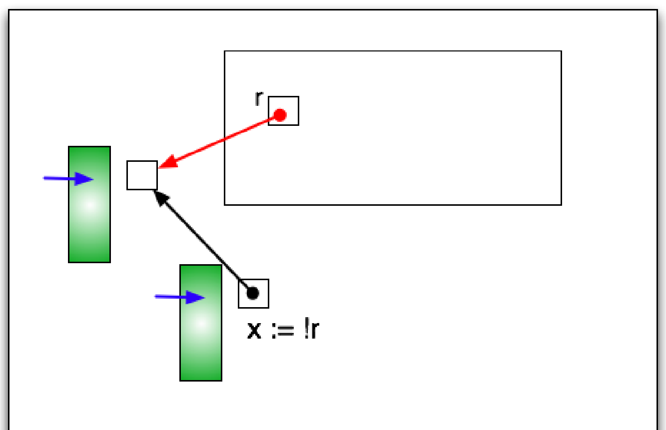
Separation of the Memory - 3

- 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

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

- 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

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

- 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 $r : \text{`}a\ list\ ref_k$
- Types with read/write effect: $f : unit \rightarrow \text{`}a\ list\ [read : k, write :]$
- $e_1 := e_2$ 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

$let\ process\_all\_collisions\ (me, list)\ =$

$\quad match\ list\ with$

$\quad\quad Nil\_list\ -> ()$

$\quad \mid Cons\_list\ (other, tail)\ ->$

$\quad\quad begin\ collision\ (me, other);\ process\_all\_collisions\ (me, tail)\ end$

$\quad end$

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

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, F. Dabrowski.

Plan

1. Modularity & dynamicity: the causality issue
2. Mixing synchrony & asynchrony: the FairThreads model
3. Safe reactive programming: the FunLoft language
4. **Efficiency: implementation on multicore architectures**
5. Conclusion

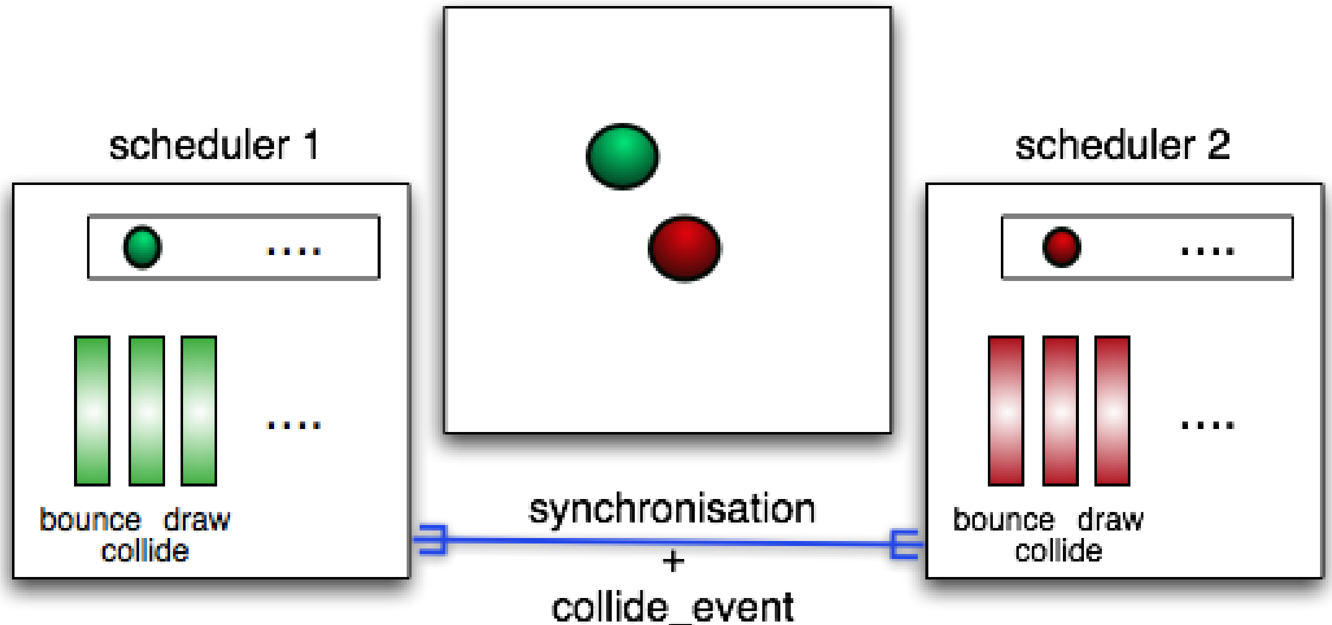
Multicore Programming

- 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 as 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

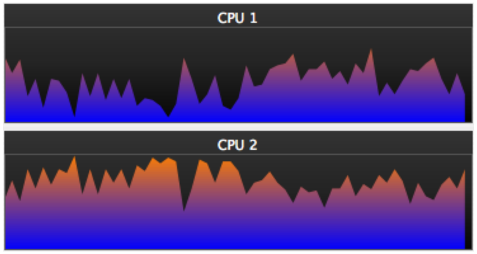


Multithreaded Colliding Particles

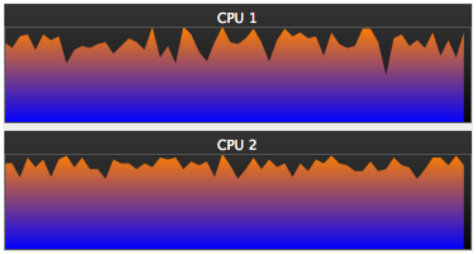
```
let s1 = scheduler and s2 = scheduler

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

Gain: $21/14 = 1.5$

- Gain (1000 instants)

particles	100	200	300	400	500	600	1000
gain	1.2	1.3	1.4	1.51	1.52	1.56	1.57

Conclusion

FunLoft is experimental and far from being a GPSL!

- 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
- Syntax for multithreaded applications on multicore architectures

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