Event-driven Architectures meet Threads, a programming language perspective

Frédéric Boussinot, Bernard Serpette, Manuel Serrano

Inria Sophia Antipolis

`2004 route des Lucioles - BP 93`

`F-06902 Sophia Antipolis, Cedex`

`France`

1/31 – **Overview of the presentation**

- Threads:
  1. Motivations: the Bee programming environment
  2. Preemptive threads vs cooperative threads
  3. An example illustrating threads difficulties
  4. FairThreads
- API:
  - Cooperative IOs
  - Various examples
  - Semantics

– Motivations

Adding threads to the Scheme Bigloo runtime system

- The former versions did not contain threads
- High pressure from users

Threads? What is that?

- Concurrency
- Shared memory
- Preemptive scheduling

Threads? What are they good for?

- Programming servers (files, graphics, ...)
- SMP (efficiency)
- Reusability thanks to preemption

3/31 – **Threads, a hard living**

It is difficult to add threads to a non "thread safe" system

- Difficult to implement **correctly**
- Difficult to implement **efficiently**

It is difficult to program with threads

- **Debugging** is *extremely* difficult (no replay)
- Heavy-weight **synchronization**
- Weak **modularity** ("independently correct" codes may deadlock)

Threads have a loose semantics

- The semantics of the **scheduler** is generally left unspecified
- **Non deterministic** APIs (*e.g.*, "who" is notified for a `join`)
- Restricted APIs (*e.g.*, Java `suspend` (deprecated) and `destroy` (not implemented))

 Sine + Cosine = ? (1/3)

The principle

- Two threads are moving the same ball:
  - One thread follows a sine path
  - One thread follows a cosine path
- The color of the ball changes each time it is painted

Excerpt of the source code

```
class Ball {
   double x, y, x0, y0, angleX = 0, angleY = 0; ...
   public void sine() {
      angleY += ANGLE_INC;
      y = y0 + radiusY*Math.sin(angleY);
   }
   public void cosine() { ... }
}
```

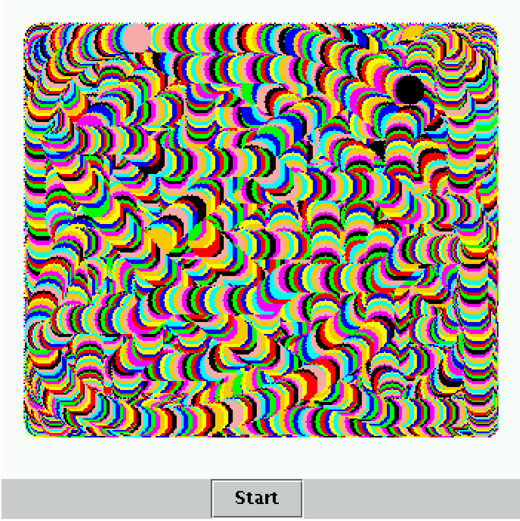
 – **Sine + Cosine = ? (2/3)**

Excerpt of the source code (end)

```
class Cos extends Thread {
   Ball ball; Panel panel;
   public Cos(Ball b,Panel a){ ball = b; panel = a; }
   public void run() {
      while(true) {
        ball.cosine();
        panel.paint(panel.getGraphics());
      }
   }
}
void main( String args[] ) {
   ...
   new Sin(ball,panel).start();
   new Cos(ball,panel).start();
}
```

6/31 – **Sine + Cosine = ? (3/3)**

What is to be expected?



Interpretation

- The two threads **do not** cooperate
- The scheduler **switches** from one thread to the other
- Switches occur at **arbitrary** moments
- Switches are visualized by lines and colums in the output

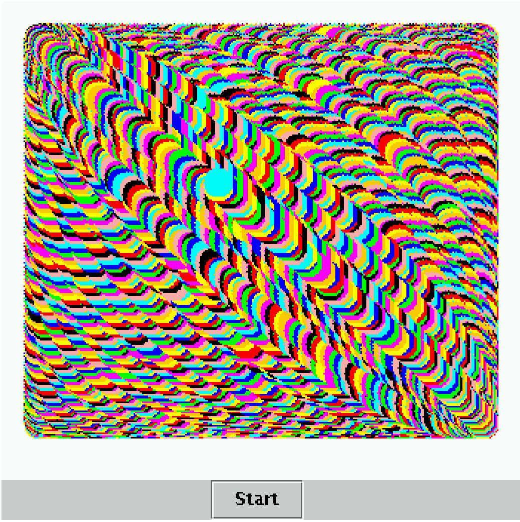
 – Sine + Cosine + Yield = ? (1/2)

Enforcement of cooperation with `Thread.yield()`

```
class Cos extends Thread {
    ...
    public void run() {
        while(true) {
            ball.cosine();
            panel.paint(panel.getGraphics());
            Thread.yield();
        }
    }
}
```

8/31 – Sine + Cosine + Yield = ? (2/2)

What is to be expected?



Interpretation

- The two threads **do** cooperate
- The scheduler has complete **freedom** to choose the next thread to run
- Switches can still occur at **arbitrary** moments

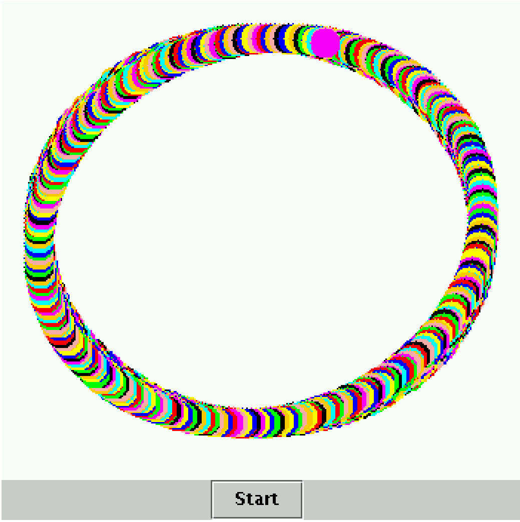
 – **Sine + Cosine + Sleep = ? (1/2)**

Enforcement of cooperation with `Thread.sleep()`

```
class Cos extends Thread {
    ...
    public void run() {
        while(true){
            ball.cosine();
            panel.paint(panel.getGraphics());
            try {
                Thread.sleep(1);
            } catch(Exception _) {}
        }
    }
}
```

 – **Sine + Cosine + Sleep = ? (2/2)**

What is to be expected?



Interpretation

- Using `Thread.sleep` forces the scheduler to choose the **other** thread as next thread to run
- The scheduler has the choice **when** the two threads are awake
- A **third** thread, behind the scene, deals with graphics

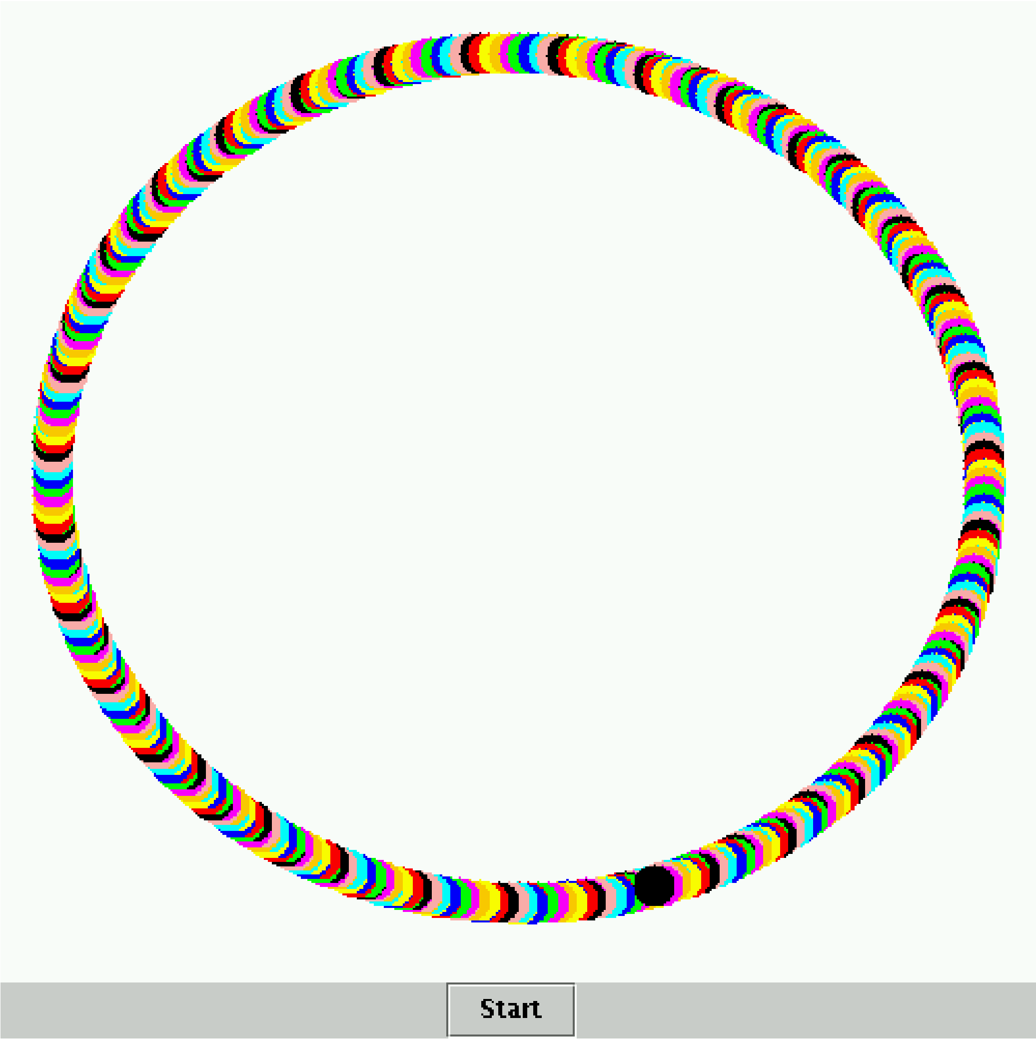
11/31 – Sine + Cosine + Wait/Notify = ? (1/2)

Synchronization with wait/notify

```
class Cos extends Thread {
    ...
     public void run() {
        while(true) {
           synchronized(ball){
              ball.cosine();
              panel.paint(panel.getGraphics());
              ball.notify();
              try {
                 ball.wait();
              } catch(Exception _) {}
           }
        }
    }
}
```

12/31 – Sine + Cosine + Wait/Notify = ? (2/2)

What is to be expected?

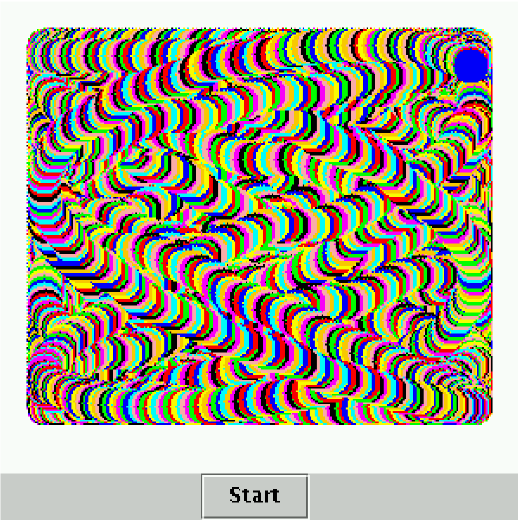


13/31 – Lissajous

What about adding a new thread?

```
void main( String args[] ) {
   ...
   new Sin(ball,panel).start();
   new Cos(ball,panel).start();
   new Cos(ball,panel).start();
}
```

Weak modularity...



FairThreads are deterministic

- A cooperative round robin scheduler
- The API is defined in term of instant

Library is nondeterministic

- Concurrent inputs/outputs
- Asynchronous processes

Principles

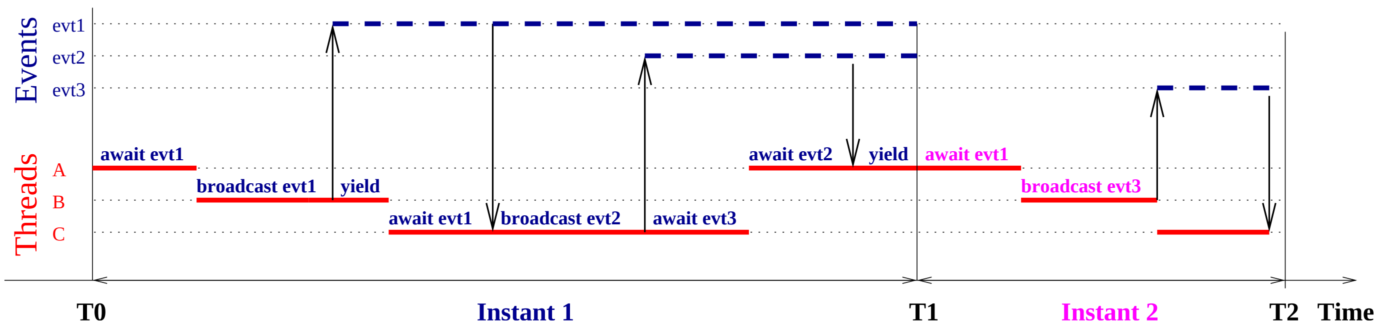
- No **preemption**: threads yield the processor
    - *Explicitly*
    - *Implicitly* (when waiting for an event)
- **Events**:
    - Threads *communicate* by the means of broadcast events
    - Values can be associated to events
- **Instants**:
    - The instant is completed when all threads:
        - have yield the processor (blocked)
        - are completed
    - Every thread *sees* all the events emitted during the *instant*
    - Events are *erased* at the end of instants

 Example

3 threads

A	B	C
1: await evt1 2: await evt2 3: yield 4: await evt1	1: broadcast evt1 2: yield 3: broadcast evt3	1: await evt1 2: broadcast evt2 3: await evt3

Scheduling



17/31 – **Overview of FairThread API**

- Threads:
  1. Motivations: the Bee programming environment
  2. Preemptive threads vs cooperative threads
  3. An example illustrating threads difficulties
  4. FairThreads
- API:
  1. Cooperative IOs
  2. Various examples
  3. Semantics

FairThreads programming pattern

A typical Bigloo multithreaded application proceeds as follows:

1. **Creates** a scheduler
2. **Creates** threads
3. **Adds** the threads to a scheduler
4. **Proceeds** schedulers reactions

Scheduler API

Scheduler creation and test:

```
(make-scheduler)
(current-scheduler)
(scheduler? obj)
```

Scheduler reaction:

```
(scheduler-react! . scheduler)
(scheduler-start! . args+scheduler)
```

Scheduler Broadcast:

```
(scheduler-broadcast! scheduler event val)
```

Thread API (1/2)

Thread creation, test and run:

```
(make-thread thunk . name)
(thread? obj)
(current-thread)
(thread-start! thread . scheduler)
```

Thread control:

```
(thread-sleep! timeout)
(thread-yield!)
(thread-join! thread . timeout)
(thread-suspend! thread)
(thread-resume! thread)
(thread-terminate! thread)
```

Thread API (2/2)

Events:

```
(broadcast! event val)
(thread-await! event . timeout)
(thread-get-values! event)
```

Concurrent IOs and Processes:

```
(make-input-signal port param)
(make-output-signal port string)
(make-connect-signal socket)
(make-process-signal process)
```

Examples of await:

```
(thread-await! 'foo) ;; wait until the event 'foo is broadcast
(thread-await! (make-input-signal p 1024)) ;; wait for 1024 chars
(thread-await! (make-input-signal p "MARK")) ;; wait for "MARK"
(thread-await! (make-connect-signal s)) ;; wait for a socket connection
```

HTTP server (1/3)

An HTTP server illustrating cooperative IOs

```
(define (start-server)
   (let* ((s (make-socket-server))
          (thd (make-thread (make-http-server s) 'http-server))
          (ts (thread-start! thd)))
      (scheduler-start!)
      (fprint (current-error-port) "Shuting down http server...")
      (socket-shutdown s)))
(define (make-http-server s::socket)
   (lambda ()
      (let loop ()
         (let ((s2 (socket-dup s)))
            (thread-await! (make-connect-signal s2))
            (thread-start! (make-thread (lambda () (http-eval s2))))
            (thread-yield!)
            (loop))))))
```

Http server (2/3)

Executing a request

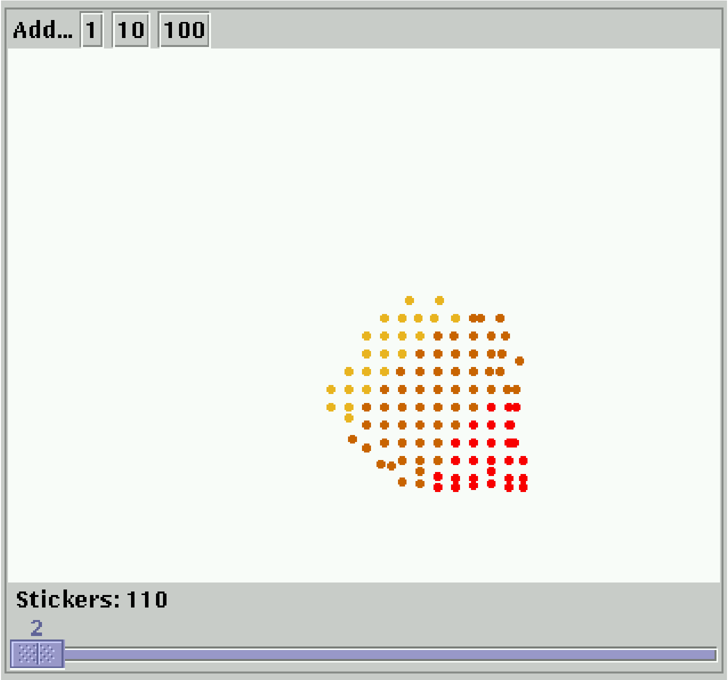
```
(define (http-eval s::socket)
   (let ((p (open-input-file (request-file-name s))))
      (http-get-input p s)
      ...))
(define (http-get-input p::input-port s::socket)
   (let loop ((res '())
              (num 0))
      (let ((v (thread-await! (make-input-signal p 1024))))
         (if (signal-eof? v)
             (apply http-reply s "html"
                    (reverse! (cons (signal-chars v) res)))
             (begin
               (thread-yield!)
               (loop (cons (signal-chars v) res)
                     (+ num (string-length (car v))))))))))
```

 Stickers (1/2)

That illustrates...

- Scheduler reactions
- Event synchronization
- External broadcast

The demonstration



Stickers (2/2)

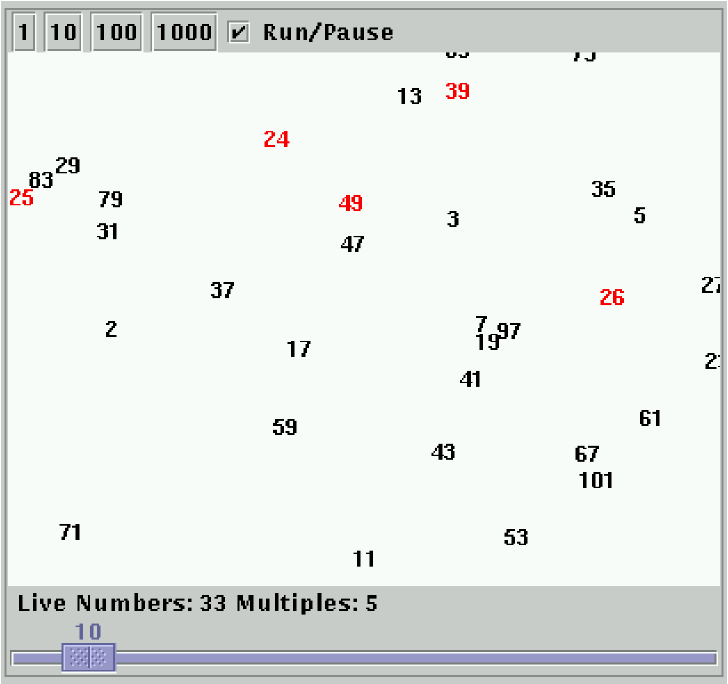
```
(define (motion-event e)
   (with-access::event e (widget x y)
      (if (and (>= x 0) (>= y 0))
          (scheduler-broadcast! *scheduler* 'target (cons x y)))))

(define (make-stick s)
   (lambda ()
      (let loop ()
         (move-sticker s)
         (colorize-sticker s)
         (let ((val (thread-await! 'target 1)))
            (if (pair? val)
               (begin
                  (with-access::sticker s (tx ty)
                     (set! tx (car val))
                     (set! ty (cdr val)))
                  (thread-yield!))))
         (loop))))
```

Primes (1/2)

That illustrates...

- Internal broadcast
- threads termination

The demonstration



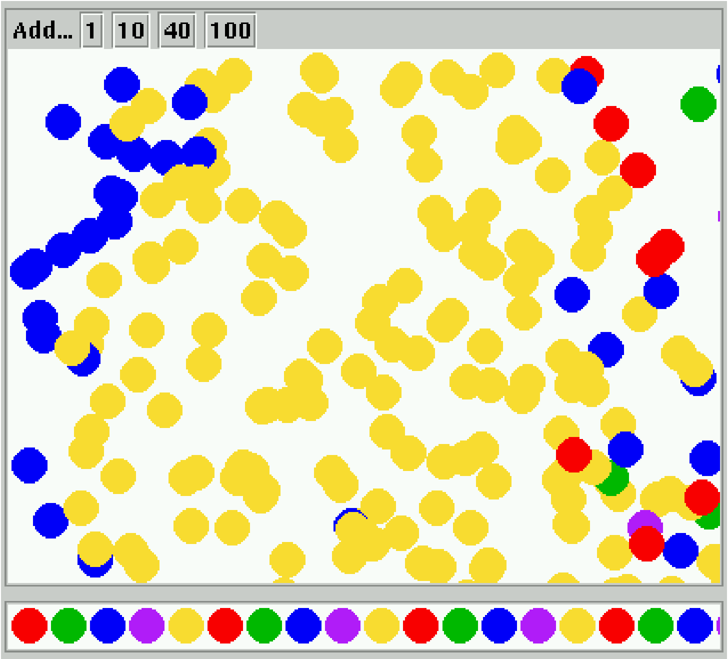
27/31 – Primes (2/2)

```
(define (make-num-prgm num)
   (lambda ()
      (let loop ()
         (move-num num)
         (broadcast! 'num-here num)
         (for-each (lambda (num2) (detect num num2))
                  (thread-get-values! 'num-here))
         (loop))))
```

That illustrates...

- Scheduler environments

The demonstration



The events

```
(define (make-evt a)
   (with-access::atom a (oldx oldy)
      (cons (/fx oldx *atom-size*) (/fx oldy *atom-size*))))
```

The threads

```
(define (make-atom-prgm a1)
   (lambda ()
      (let loop ()
         (move a1)
         (let ((evt (make-evt a1)))
            (broadcast! evt a1)
            (for-each (lambda (a2) (collision a1 a2))
                      (thread-get-values! evt)))
         (loop))))
```

Semantics Inside

$$\mathcal{E}[\![\texttt{(broadcast } \pi_1\ \pi_2\texttt{)}]\!]\rho\sigma\kappa = \\ (\mathcal{E}[\![\pi_1]\!]\ \rho\ \sigma\ \lambda\varepsilon_1\sigma_1.(\mathcal{E}[\![\pi_2]\!]\ \rho\ \sigma_1\ \lambda\varepsilon_2\sigma_2.(\kappa\ \varepsilon_2\ \sigma_2[\varepsilon_1/\varepsilon_2])))$$

$$\mathcal{E}[\![\texttt{(thread-yield!)}]\!]\rho\sigma\kappa = \\ \langle\texttt{(block } \kappa\texttt{)},\ \sigma,\ \emptyset\rangle$$

$$\mathcal{E}[\![\texttt{(thread-await } \pi\texttt{)}]\!]\rho\sigma\kappa = \\ (\mathcal{E}[\![\pi]\!]\ \rho\ \sigma\ \lambda\varepsilon\sigma_1.(\sigma_1\ \varepsilon)\neq ?\ \to (\kappa\ (\sigma_1\ \varepsilon)\ \sigma_1),\ \langle\texttt{(wait } \kappa\ \varepsilon\texttt{)},\ \sigma_1,\ \emptyset\rangle)$$

$$\mathcal{E}[\![\texttt{(make-thread! } \pi\texttt{)}]\!]\rho\sigma\kappa = \\ \langle\texttt{(ready } \kappa\texttt{)},\ \sigma,\ \mathit{list}(\texttt{(ready } \lambda\varepsilon\sigma.(\mathcal{E}[\![\texttt{(thread-terminate! } \pi\texttt{)}]\!]\ \rho\ \sigma\ \mathcal{K})))\rangle$$

$$\mathcal{E}[\![\texttt{(scheduler-start! } \pi^*\texttt{)}]\!]\rho\sigma\kappa = \\ (\kappa\ \mathit{schedule}(\mathit{map}(\lambda\pi.\texttt{(block } \lambda\varepsilon\sigma.(\mathcal{E}[\![\texttt{(thread-terminate! } \pi\texttt{)}]\!]\ \rho\ \sigma\ \mathcal{K})),\ \pi^*))\ \sigma)$$

$$\mathcal{S}[\![\texttt{(ready } \kappa\texttt{)}]\!]\tau\sigma T_1 = \\ \underline{\texttt{let}}\ \omega = (\kappa\ \mathit{unspecified}\ \sigma)\ \underline{\texttt{in}} \\ \langle\omega{\downarrow}_1,\ \omega{\downarrow}_2,\ T_1\S\omega{\downarrow}_3\rangle$$

$$\mathcal{S}[\![\texttt{(wait } \kappa\ \varepsilon\texttt{)}]\!]\tau\sigma T_1 = \\ (\sigma\ \varepsilon)\neq ?\ \to \underline{\texttt{let}}\ \omega = (\kappa\ (\sigma\ \varepsilon)\ \sigma),\ \tau = \omega{\downarrow}_1,\ \sigma_1 = \omega{\downarrow}_2,\ T_2 = \omega{\downarrow}_3\ \underline{\texttt{in}} \\ \langle\tau,\ \sigma_1,\ T_1\S T_2\rangle,\ \langle\tau,\ \sigma,\ T_1\rangle$$

$$\mathit{schedule} = \lambda T.\underline{\texttt{let}}\ T_1 = \mathit{scheduler\text{-}react}(T,\ \mathit{empty\text{-}event},\ \emptyset)\ \underline{\texttt{in}} \\ \mathit{finish?}(T_1) \to \mathit{map}(\lambda\tau.\mathit{thread\text{-}end\text{-}value}(\tau),\ T_1),\ \mathit{schedule}(T_1)$$

$$\mathit{scheduler\text{-}react} = \lambda T\sigma T_1.\mathit{instant\text{-}finish?}(T,\ \sigma) \to \mathit{next\text{-}ready}(T)\S T_1,\ \underline{\texttt{let}}\ \Omega = \mathit{it\text{-}list3}(\mathcal{S},\ T,\ \sigma,\ T_1)\ \underline{\texttt{in}} \\ \mathit{scheduler\text{-}react}(\Omega{\downarrow}_1,\ \Omega{\downarrow}_2,\ \Omega{\downarrow}_3)$$

$$\mathit{instant\text{-}finish?} = \lambda T\sigma.\forall\tau\in T,\ \mathit{thread\text{-}ready?}(\tau) \to \mathit{false},\ \mathit{thread\text{-}wait?}(\tau) \to (\sigma\ \mathit{thread\text{-}wait\text{-}signal}(\tau))=?,\ \mathit{true}$$

$$\mathit{next\text{-}ready} = \lambda T.\mathit{map}(\lambda\tau.\mathit{thread\text{-}block?}(\tau) \to \texttt{(ready } \mathit{thread\text{-}cont}(\tau)\texttt{)},\ \tau,\ T)$$

Conclusion

- We have added cooperative FairThreads to Bee
- It supports for cooperative (*i.e.*, non blocking) IOs
- It supports full SRFI-18
- It is available (`http://www.inria.fr/mimosa/fp/Bigloo`)