# Dynamic Synchronous Language

Frédéric Boussinot
INRIA Méditerranée
http://www-sop.inria.fr/members/Frederic.Boussinot

Jean-Ferdy Susini
CNAM - Cédric

January 2010

ANR-08-EMER-010

# Introduction

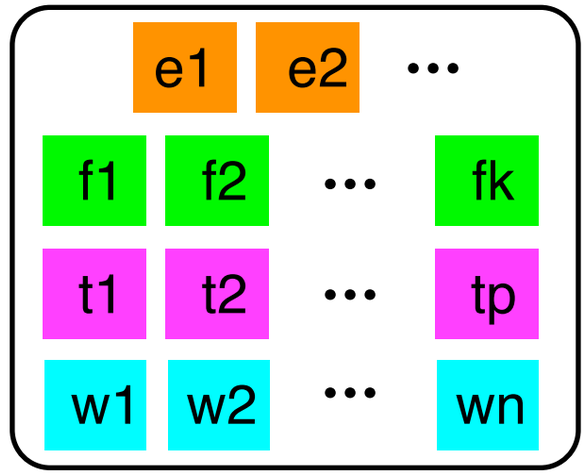
- How to preserve compile-time checked safety while allowing dynamic features, such that program interpretation (scripting) ?
- How dynamic features can be introduced in FunLoft ?

## Plan

- Basic model considered
- Description of the instructions
- Issue of memory isolation
- Implementation
- Related work

# Basic Model

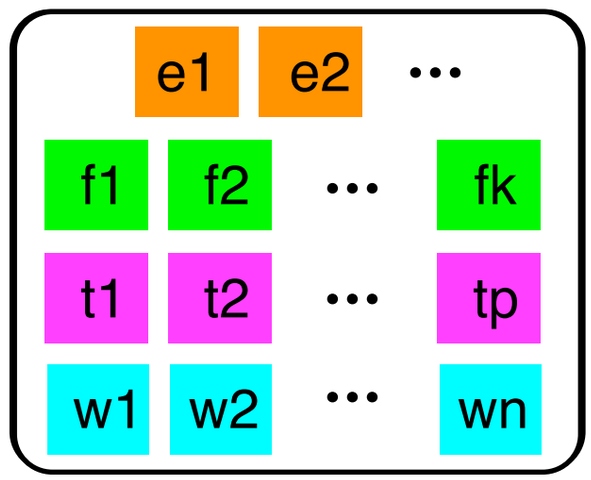
- Systems are composed of several sites. Sites define:
  - events ei
  - functions fi and wrappers wi
  - tasks ti




- Events are used to trigger execution (basically, they are generated and awaited)
- Events are dynamically created and different on each site

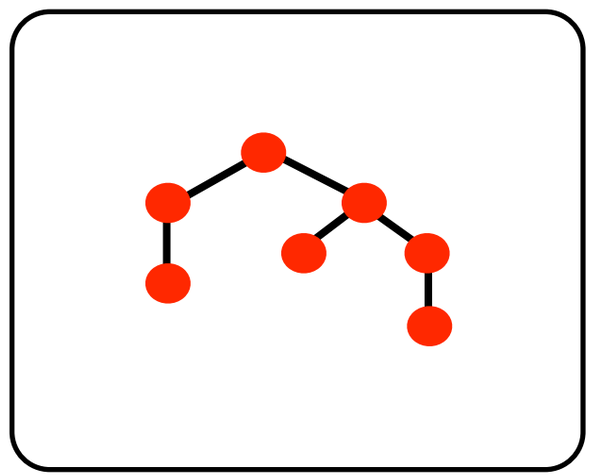
# Basic Model - 2

- Functions and wrappers are atomic and terminate immediately, tasks are not
- Wrappers are special functions returning a value (int or bool)
- All sites define the same functions, tasks, and wrappers which are implemented by the programmer
- Each site executes an instruction (a tree)
- Instructions can be dynamically added to sites



# Basic Model - 2

- Functions and wrappers are atomic and terminate immediately, tasks are not
- Wrappers are special functions returning a value (int or bool)
- All sites define the same functions, tasks, and wrappers which are implemented by the programmer
- Each site executes an instruction (a tree)
- Instructions can be dynamically added to sites



# Scripts

- nothing
- *fun* (*params*)
- launch *task* (*params*)
- s1; s2
- s1 || s2
- if *bw* (*params*) then s1 else s2 end
- repeat *iw* (*params*) do s end
- while *bw* (*params*) do s end
- cooperate
- generate e
- await e
- do s watching e
- drop s in *site*

*Cooperative parallelism + round-robin*

*Synchronous parallelism (instants)*

*Asynchronous parallelism*

# Infinite Loops

- while const_true () do i end executes i forever

```
while const_true () do
   await go; print ("ok");
   cooperate
end
```

Prints a message each time “go” is generated

- The body of a loop should not terminate instantly; if it is the case, a cooperate is automatically introduced by the system (at run-time)

# Preemption

- do i watching e forces the termination of the execution of i when e is generated

- Preemption is "weak"

```
generate e;
do print ("ok") watching e
```

ok is printed

- Preemption is never immediate

```
generate e;
do nothing watching e;
print ("ok")
```

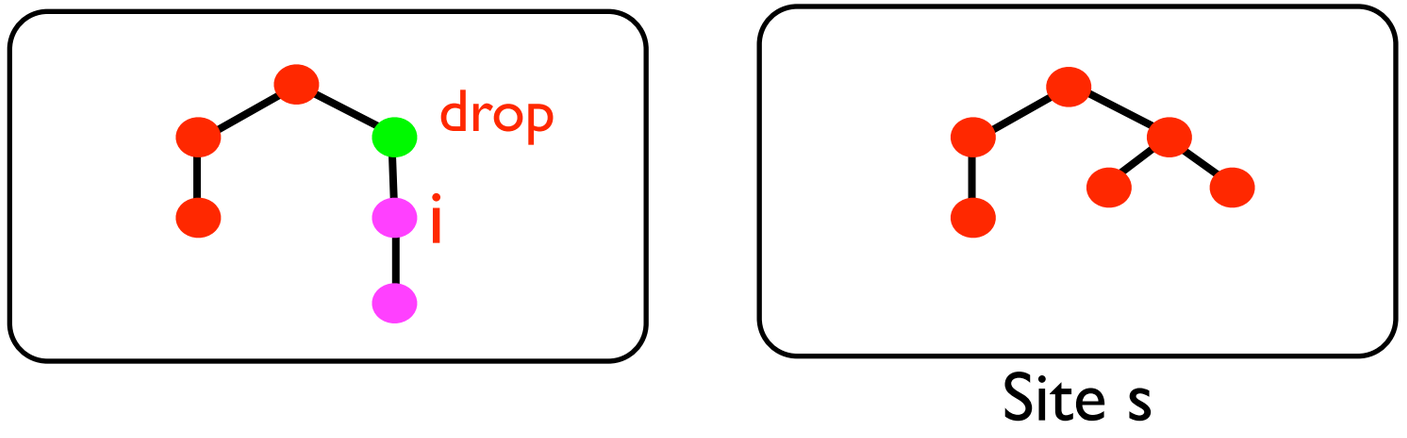
ok is printed
at next instant

- No "causality error"

```
do generate e watching e
```

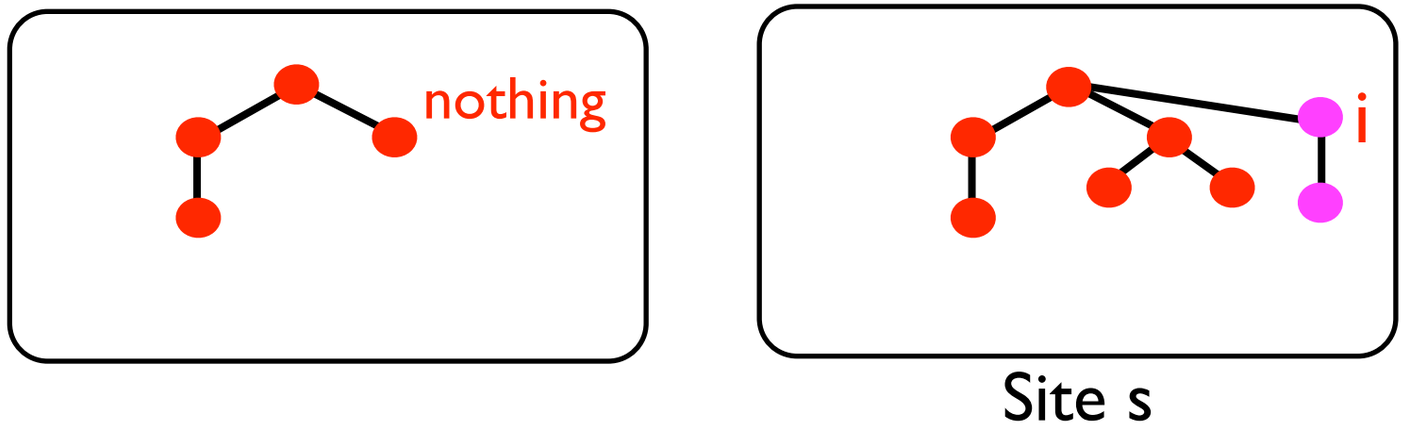
# Migration

- drop i in s drops i in the site s, and terminates



# Migration

- drop i in s drops i in the site s, and terminates




- Synchronous execution of i on the remote site s:
  drop i; generate done in s || await done
- To generate an event in the remote site s:
  drop generate e in s
- To call a function of the remote site s:
  drop f (...) in s

# Tasks

- Task are used when atomicity and instantaneity cannot be assumed:
  - use of blocking functions
  - need of migration

- Example: getting a character from the keyboard

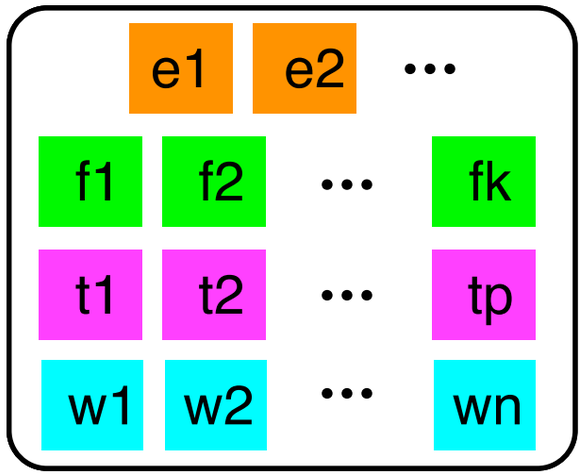
```
let module getchar () =
  let loc = local ref ' ' in
    begin
      unlink
        loc := fl_getchar ();
        link main_scheduler do
            getchar_result := !loc;
    end
```

```
let getchar_result = ref ' '


    let module task (fun,params) =
        if fun = "getchar" then
              run getchar ()
        ...
```

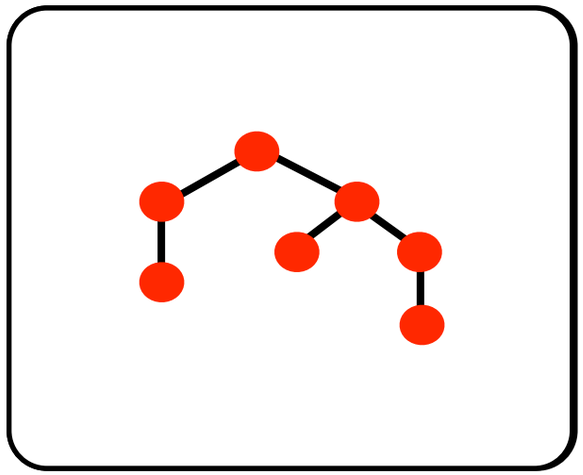
# Memory Isolation

- Memory is accessed by calls, tasks, and wrappers



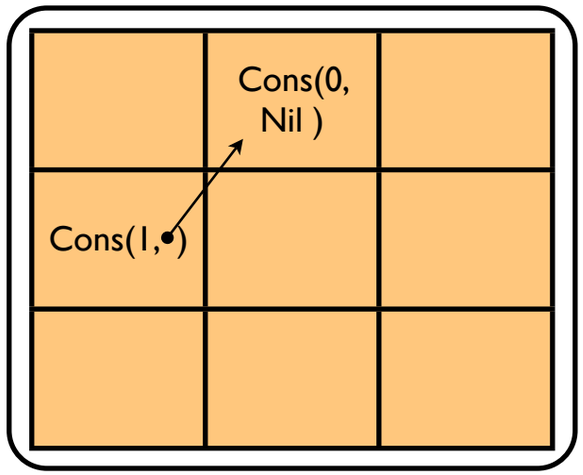
# Memory Isolation

- Memory is accessed by calls, tasks, and wrappers



# Memory Isolation

- Memory is accessed by calls, tasks, and wrappers




- The language definition demands that the memory of one site cannot be accessed concurrently by the functions, tasks, and wrappers run on an other site

# Memory Isolation - 2

- Consider a reference used in the following way:

```
let r = ref 0
let call_dispatch (fun,params) =
  if fun = "r_write" then r := 1
  ...
```

```
let module int_wrapper (fun,res) =
  if fun = "r_read" then res := !r
  ...
```

there is a type error: r could be accessed concurrently by two distinct sites

- Possible solution: link to the same site

```
let module task_dispatch (fun,params) =
   if fun = "r_write" then
     link s do r := 1
  ...
```

```
let module int_wrapper (fun,res) =
   if fun = "r_read" then
     link s do res := !r
   ...
```

# Lacks

- let event e in i end declares an event e local to the instruction i
- Valued events
- Suspend/resume instructions
- Interface with the network
- Interpretor of scripts
- ...

# Programming

1. Include the definition of instructions, evaluation, etc.
2. Define the sites
3. Define the calls, tasks, and wrappers needed
   - adapt the dispatchers for functions, tasks, and wrappers, to call them
4. Compile the whole system with FunLoft
5. Run the executable code produced

## Strong Limitations

- All the sites share the same calls, tasks, and wrappers
- Sites cannot be dynamically created

# Implementation

- Coded in FunLoft with finite memory checks switched-off (recursive modules and thread creation in loops allowed)
- Program of ~ 800 lines of code
- Micro-steps based execution, threads used to evaluate instructions, termination of instructions signaled by events
- Site = scheduler

- Code production in SugarCubes

# Related Work

- SugarCubes/Jr/ReactiveMachines
- Reactive Scripts (over Tcl/Tk)
- ReactiveML

# Conclusion

- Synchronous language with dynamic features
- A kind of "orchestration language"
- Mixing of cooperative/preemptive approaches
- Safety coming from FunLoft (memory protection)
- Mapping of sites on multicore architectures

*Draft paper*

## Future Work:

- Distribution over the network
- Interpretor

# Summary

```
type instruction_t =
    Nothing
  | Cooperate
  | Print of string
  | Call of string * string list
  | Launch of string * string list
  | Seq of instruction_t * instruction_t
        * ref bool * ref thread_t
| If of bool_wrapper_t * instruction_t * instruction_t
        * ref bool * ref thread_t
| Par of instruction_t * instruction_t
        * ref thread_t * ref thread_t
| Loop of instruction_t
        * ref thread_t
| Repeat of int_wrapper_t * instruction_t
        * ref thread_t
| Generate of string
| Await of string
| Watching of string * instruction_t
        * ref thread_t * ref thread_t
| Drop of ref site_r * instruction_t
```

```
type bool_wrapper_t =
    True
  | False
  | BoolWrapper of string * string list
```

```
type int_wrapper_t =
    IntConstWrapper of int
  | IntWrapper of string * string list
```