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 \( Q \)
  - \( P = \text{present } s_1 \text{ else emit } s_2 \text{ end}, Q = \text{present } s_2 \text{ then emit } s_1 \text{ end}. P \parallel Q \) has no solution (causality error)
  - \((\text{pause};P) \parallel Q\) is correct (constructive causality), but \((\text{pause};P) \parallel (\text{pause};Q)\) is not.
  - \((\text{pause};\text{pause};P) \parallel (\text{pause};Q)\) 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:
  - Interpreter: 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 correspond 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 ⇒ 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

\[ p ::= x \mid C(p, \ldots, p) \]
\[ e ::= x \mid C(e, \ldots, e) \mid \text{match} \ x \ \text{with} \ p -> e \mid \ldots \mid p -> e \]
\[ \mid f(e, \ldots, e) \mid \text{let} \ x = e \ \text{in} \ e \mid \text{ref} \ e \mid !e \mid e := e \]
\[ \mid \text{cooperate} \mid \text{thread} \ f(e, \ldots, e) \mid \text{join} \ e \mid \text{unlink} \ e \mid \text{link} \ s \ \text{do} \ e \]
\[ \mid \text{event} \mid \text{generate} \ e \ \text{with} \ e \mid \text{await} \ e \mid \text{get\_all\_values} \ e \ \text{in} \ e \]
\[ \mid \text{loop} \ e \mid \text{while} \ e \ \text{do} \ e \]

- 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

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 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
  – $\tau \text{ref}_s$ : type of a public reference created in scheduler $s$
  – $\tau \text{ref}_-$ : 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
  \[
  \frac{\Gamma \vdash e : \tau \text{ref}_s, F}{\Gamma \vdash !e : \tau, F \cup \{s\}} \quad \frac{\Gamma \vdash e : \tau \text{ref}_-, F}{\Gamma \vdash !e : \tau, F}
  \]
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:

\[
\begin{align*}
\Gamma \vdash e : F & \quad F \subseteq \{s\} \\
\Gamma \vdash \text{link } s \text{ do } e : \emptyset \\
\Gamma \vdash \text{unlink } e : \emptyset
\end{align*}
\]
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

\[
\frac{f : \tau \rightarrow ()/F \quad \Gamma \vdash e_i : \tau_i, F_i \quad \tau_i = \tau'_i \text{ref}_{\alpha_i} \Rightarrow \alpha_i \neq -}{\Gamma \vdash \text{thread } f(e) : \bigcup F_i}
\]

- 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

\[
\frac{\Gamma \vdash e : \tau, F \quad \tau = \tau' \ref_\alpha \Rightarrow \alpha \neq -}{\Gamma \vdash \ref_s e : \tau \quad \ref_s, F} \quad \frac{\Gamma \vdash e : \tau, F \quad \tau = \tau' \ref_\alpha \Rightarrow \alpha = -}{\Gamma \vdash \ref_e e : \tau \quad \ref_e, F}
\]

• 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 \( m () = \)

\[
\text{loop begin } r := \text{Cons_list} (0,f()); \text{ cooperate end}
\]

- Stratification of references: region associated to each reference creation \( r : \text{‘a list ref} \)

- Types with read/write effect:
  \( f : \text{unit} \to \text{‘a list [read :} k, \text{write :]} \)

- \( e_1 := e_2 \) 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

- \( f : \text{'a ref}_k \times \text{'b ref}_l \rightarrow \text{unit} \) [\text{read} : \text{'b ref}_l, \text{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)

- \( \text{'a list ref}_k \leftarrow \text{'a list ref}_k \Rightarrow k \leftarrow k \Rightarrow \text{error} \)

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

\[ \text{type } \texttt{\`a list} = \texttt{Nil_list} \mid \texttt{Cons_list \ of \ `a \ast `a list} \]

- **Strict sub-term order**: \( \texttt{Cons_list (head, tail)} \succ \texttt{tail} \)

- **Lexicographic extension**: 
  \[ f (a, \texttt{Cons_list (h, tail)}, t) \succ f (a, \texttt{tail}, \texttt{Cons_list (h, t)}) \]

- Analyses of chains of calls for arguments of inductive types

```
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 
```

\[ \texttt{list} = \texttt{Cons_list(other, tail)} \Rightarrow \texttt{list} \succ \texttt{tail} \Rightarrow (\texttt{me, list}) \succ (\texttt{me, tail}) \]
Several other Static Analyses

- No instantaneous loops
- No uncontrolled thread creation in loops
  \[\text{loop begin thread } m(); \text{ cooperate end}\]
- No thread creation while unlinked (\texttt{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

\textbf{Result:} a well-typed program runs in bounded memory, without data-races, and instants always terminate
References

Basic reactive model:


Memory separation only, 1 scheduler, no events:


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


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;
        end;
        link s2 do
            repeat particle_number / 2 do
                thread particle_behavior (collide_event, draw_event, RED);
            end;
    end
Demo

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

100% CPU 150% CPU

• Time to simulate 500 particles during 100 instants

<table>
<thead>
<tr>
<th></th>
<th>1 sched</th>
<th>2 scheds</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td><strong>0m21.832s</strong></td>
<td><strong>0m14.189s</strong></td>
</tr>
<tr>
<td>user</td>
<td>0m21.102s</td>
<td>0m21.369s</td>
</tr>
<tr>
<td>sys</td>
<td>0m0.220s</td>
<td>0m0.379s</td>
</tr>
</tbody>
</table>

• Gain (1000 instants)

<table>
<thead>
<tr>
<th>particles</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>gain</td>
<td>1.2</td>
<td>1.3</td>
<td>1.4</td>
<td>1.51</td>
<td>1.52</td>
<td>1.56</td>
<td>1.57</td>
</tr>
</tbody>
</table>
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