Programmation réactive en OCaml
Implémentation de la bibliothèque TML

Christophe Deleuze
Laboratoire de Conception et d'Intégration des Systèmes (LCIS)
50, rue Barthélémy de Laffemas, BP54
26902 Valence Cedex 09
France

Juillet 2009

Résumé

Le langage ReactiveML est une extension de OCaml proposant des constructions réactives. Nous proposons une réalisation de ces constructions réactives dans le cadre du langage OCaml pur. Les processus doivent pour cela être rédigés en style *trampolined* et une bibliothèque (appelée TML) met en œuvre l'ordonnancement et les communications. Ce document est une description détaillée de l'implémentation de cette bibliothèque.

Introduction

La bibliothèque TML, qui permet d'ajouter à OCaml des fonctionnalités de programmation réactive sur le modèle de ReactiveML est décrite dans [1]. Ce document décrit en détail son implémentation. Nous en réalisons une construction progressive, par ajout incrémental de fonctionnalités. Les fragments de code utilisés dans les versions ultérieures sont repérés par des noms sur le côté droit de la page. Le texte est rédigé en style *litterate programming*, le code des modules étant extrait de ce document. Des détails sont donnés en annexe A.

Nous commencerons par la réalisation de l'exécution concurrente de processus par instant (section 1), puis nous ajouterons la communication par diffusion de signaux non valués (section 2), de signaux mono-valués (une seule émission par instant, section 3), de signaux multi-valués (section 4), et enfin les constructions de contrôle. Ces dernières étant assez difficiles à mettre en œuvre nous procéderons en deux étapes (sections 5 et 6), la première n'autorisant pas l'imbrication de ces constructions.

1 Processus parallèles

1.1 Données de base

Un processus est une fonction prenant un *unit*, il coopére en retournant une valeur de type *coop*, qui contient soit rien du tout (processus terminé), soit un processus à lancer à l'instant suivant. Dans cette première version les processus ne peuvent être qu'en attente d'éxécution (liste *runq*) ou en pause, en attente du prochain instant (liste *pauseq*). Nous allons définir un ensemble de **primitives** qui peuvent être utilisées par les processus, et de fonctions internes à la bibliothèque. On définit les deux premières primitives :

– *term* : fin du processus.
– *pause* : suspension du processus et attente de l'instant suivant.

L'ordonnanceur est défini en deux parties : une boucle activant les processus de manière répétée et gérant les listes, et une procédure appelée à la fin d'instant, préparant l'exécution de l'instant suivant.

La fin d'instant est atteinte quand tous les processus ont terminé leur exécution pour l'instant. Ils sont alors tous dans la liste *pauseq*, la liste *runq* étant vide. Pendant chaque instant l'ordonnanceur retire un à un les processus de *runq* et les exécute.

```
type process = unit → coop
and coop = Done | Pause of process

let runq = ref []
and pauseq = ref []

let term () = Done
let pause p = Pause p

let next_instant () =
  runq := !pauseq; pauseq := []

let rec sched () =
  match !runq with
  | [] → next_instant (); if !runq = [] then () else sched ()
  | h :: t → begin
      runq := t;
      match h () with
      | Done → sched ()
      | Pause p → pauseq := p :: !pauseq; sched ()
  end
```

1.2 Composition parallèle

La fonction *spawn* ajoute un processus dans la liste *runq* et est utilisée comme base pour les primitives de composition parallèle.

SPAWN

```
let spawn p = runq := p :: !runq
```

...

PAR ↑

La primitive *par* permet la composition parallèle de deux processus. Le troisième processus, la continuation de la composition, est exécuté quand les deux processus composés sont terminés. *p1* et *p2* doivent accepter un seul paramètre qui est leur fonction de continuation. Ils n'ont donc pas le type *process* mais sont bien des processus au sens où nous l'entendons.

```
let par_help k1 k2 k3 =
  let you_re_last = ref false in
  let test _ =
    if !you_re_last then k3 () else begin
        you_re_last := true;
        term()
      end
  in
  (fun () → k1 test),
  (fun () → k2 test)

let par p1 p2 k =
  let k1, k2 = par_help p1 p2 k
  in
  spawn k1;
  k2 ()
```

La primitive *tail_par* fait la même chose sans processus de continuation, elle est utile en particulier dans

le cas où l'on compose des processus qui ne terminent jamais. On pourrait la définir simplement par let *tail_par p1 p2 = par p1 p2 term*, mais la définition donnée ici est plus efficace puisqu'on n'a pas besoin de se synchroniser sur la fin des deux processus. Ici, *p1* et *p2* ne prennent pas de fonction de continuation en paramètre.

```
let tail_par p1 p2 =
  spawn p1;
  p2 ()
```

On généralise la composition parallèle à une liste de n processus, en version avec et sans continuation (avec et sans *tail*).

```
let tail_parn (p :: ps) =
  List.iter spawn ps;
  p ()

let parn (p :: ps) k =
  let cpt = ref (List.length ps) in
  let pk :: pks = List.map
              (fun p →
                fun () → p
                     (fun () → if !cpt = 0 then k () else
                     begin decr cpt; term() end)
                     ())
              (p :: ps)
  in
  List.iter spawn pks;
  pk ()
```

La fonction *fordopar* fait une chose similaire avec une touche plus impérative : elle exécute en parallèle un processus pour chaque itération, chaque processus recevant en argument la valeur actuelle du compteur de boucle ainsi qu'une fonction de continuation. Elle existe elle aussi en version *tail*.

```
let fordopar e1 e2 pi k =
  let cpt = ref (e2 − e1 + 1) in
  let finish () = decr cpt; if !cpt = 0 then k () else term()
  in
  for i = e1 to e2 do
    spawn (fun () → pi i finish)
  done;
  term ()

let tail_fordopar e1 e2 pi =
  for i = e1 to e2 do
    spawn (fun () → pi i)
  done;
  term ()
```

Enfin, on définit la composition parallèle avec récupération de valeurs à la terminaison des processus (le let/and de ReactiveML). Ici *e1* et *e2* retournent des valeurs de type α et β, k étant une continuation prenant un argument de type $\alpha \times \beta$. Ici encore l'ordonnanceur n'est sollicité que pour la création (*spawn*) d'un processus.

```
let letand_help p1 p2 k =
  let r1 = ref None
  and r2 = ref None
  in
  p1
```

```
    (fun r → match !r2 with
    | None → r1 := Some r; term()
    | Some v2 → k (r, v2)),
  p2
    (fun r → match !r1 with
    | None → r2 := Some r; term()
    | Some v1 → k (v1, r))

let letand p1 p2 k =
  let k1, k2 = letand_help p1 p2 k
  in
  spawn k1;
  k2 ()
```

... ↓

1.3 Pour finir...

... avec la fonction *start* qui prend une liste de processus en paramètre et démarre le système avec ces processus.

START

```
let start pl =
  runq := pl;
  sched ()
```

...

2 Signaux non valués

Nous ajoutons maintenant la possibilité pour les processus de communiquer au moyen de signaux non valués. Les primitives additionnelles seront :
- *signal* création d'un signal
- *emit* émission d'un signal
- *await* attente d'un signal : processus débloqué à l'instant suivant l'émission du signal
- *await_immediate* attente immédiate : processus débloqué à l'instant d'émission du signal
- *present* test de la présence d'un signal : si présent, partie then exécutée dans l'instant sinon partie else exécutée à l'instant suivant
- *pre* status du signal à l'instant précédent

2.1 Types

Un processus se suspend quand il se place en attente d'un signal, on étend la définition des coopérations pour prendre en compte ces cas. Un signal est défini comme une structure de champs mutables. Le status du signal est encodé par le numéro du dernier instant où le signal a été émis, ce qui a l'avantage que les signaux sont automatiquement rétablis à "non émis" à chaque début d'instant.[1] Le numéro de l'instant courant est contenu dans la référence entière *instant*.

Les trois listes contiennent les processus en attente sur ce signal, pour chacune des trois primitives. Ainsi, un parcours des listes au moment opportun permettra de relancer les processus qui doivent l'être. Ceci évite de mettre en œuvre une attente active où les processus testent régulièrement si la condition attendue est réalisée, ce qui serait très mauvais en terme de performances.

[1] On ne peut pas totalement exclure qu'un rebouclage du compteur d'instant entraîne une erreur, mais cela parait hautement improbable en pratique.

```
type process = unit → coop
and coop =
  | Done
  | Pause of process
  | Wait of signal_t × process
  | Wait_im of signal_t × process
  | Present of signal_t × process × process

and signal_t = {
    mutable last_emit : int;
    mutable prev_emit : int;
    mutable used : bool;

    mutable await : process list;
    mutable await_im : process list;
    mutable present : (process × process) list
  }
```

DEF2

```
let instant = ref 1

let runq = ref []
and pauseq = ref []

let run f = f ()
let term () = Done
let pause p = Pause p

let put_in_runq p = runq := p :: !runq
```

...

2.2 Primitives

La création d'un signal consiste simplement à créer une valeur de type *signal_t* avec les bonnes valeurs initiales. Les signaux *utilisés*[2] à un instant donné seront stockés dans la liste *used_signals*.

```
let signal () =
  { last_emit = −1; prev_emit = −1; used = false;
    await = []; await_im = []; present = [] }
```

USED

```
let used_signals = ref ([] : signal_t list)

let signal_is_used s =
  if ¬ s.used then begin
    s.used ← true;
    used_signals := s :: !used_signals
  end
```

...

La première émission d'un signal pour un instant donné marque le signal comme émis, utilisé, met à jour le champ codant le status à l'instant précédent et vide les files *await_im* et *present* dans *runq*.

[2] Nous définissons cette notion en 2.3.

```
let emit s v =
  if s.last_emit ≠ !instant then begin
    s.prev_emit ← s.last_emit;
    s.last_emit ← !instant;
    signal_is_used s;
    List.iter (fun p → put_in_runq p) s.await_im;
    s.await_im ← [];
    List.iter (fun (p1, p2) → put_in_runq p1) s.present;
    s.present ← []
  end
```

Voici les définitions des trois primitives d'attente. On note que pour *await_immediate* et *present* le processus continue immédiatement son exécution si le signal est présent. Il ne se suspend que dans le cas contraire, auquel cas l'ordonnanceur le placera dans la file d'attente correspondante.

```
let await s p = Wait(s, p)

let await_immediate s p =
  if s.last_emit = !instant then p ()
  else Wait_im(s, p)

let present s p1 p2 =
  if s.last_emit = !instant then p1 ()
  else Present(s, p1, p2)
```

Le signal a été émis à l'instant précédent si le numéro d'instant précédent se trouve dans le champ *prev_emit* (cas où le signal a été émis à l'instant courant) ou dans *last_emit* (cas où le signal n'a pas – encore – été émis à l'instant courant).

```
let pre s = s.prev_emit = !instant − 1 ∨ s.last_emit = !instant − 1
```

2.3 Gestion des signaux par l'ordonnanceur

L'ordonnanceur devra gérer les signaux. En fin d'instant il devra débloquer certains processus placés en attente dans leurs listes.

Cependant il n'est pas nécessaire que l'ordonnanceur se préoccupe de tous les signaux à chaque fin d'instant. On dira qu'un signal est *utilisé* dans un instant donné, si un processus a émi ce signal, ou bien si un ou plusieurs processus ont fait une opération de test (*present*) sur ce signal. Dans le cas contraire, le signal n'est pas *utilisé*.[3]

En fin d'instant, pour chaque signal utilisé :
– si le signal est présent, on déplace en *runq* tous les processus en *await*,
– si le signal est absent, pour chaque entrée de la liste *present* on déplace en *runq* le processus de la branche else.

Note : si le signal est présent, les listes *await_immediate* et *present* sont nécessairement vides. Si le signal est absent les listes *await* et *await_immediate* n'ont pas à être modifiées, puisque les processus resteront en attente du signal à l'instant suivant.

Les signaux non utilisés n'ont pas à être pris en compte en fin d'instant. Ainsi la quantité de traitement en fin d'instant ne dépendra pas du nombre total de signaux mais seulement du nombre de signaux utilisés (au sens défini ci-dessus).

[3] Donc, selon cette définition, un signal sur lequel un processus est en attente depuis un ou plusieurs instants mais sur lequel, à l'instant courant, aucune émission ni test n'a été effectué, n'est pas utilisé.

```
let next_instant () =
  runq :=!pauseq; pauseq := [];
  List.iter
    (fun s → if s.last_emit = !instant then begin
      List.iter (fun p → put_in_runq p) s.await;
      s.await ← []
    end else begin
      List.iter (fun (p1, p2) → put_in_runq p2) s.present;
      s.present ← []
    end;
    s.used ← false)
    !used_signals;
  incr instant;
  used_signals := []
```

2.4 Boucle de l'ordonnanceur

Le corps de l'ordonnanceur prend en compte simplement les nouveaux cas de suspension d'un processus.

ORDO2

```
let rec sched () =
  match !runq with
  | [] →
      next_instant ();
      if !runq = [] then () else sched ()
  | p :: t → begin runq := t; match p () with
    | Done → sched ()
    | Pause p →
        pauseq := p :: !pauseq;
        sched ()
    | Wait(s, p) →
        s.await ← p :: s.await;
        sched ()
    | Wait_im(s, p) →
        s.await_im ← p :: s.await_im;
        sched()
    | Present(s, p1, p2) →
        s.present ← (p1, p2) :: s.present;
        signal_is_used s;
        sched()
  end
```

...

La composition parallèle reste inchangée. → SPAWN, sec 1.2, page 2. → PAR, sec 1.2, page 2. → START, sec 1.3, page 4.

3 Signaux mono-valués

Nous voulons maintenant pouvoir émettre des valeurs sur les signaux. Pour commencer, nous considérerons que le type de valeur est le même pour tous les signaux, nous verrons ensuite comment permettre des types distincts. Les modifications vont porter sur :

– ajout des valeurs courante et précédente dans la structure *signal_t*
– spécification de la valeur à l'émission
– récupération de la valeur du signal

3.1 Type unique des signaux

Nous définissons *sval_t* comme étant le type des valeurs émises sur les signaux. Nous le définissons pour le moment égal à *int*. Les processus qui attendent l'émission d'un signal doivent maintenant récupérer sa valeur. Ils donneront donc en paramètre non pas une valeur de type *process*(ie *unit* → *coop*) mais une fonction de type *sval_t* → *coop*, que nous appelerons aussi processus. Ceci se reflète dans la définition de *coop* et des types des listes de processus en attente sur un signal.

```
type sval_t = int

type process = unit → coop
and coop =
  | Done
  | Pause of process
  | Wait of signal_t × (sval_t → coop)
  | Wait_im of signal_t × (sval_t → coop)
  | Present of signal_t × process × process
and signal_t = {
    mutable last_emit : int;
    mutable prev_emit : int;
    mutable value : sval_t;
    mutable prev_value : sval_t;
    mutable used : bool;
    mutable await : (sval_t → coop) list;
    mutable await_im : (sval_t → coop) list;
    mutable present : (process × process) list
  }
```

→ DEF2, sec 2.1, page 5.

L'émission sur un signal prend la valeur émise en paramètre et modifie les champs *pre_value* et *value*. Dans le cas où plusieurs émissions ont lieu au cours du même instant, la valeur sera la première valeur émise, de manière non déterministe.

prev (le pre ?*s* de ReactiveML) donne la dernière valeur émise sur le signal à un instant précédent. Si le signal a été émis au cours de l'instant courant, c'est la valeur contenue dans *prev_value*, sinon dans *value*. Tant que le signal n'a pas été emis, c'est la valeur par défaut.

```
let signal () =
  { last_emit = −1; prev_emit = −1; value = 0; prev_value = 0;
    used = false; await = []; await_im = []; present = [] }

let emit s v =
  if s.last_emit ≠ !instant then begin
    s.prev_emit ← s.last_emit;
    s.last_emit ← !instant;
    s.prev_value ← s.value;
    s.value ← v;
    signal_is_used s;
    List.iter (fun p → put_in_runq (fun () → p v)) s.await_im;
    s.await_im ← [];
    List.iter (fun (p1, p2) → put_in_runq p1) s.present;
    s.present ← []
```

```
  end

let ispst = fun s → s.last_emit = !instant

let await_immediate s p =
  if ispst s then p s.value
  else Wait_im(s, p)

let await s p = Wait(s, p)

let present s p1 p2 =
  if ispst s then p1 ()
  else Present(s, p1, p2)

let pre s = s.prev_emit = !instant − 1 ∨ s.last_emit = !instant − 1

let prev s = if s.last_emit = !instant then s.prev_value else s.value
```

Seul changement pour la fonction de fin d'instant : elle passe la valeur du signal aux processus qui étaient en attente sur un signal présent.

```
let next_instant () =
  runq := !pauseq; pauseq := [];
  List.iter
    (fun s → if s.last_emit = !instant then begin
      List.iter (fun p → put_in_runq (fun () → p s.value)) s.await;
      s.await ← []
    end else begin
      List.iter (fun (p1, p2) → put_in_runq p2) s.present;
      s.present ← []
    end;
    s.used ← false)
    !used_signals;
  incr instant;
  used_signals := []
```

Aucune modification n'est nécessaire sur la boucle de l'ordonnanceur.

→ ORDO2, sec 2.4, page 7.

La composition parallèle reste inchangée. → SPAWN, sec 1.2, page 2. → PAR, sec 1.2, page 2. → START, sec 1.3, page 4.

3.2 Polymorphisme des signaux

À chaque signal correspond un type, celui des valeurs qui peuvent être émises sur ce signal. Cela signifie que les champs de la structure contenant la valeur courante et précédente du signal devraient avoir un type dépendant du signal. Ce seraient donc des types différents et l'on ne pourrait plus gérer la liste des signaux utilisés à chaque instant. Plutôt que de gérer une liste par type, on construira un type somme ad'hoc. Bien entendu, un système de type pour les signaux serait bien plus confortable et sur... mais cela demanderait d'étendre le langage.

Le module sera défini comme un foncteur prenant le type somme des valeurs de signaux comme module paramètre. On aura donc au début de notre module :

BFONCTEUR

```
module type SVAL_TYPE = sig type t end

module Gen_TML = functor(SV : SVAL_TYPE) →
  struct
    type sval_t = SV.t
```

...

et en fin, la fermeture du foncteur :

```
                                                                    EFONCTEUR

  end

                                                                            ...
```

En fait il subsiste le problème de la valeur initiale des signaux (primitive *signal*)[4] mais il se résoudra de lui même dans le cas des signaux multi-valués.

4 Signaux multi-valués

On ajoute maintenant la multi-émission, c'est à dire la possibilité d'avoir plusieurs émissions sur le même signal pendant un instant. La valeur associée au signal est alors la *collecte* des valeurs émises. La façon dont les valeurs sont collectées peut être fixée à la définition du signal, à l'aide d'un couple *(init, gather)* où *init* est la valeur initiale du signal et *gather* une fonction construisant la valeur finale du signal à partir de la liste des valeurs émises durant l'instant.[5]

Deux nouvelles primitives sont ajoutées :

– *await_one* : identique à *await*, mais ne récupère qu'*une* des valeurs émises (choix non déterministe),
– *await_immediate_one* : identique à *await_immediate* mais récupère une quelconque des valeurs émises.

4.1 Mise en œuvre

Le type de la valeur finale d'un signal peut maintenant être différent du type des valeurs émises sur le signal. On pourrait ajouter des cas "listes" à notre type somme ad'hoc, mais on préfère considérer que la valeur (telle que stockée dans la structure *signal_t*) est la liste des valeurs émises. La valeur finale ne sera pas stockée mais produite en fin d'instant.

4.2 Les différents *await*

Il existe donc maintenant quatre primitives de la famille *await* où le processus recevra comme valeur :
– *await* : une liste produite par la fonction *gather* du signal,
– *await_one* : une des valeurs émises,
– *await_immediate* : aucune valeur,
– *await_immediate_one* : une des valeur émises.
Selon les cas, le processus recevra une liste, une valeur simple ou rien. Plutôt que de créer dans *signal_t* une liste pour chacun de ces cas, on construit le type *awproc*, type somme des processus n'attendant aucune valeur, une valeur, ou toutes (ie une liste) les valeurs.

```
type process = unit → coop
and coop =
  | Done
  | Pause of process
  | Wait of signal_t × awproc
  | Wait_im of signal_t × awproc
  | Present of signal_t × process × process
and awproc =
  | No of (unit → coop)
  | One of (sval_t → coop)
  | All of (sval_t list → coop)
```

[4] Que nous pourrions traiter en utilisant le type *sval_t option*.
[5] Cela diffère légèrement, mais est équivalent, à la façon dont les choses sont définies dans ReactiveML.

```
and signal_t = {
    mutable last_emit : int;
    mutable prev_emit : int;
    mutable value : sval_t list;
    mutable prev_value : sval_t list;
    mutable used : bool;
    gather : (sval_t list → sval_t list);

    mutable await : awproc list;
    mutable await_im : awproc list;
    mutable present : (process × process) list
  }
```

La primitive *signal* crée un signal avec la fonction de collecte par défaut, l'identité (la valeur finale sera alors la liste des valeurs émises), *signalc* avec la fonction spécifiée en paramètre. Les valeurs émises sont accumulées dans une liste, la fonction de collecte ne sera appliquée qu'en fin d'instant pour construire la valeur finale du signal.

SIGNAL

```
let signal () =
  { last_emit = −1; prev_emit = −1; value = []; prev_value = [];
    used = false; gather = (fun x → x); await = []; await_im = [];
    present = []
  }
let signalc g =
  { last_emit = −1; prev_emit = −1; value = []; prev_value = [];
    used = false; gather = g; await = []; await_im = [];
    present = []
  }
```

...

→ USED, sec 2.2, page 5. → DEF2, sec 2.1, page 5.

Les définitions des quatre primitives en découlent sans difficulté. Le *await_immediate_one* recevra la dernière valeur émise au moment où il est exécuté. Ceci parce que c'est le plus simple à réaliser, la spécification indique que l'une quelconque des valeurs sera reçue.

pre et *prev* ne changent pas.

AWAIT4

```
let ispst = fun s → s.last_emit = !instant

let await s p = Wait(s, All p)

let await_one s p = Wait(s, One p)

let await_immediate s p =
  if ispst s then p () else Wait_im(s, No p)

let await_immediate_one s p =
  if ispst s then p (List.hd s.value) else Wait_im(s, One p)

let present s p1 p2 =
  if ispst s then p1 () else Present(s, p1, p2)

let pre s = s.prev_emit = !instant − 1 ∨ s.last_emit = !instant − 1

let prev s = if s.last_emit = !instant then s.prev_value else s.value
```

...

Le code pour *emit* est modifié conformément au nouveau type de la liste *await_im*.

```
let emit s v =
  if s.last_emit ≠ !instant then begin
    s.prev_emit ← s.last_emit;
    s.last_emit ← !instant;
    s.prev_value ← s.value;
    s.value ← [];
    signal_is_used s
  end;
  s.value ← v :: s.value;
  List.iter
    (fun awp → match awp with
    | No p → put_in_runq p
    | One p → put_in_runq (fun () → p v))
    s.await_im;
  s.await_im ← [];
  List.iter (fun (p1, p2) → put_in_runq p1) s.present;
  s.present ← []
```

...

4.3 Fin d'instant

Le seul changement de la fonction de fin d'instant permet de distinguer les processus en *await* de ceux en *await_one*.

```
let next_instant () =
  runq :=!pauseq; pauseq := [];

  List.iter
    (fun s → if ispst s then begin (* s présent *)
      (* parcours de s.await *)
      let v = s.gather s.value in
      let ov = List.hd s.value
      in
      List.iter
        (fun qp → match qp with
        | One p → put_in_runq (fun () → p ov)
        | All p → put_in_runq (fun () → p v))
        s.await;
      s.await ← []

    end else begin (* s absent *)

      List.iter
        (fun (p1, p2) → put_in_runq p2)
        s.present;
      s.present ← [];
    end;
      s.used ← false)

    !used_signals;

  incr instant;
  used_signals := []
```

4.4 Suite

L'ordonnanceur est absolument identique à la version précédente. → ORDO2, sec 2.4, page 7.

La composition parallèle reste aussi inchangée. → SPAWN, sec 1.2, page 2. → PAR, sec 1.2, page 2. → START, sec 1.3, page 4.

5 Constructions de contrôle

ReactiveML définit trois constructions de contrôle :

- do $< p >$ until s done (préemption) : $< p >$ est définitivement interrompu à la fin d'instant si s est présent, et sa variante do $< p >$ until $s(v) \rightarrow < e >$ done (préemption avec récupération de valeur) : $< p >$ est définitivement interrompu à la fin d'instant si s est présent. $< e >$ est alors exécuté avec la valeur du signal dans v.
- control $< p >$ with s done (désactivation) : $< p >$ est désactivé ou réactivé en fin d'instant chaque fois que s est présent.
- do $< p >$ when s done (suspension) : $< p >$ n'est n'exécuté qu'aux instants où s est présent.

5.1 Contextes d'exécution

Pour les mettre en œuvre il faut introduire la notion de contexte d'exécution d'un processus. Dans un premier temps, on se limitera au cas simplifié où ces constructions ne peuvent pas être imbriquées. Cela nous donne la définition du type *ctxt* ci-dessous. On ajoute une entrée *Next* au type *coop* correspondant à un processus changeant de contexte (voir plus loin les primitives de contrôle).

```
type process = unit → coop
and coop =
  | Done
  | Next
  | Pause of process
  | Wait of signal_t × awproc
  | Wait_im of signal_t × awproc
  | Present of signal_t × process × process

and awproc =
  | No of (unit → coop)
  | One of (sval_t → coop)
  | All of (sval_t list → coop)

and signal_t = {
    mutable last_emit : int;
    mutable prev_emit : int;
    mutable value : sval_t list;
    mutable prev_value : sval_t list;
    mutable used : bool;
    gather : (sval_t list → sval_t list);

    mutable await : (ctxt × awproc) list;
    mutable await_im : (ctxt × awproc) list;
    mutable present : (ctxt × process × process) list
  }
```

```
and ctxt =
    Free
  | Until of signal_t × process
  | Untilv of signal_t × (sval_t list → coop) × process
  | Control of signal_t × process
  | When of signal_t × awproc
```

→ DEF2, sec 2.1, page 5.

Nous ajoutons la fonction *put_in_await_im* qui sera utile pour les processus suspendus.

AWAITIM

```
let put_in_await_im s p =
  s.await_im ← p :: s.await_im
```

...

→ SIGNAL, sec 4.2, page 11. → USED, sec 2.2, page 5. → AWAIT4, sec 4.2, page 11.

L'émission est identique à 4.2 en ajoutant le contexte associé au processus dans chaque file d'attente.

```
let emit s v =
  if s.last_emit ≠ !instant then begin
    s.prev_emit ← s.last_emit;
    s.last_emit ← !instant;
    s.prev_value ← s.value;
    s.value ← [];
    signal_is_used s
  end;
  s.value ← v :: s.value;
  List.iter
    (fun (c, awp) → match awp with
    | No p → put_in_runq (c, p)
    | One p → put_in_runq (c, fun () → p v))
    s.await_im;
  s.await_im ← [];
  List.iter (fun (c, p1, p2) → put_in_runq (c, p1)) s.present;
  s.present ← []
```

Les primitives de contrôle insèrent dans *runq* le processus courant avec son nouveau contexte puis terminent avec le status *Next*. Le processus sera à nouveau exécuté dans l'instant courant dans son nouveau contexte. Nous noterons généralement k les processus continuations des constructions de contrôle (bien qu'ils ne différent pas des autres processus que nous notons p).

```
let dountil s e k =
  put_in_runq (Until(s, k), e); Next

let dountilv s e e2 k =
  put_in_runq (Untilv(s, e2, k), e); Next

let controlwith s e k =
  put_in_runq (Control(s, k), e); Next

let dowhen s e k =
  if s.last_emit = !instant then put_in_runq (When(s, No k), e)
  else put_in_await_im s (When(s, No k), No e);
  Next
```

5.2 Mise en œuvre de la suspension

Un processus suspendu ne peut s'exécuter au cours d'un instant que si le signal associé est présent. Pour mettre en œuvre ce comportement, on placera en début d'instant le processus dans la liste *await_im* du signal. Si le signal est absent, il y restera. S'il est présent, il sera exécuté normalement. Il faudra l'y remettre en fin d'instant afin qu'il reste suspendu au signal.

5.3 Limiter les traitements de fin d'instant

La mise en œuvre de ces constructions de contrôle apporte de la complexité et va avoir un impact sur la performance du système. Notre but est de faire en sorte que cet impact soit le plus limité possible, en particulier sur les traitements de fin d'instant. En bref nous voulons que des traitements supplémentaires soient ajoutés autant que possible uniquement aux processus concernés et pas à l'ensemble des processus.

En fin d'instant un processus peut se trouver :
– dans *pauseq*,
– dans une liste d'un signal

En l'absence de constructions de contrôle, les traitements de fin d'instant sont les suivants :

– copie de *pauseq* dans *runq* (temps constant),
– pour tous les signaux *utilisés*,
  – si présent, parcours de la liste *await*,
  – si absent, parcours de la liste *present*.

Autrement dit, on ne parcourt qu'une seule liste pour chaque signal utilisé.

Avec les constructions de contrôle, en fin d'instant, pour les signaux présents :

1. les processus préemptibles doivent être préemptés,
2. les processus désactivables actifs doivent être désactivés,
3. les processus inactifs doivent être activés,
4. les processus suspensibles doivent être replacés dans la liste *await_im* de leur signal.

Ces différents points impliquent à priori de parcourir *toutes* les listes (*pauseq* et les listes de processus des signaux [6]).

Nous pouvons facilement supprimer le besoin de parcourir la liste *pauseq*. Il suffit pour cela qu'à l'exécution d'une pause, les processus préemptibles et désactivables soient placés dans une liste dédiée *pause2* et les processus suspensibles directement replacés dans leur *await_im*[7] où ils attendront l'émission éventuelle du signal à l'instant suivant. Quand aux processus inactifs, ils seront placés dans la liste *unactive*. Ainsi, *pauseq* ne contiendra que des processus libres ne nécessitant aucun traitement, et on pourra se contenter de la transférer dans *runq*, en coût constant. *pause2* devra par contre être parcourue, ce qui sera peu coûteux si la plupart des processus sont libres.

```
let pause2 = ref [] (* pour les proc. suspensibles et désactivables *)
let unactive = ref [] (* pour les proc. désactivés *)

let current_context = ref Free

let put_in_unact p = unactive := p :: !unactive
```

De plus, nous redéfinissons la notion de signal *utilisé* pour inclure, en plus des signaux émis ou sur lesquels un test de présence est fait dans l'instant, les signaux sur lesquels au moins un processus préemptible ou désactivable est en attente (c'est à dire dans la liste *await* ou *await_im* du signal, les processus ne restent pas plus d'un instant dans la liste *present*, si elle n'est pas vide le signal est forcément marqué utilisé).

[6] Et pas seulement celles des signaux utilisés au sens défini précédemment. Un processus préemptible sur s peut être en attente d'un signal s' non émis depuis plusieurs instants. Il devra malgrès tout être préempté si s est émis.

[7] S'ils ont été exécutés, c'est que leur signal est présent, la liste ne sera donc plus parcourue pendant cet instant.

5.4 Fin d'instant

La fin d'instant se complique sérieusement. On commence en parcourant la liste *unactive* pour transférer dans *runq* tous les processus désactivés dont le signal de contrôle est présent, puis en traitant le cas de *pause2*. Les processus préemptibles sont soit préemptés (ce qui revient à placer leur continuation dans *runq*), soit placés dans *runq*, les processus désactivables sont soit remis dans *runq* soit désactivés.

```
let ispst = fun s → s.last_emit = !instant

let next_instant_unactive_pause2 () =

  unactive := List.filter
      (fun (Control(s', k) as c, p) →
        if ispst s' then (put_in_runq (c, p); false)
        else true)
      !unactive;

  List.iter
    (fun (c, p) → match c with
    | Until(s', k) → put_in_runq
          (if ispst s' then (Free, k) else (c, p))

    | Untilv(s', e2, k) →
        put_in_runq
          (if ispst s'
          then (Free, fun () → e2 s'.value; k())
          else (c, p))

    | Control(s', k) →
        (if ispst s' then put_in_unact else put_in_runq) (c, p)
    )
    !pause2;
  pause2 := []
```

Nous devrons ensuite parcourir la liste des signaux utilisés et traiter les processus se trouvant dans les diverses listes.

Signaux absents Voici le traitement de la liste *await* pour un signal absent s. Les processus s'exécutant dans un contexte libre y restent, par contre les processus suspensibles ($When(s', k)$) sont déplacés dans $s'.await\_im$ et re-attendent s (insertion de l'instruction d'attente en tête du processus). Les processus préemptibles sont préemptés le cas échéant, leur continuation est alors placée dans *runq*. Les processus désactivables sont déplacés dans *unactive* le cas échéant.

En plus de modifier la liste, la fonction retourne un booléen indiquant si au moins un processus préemptible ou désactivable est en attente. Dans ce cas, le signal restera marqué comme utilisé à l'instant suivant.

```
let do_await_queue_absent s =
  let new_list, used = List.fold_left
      (fun (acc, used) (c, qp) → match c with
      | Free → (Free, qp) :: acc, used

      | When(s', k) →
          let next = match qp with
          | One p → fun () → await_one s p
          | All p → fun () → await s p
          in
          put_in_await_im s' (c, No next); acc, used
```

```
      | Until(s', k) →
          if ispst s' then (put_in_runq (Free, k); acc, used)
          else (c, qp) :: acc, true

      | Untilv(s', e2, k) →
          if ispst s' then
            (put_in_runq (Free, fun () → e2 s'.value; k()); acc, used)
          else
            (c, qp) :: acc, true

      | Control(s', k) →
          let next = match qp with
          | One p → fun () → await_one s p
          | All p → fun () → await s p
          in
          if ispst s' then (put_in_unact (c, next); acc, used)
          else (c, qp) :: acc, true
      )

      ([], false)
      s.await
  in s.await ← new_list;
  used
```

Le traitement de la liste *await_im* d'un signal absent est quasiment identique.

```
let do_await_im_queue_absent s =
  let new_list, used = List.fold_left
      (fun (acc, used) (c, qp) → match c with
      | Free → (Free, qp) :: acc, used

      | When(s', k) →
          let next = match qp with
          | One p → fun () → await_immediate_one s p
          | No p → fun () → await_immediate s p
          in
          put_in_await_im s' (c, No next); acc, used

      | Until(s', k) →
          if ispst s' then (put_in_runq (Free, k); acc, used)
          else (c, qp) :: acc, true

      | Untilv(s', e2, k) →
          if ispst s' then
            (put_in_runq (Free, fun () → e2 s'.value; k()); acc, used)
          else
            (c, qp) :: acc, true

      | Control(s', k) →
          let next = match qp with
          | One p → fun () → await_immediate_one s p
          | No p → fun () → await_immediate s p
          in
          if ispst s' then (put_in_unact (c, next); acc, used)
          else (c, qp) :: acc, true
      )
      ([], false)
      s.await_im
  in s.await_im ← new_list; used
```

Signaux présents Traitement de la liste *await* d'un signal présent : les suspendus sont replacés dans leur file *await_im*, les préemptibles non préemptés sont placés dans *runq*, de même que la continuation des préemptés. Les processus libres passent aussi en *runq*. v et ov représentent la valeur (respectivement complète et "one") du signal.

```
let do_await_queue_present s v ov =
    List.iter
      (fun (c, qp) →
        let next = match qp with
        | One p → fun () → p ov
        | All p → fun () → p v
        in
        match c with
      | Free → put_in_runq (Free, next)
      | When(s', k) → put_in_await_im s' (c, No next)
      | Until(s', k) → put_in_runq
            (if ispst s' then (Free, k) else (c, next))
      | Untilv(s', e2, k) →
            put_in_runq
            (if ispst s'
            then (Free, fun () → e2 s'.value; k())
            else (c, next))
      | Control(s', k) →
          (if ispst s' then put_in_unact else put_in_runq)
            (c, next)
      )
      s.await;
      s.await ← []
```

Le traitement de la liste *present* d'un signal absent est similaire. Les processus libres passent en *runq*, les suspensibles dans leur *await_im*, les préemptibles (ou leur continuation) en *runq*.

```
let do_present_queue_absent s =
    List.iter
      (fun (c, p1, p2) → match c with
      | Free → put_in_runq (Free, p2)
      | When(s', k) → put_in_runq (c, p2)
      | Until(s', k) → put_in_runq (if ispst s' then (Free, k) else (c, p2))
      | Untilv(s', e2, k) →
          put_in_runq
            (if ispst s'
              then (Free, fun () → e2 s.value; k())
            else (c, p2))
      | Control(s', k) →
          (if ispst s' then put_in_unact else put_in_runq) (c, p2)
      )
      s.present;
    s.present ← []
```

Globalement La fonction *next_instant* se contente d'appeler les fonctions précédentes en fonction du status du signal, pour chaque signal utilisé. Pour les signaux absents, la valeur booléenne retournée indique

si le signal doit rester utilisé ou pas. On filtre pour ne garder que les signaux déjà utilisés pour le prochain instant.

NEXTINSTANT5

```
let next_instant () =
  runq :=!pauseq; pauseq := [];
  next_instant_unactive_pause2 ();
  let new_used = List.filter
      (fun s →
        let keep =
          if ispst s then (* s présent *)
            let v = s.gather s.value in
            let ov = List.hd s.value
            in
            do_await_queue_present s v ov;
            false
          else begin (* s absent *)
            do_present_queue_absent s;
            let u1 = do_await_queue_absent s in
            let u2 = do_await_im_queue_absent s
            in u1 ∨ u2
          end
        in
        if ¬ keep then s.used ← false;
        keep)
      !used_signals
  in
  incr instant;
  used_signals := new_used
```

...

5.5 Ordonnanceur

Il reste lui relativement simple. Avant d'exécuter un processus, il fixe la variable *current_context* et quand un processus se suspend il place la continuation si elle existe dans la liste appropriée.

Les processus se plaçant en attente sur un signal rendent le signal utilisé si ils sont préemptibles ou désactivables (ceci est testé par la fonction *until_or_ctrl* ci-dessous).

```
let until_or_ctrl c =
  match c with
  | Until _
  | Untilv _
  | Control _ → true
  | _ → false

let rec sched () =
  match !runq with
  | [] →
      next_instant();
      if !runq = [] then () else sched()
  | (c, p) :: t → begin runq := t; current_context := c; match p() with
    | Next → sched ()
```

```
    | Done → begin match c with
      | Free → sched ()
      | Until(s, k)
      | Untilv(s, _, k)
      | When(s, No k)
      | Control(s, k) → put_in_runq (Free, k); sched ()
    end
    | Pause p → begin match c with
      | Free → pauseq := (c, p) :: !pauseq; sched ()
      | When(s, _) → s.await_im ← (c, No(fun _ → p ())) :: s.await_im; sched ()
      | Until _
      | Untilv _
      | Control _ → pause2 := (c, p) :: !pause2; sched ()
    end
    | Wait(s, p) →
        s.await ← (c, p) :: s.await;
        if until_or_ctrl c then signal_is_used s;
        sched ()
    | Wait_im(s, p) →
        s.await_im ← (c, p) :: s.await_im;
        if until_or_ctrl c then signal_is_used s;
        sched ()
    | Present(s, p1, p2) →
        s.present ← (c, p1, p2) :: s.present;
        signal_is_used s;
        sched ()
  end
```

La fonction *spawn* utilise le contexte courant, le processus créé en "hérite" de son créateur. La fonction *start* crée des processus dans un contexte libre. Le reste de la composition parallèle ne change pas.

SPAWN5

```
let spawn = fun p → runq := (!current_context, p) :: !runq

let start pl =
  runq := List.map (fun p → Free, p) pl;
  sched ()
```

...

→ PAR, sec 1.2, page 2.

6 Constructions de contrôle imbriquées

6.1 Pile de contexte

Le contexte d'un processus sera maintenant défini par une pile d'éléments de contexte, qui sera vide pour un processus "libre". La pile sera mise en œuvre par une liste avec le sommet en tête, la contexte le plus englobant sera donc le plus profond dans la liste. Nous associons à la liste un entier indiquant le nombre d'élements de contexte de type préemption et désactivation. Ceci nous permettra de déterminer efficacement si un processus est préemptible ou désactivable sans parcours de la pile. En effet nous avons besoin de tester ce point quand un processus se place et reste en attente sur un signal : si son contexte contient un élément de contexte de type préemption ou désactivation (et que le processus est actif), le signal devra être marqué utilisé.

Pour différencier l'état du processus des actions concernant son contexte on qualifiera un processus d'actif ou inactif suivant qu'il a été désactivé ou pas. Tout processus sera à chaque instant soit actif soit inactif. Actif n'implique pas qu'il s'exécute dans l'instant car il peut de plus être suspendu. Un processus peut être inactif pour plusieurs raisons (plusieurs control/with). On dira d'un élément de contexte de désactivation (*Control*) qu'il est *valide* si il contribue à l'état inactif du processus et *invalide* dans le cas contraire. Un processus est inactif si et seulement si au moins un de ses contextes de désactivation est valide.

```
type process = unit → coop
and coop =
  | Done
  | Next
  | Pause of process
  | Wait of signal_t × awproc
  | Wait_im of signal_t × awproc
  | Present of signal_t × process × process

and awproc =
  | No of (unit → coop)
  | One of (sval_t → coop)
  | All of (sval_t list → coop)

and signal_t = {
    mutable last_emit : int;
    mutable prev_emit : int;
    mutable value : sval_t list;
    mutable prev_value : sval_t list;
    mutable used : bool;
    gather : (sval_t list → sval_t list);

    mutable await : (ctxt × awproc) list;
    mutable await_im : (ctxt × awproc) list;
    mutable present : (ctxt × process × process) list
  }

and contextelement =
  | Until of signal_t × process
  | Untilv of signal_t × (sval_t list → coop) × process
  | Control of signal_t × process
  | When of signal_t × awproc

and ctxt = contextelement list × int
```

→ DEF2, sec 2.1, page 5.

→ AWAITIM, sec 5.1, page 14. → SIGNAL, sec 4.2, page 11. → USED, sec 2.2, page 5. → AWAIT4, sec 4.2, page 11.

```
let ispst = fun s → s.last_emit = !instant
```

Changement par rapport à la version précédente, *current_context* est maintenant une référence sur une paire contenant une liste, initialement vide et un entier, initialement nul. *unactive* contiendra les processus désactivés associés à la *liste* des signaux les ayant désactivés (un processus pourra en effet être inactif pour plusieurs raisons). Deux nouvelles listes *old_unact* et *suspended* sont destinées à contenir respectivement les processus inactifs et suspendus pendant les traitements de fin d'instant.

```
let pause2 = ref [] (* pour les proc. suspensibles et désactivables *)
let unactive = ref [] (* pour les proc. désactivés *)
let old_unact = ref []
let suspended = ref []

let current_context = ref ([], 0)
```

```
let put_in_unact (sl, c, p) = unactive := (sl, c, p) :: !unactive
let put_in_susp s p = suspended := (s, p) :: !suspended
```

6.2 Entrée et sortie de contexte

Les primitives de contrôle poussent un élément de contexte au sommet de la pile, et mettent à jour le compteur avant de terminer avec la valeur *Next* qui indique à l'ordonnanceur de les replacer dans *runq* (voir 6.8 page 30).

```
let dountil s e k =
  let cel, n = !current_context in
  put_in_runq ((Until(s, k) :: cel, n + 1), e); Next

let dountilv s e e2 k =
  let cel, n = !current_context in
  put_in_runq ((Untilv(s, e2, k) :: cel, n + 1), e); Next

let controlwith s e k =
  let cel, n = !current_context in
  put_in_runq ((Control(s, k) :: cel, n + 1), e); Next

let dowhen s e k =
  let cel, n = !current_context in
  if s.last_emit = !instant then
    put_in_runq ((When(s, No k) :: cel, n), e)
  else
    put_in_await_im s ((When(s, No k) :: cel, n), No e);
  Next
```

6.3 En cours d'instant

Les éléments de contexte de type préemption et désactivation ne jouent aucun rôle au cours de l'instant. La préemption et la désactivation/réactivation se font en fin d'instant. Seuls les éléments de contexte de suspension nous intéressent donc pendant le déroulement de l'instant. L'imbrication implique de gérer potentiellement plusieurs niveaux de suspension.

Voici comment nous le gérerons. Le processus sera placé initialement dans la file *await_im* correspondant à la suspension la plus englobante (la plus profonde dans la pile). Si le signal est émis, on déplacera le processus dans la file de suspension suivante pour laquelle le signal n'est pas (ou pas encore) présent, ou dans *runq* si il n'y en a plus. Ceci est réalisé par la fonction *move_awi* qui renvoie une valeur de type *signal_t option* indiquant le signal correspondant à l'élement de contexte de suspension le plus profond pour lequel le signal n'a pas (encore) été émis. Cette fonction fait un simple parcours itératif de la liste d'éléments de contexte.

Notons qu'un signal sur lequel est processus est suspendu (et pas inactif) sera marqué utilisé si le processus est préemtible et/ou désactivable. Le marquage sera fait par *mawi* en cours d'instant et au moment de la distribution de la liste *suspended* pour la préparation de la fin d'instant.

```
let move_awi cel =
  let rec ma cel acc =
    match cel with
    | [] → acc
    | When(s, k) :: t → ma t (if ispst s then acc else (Some s))
    | h :: t → ma t acc
  in
  ma cel None
```

La fonction *mawi* utilise la précédente pour placer le processus dans la file *await_im* appropriée (ou *runq*).

```
let mawi (cel, n) p =
  let c = (cel, n) in
  match move_awi cel with
  | None → put_in_runq (c, p)
  | Some s → put_in_await_im s (c, (No p)); if n > 0 then signal_is_used s
```

Le code de l'émission est modifié conformément à ce que nous venons de voir pour la liste *await_im*.

```
let emit s v =
  if s.last_emit ≠ !instant then begin
    s.prev_emit ← s.last_emit;
    s.last_emit ← !instant;
    s.prev_value ← s.value;
    s.value ← [];
    signal_is_used s
  end;
  s.value ← v :: s.value;
  List.iter
    (fun (c, awp) → match awp with
    | No p → mawi c p
    | One p → mawi c (fun () → p v))
    s.await_im;
  s.await_im ← [];
  List.iter (fun (c, p1, p2) → put_in_runq (c, p1)) s.present;
  s.present ← []
```

6.4 Préemption des processus en fin d'instant

En fin d'instant le contexte de chaque processus va être examiné pour déterminer le traitement à lui appliquer. On peut diviser conceptuellement l'opération en deux étapes :
– Pour la fin de l'instant courant, une préemption peut être effectuée.
– Pour la préparation de l'instant suivant, le processus peut être désactivé/réactivé ou suspendu.
En premier lieu, il faut appliquer la préemption le cas échéant. Pour cela il faut rechercher l'élément de contexte de préemption le plus profond dont le signal soit présent, et si celui-ci s'applique "supprimer tout ce qui dépasse" de la pile. La préemption s'applique si la construction de préemption était exécutable au cours de l'instant. Ce n'est pas le cas si :

– un élément de contexte de suspension avec un signal absent est situé sous (englobe) la préemption,
– le processus était inactif à cause d'un élement de contexte de désactivation situé sous la préemption.

Précision sur le control/with Une préemption devrait-elle s'appliquer si le processus va être désactivé ? Une précision est nécessaire quant à la sémantique de la désactivation par control/with, comme le montre l'exemple ci-dessous, dans la situation où $< p >$ est actif et *s1* et *s2* sont tous deux présents.

```
control
    do
        < p >
    until s2 done;
    k2
with s1 done;
k1
```

En terme de notre représentation le contexte du processus est [*Until*(*s2*, *k2*); *Control*(*s1*, *k1*)]. Il est clair que le processus doit être désactivé, mais la préemption doit-elle s'appliquer ? En d'autres mots, la désactivation du processus doit elle s'opérer avant la préemption (et donc l'annuler) ou pas ? Le fait que, syntaxiquement, la désactivation englobe la préemption pourrait plaider pour une réponse positive.

La construction control with ne fait pas partie du noyau de ReactiveML dont la sémantique est décrite dans [2, p. 31] mais est définie comme un raccourci pour (*switch* étant un processus qui change le status de *active* — pour l'instant suivant — à chaque instant où s est présent) :

```
signal active, kill in
    do
        run p
    when active done;
    emit kill
    ||
    do
        run (switch s active)
    until kill done
```

```
control
    run p
with s done
```

Cette formulation montre clairement que pour un processus actif, une préemption s'applique même si le processus va être désactivé. On peut l'interpréter en disant que la désactivation s'applique *pour l'instant suivant* (le *switch* change le status de *active* en fin d'instant ce qui détermine l'application ou pas de la suspension pour l'instant suivant) alors que la préemption s'applique *pour l'instant qui vient de s'écouler*.

6.5 Préparation de l'instant suivant

Une fois la préemption effectuée le cas échéant, l'état du processus doit être déterminé pour l'instant suivant à partir de la pile de contexte. Il peut être exécutable, suspendu à un signal ou être inactif. Seuls les éléments de contexte de type *When* et *Control* jouent un rôle ici.

Les choses deviennent assez complexes alors nous allons considérer le cas des processus inactifs séparément. Il faut dire que la sémantique des constructions de contrôle imbriquées est un casse-tête remarquable !

Processus actifs Commençons donc par le cas des processus qui étaient actifs à l'instant courant. Dans la pile de contexte laissée par la préemption, un *When* sous lequel n'est pas présent un *Control* avec un signal présent implique que le processus sera suspendu. Si un *Control* avec un signal présent est plus profond, alors le processus va être désactivé. Notons que plusieurs désactivations peuvent se produire simultanément (plusieurs *Control* peuvent être validés le même instant), il faut les noter toutes.

Il nous faut donc trouver le *When* ou le *Control* avec un signal présent, le plus profond dans la pile. Si aucun des deux n'est trouvé, le processus sera directement exécutable.

La fonction *active_eoi* ci-dessous applique la préemption éventuelle et cherche le dernier (plus englobant) *Control* avec un signal présent ou *When* placé sous le *Until* qui a provoqué la préemption. Comme elle procède par un parcours itératif depuis le sommet de la pile, plusieurs préemptions peuvent être trouvées, seule la plus profonde sera au final effectuée.

Elle retourne une paire contexte-processus si la préemption a eu lieu et une valeur de type *pstat*.

```
type pstat = Unactive of signal_t list | Suspended of signal_t | Runnable
```

```
let active_eoi (cl, n) =
  let rec eoi (cl, n) (cp, state) =
    match cl with

    | Until(s, k) :: tl →
        if ispst s then eoi (tl, n − 1) (Some((tl, n − 1), k), Runnable)
        else eoi (tl, n − 1) (cp, state)

    | Untilv(s, e2, k) :: tl →
        if ispst s then eoi (tl, n − 1) (Some((tl, n − 1),fun () → e2 (s.gather s.value); k()), Runnable)
        else eoi (tl, n − 1) (cp, state)
```

```
  | When(s, k) :: tl →
      if ispst s then
        let newstate = match state with
        | Unactive _ → state
        | _ → Suspended s
        in
        (* devient suspendu sauf s'il a été désactivé *)
        eoi (tl, n) (cp, newstate)
      else
        eoi (tl, n) (cp, Suspended(s))

  | Control(s, k) :: tl →
      if ispst s then
        (* plusieurs désactivations simultanées possibles *)
        let ss = match state with
        | Unactive(ss) → ss
        | _ → []
        in eoi (tl, n − 1) (cp, Unactive(s :: ss))
      else eoi (tl, n − 1) (cp, state)

  | [] → (cp, state)
in
eoi (cl, n) (None, Runnable)
```

Processus inactifs Après éventuelle application de la préemption, les cas suivants peuvent se poser :

- le processus est réactivé car la préemption a supprimé le ou les contextes de désactivation valides qui le rendaient inactif,
- le processus reste inactif mais le contexte de désactivation le plus profond est invalidé (un seul peut le faire à chaque instant),
- le processus est réactivé car un seul *Control* était valide et le signal concerné était présent à l'instant courant,
- le processus reste inactif et un (ou plusieurs) nouveau *Control* (plus profond) devient valide,
- le processus reste inactif car aucune des conditions ci-dessus n'est valide,

Dans le cas où le processus est réactivé, il peut se trouver soit directement exécutable, soit suspendu à un signal. Comme dit plus haut, ces imbrications constituent un casse-tête assez éprouvant. Et le parcours depuis le sommet que nous avons adopté ne contribue probablement pas à rendre les choses limpides.

```
let unactive_eoi sl (cl, n) p =
    let seen_susp = ref None in
    let rec eoi (cl, n) (cp, state) =
      match cl with

    | Until(s, k) :: tl →

        la préemption réactive (sauf si elle est ensuite annulée)

        if ispst s then eoi (tl, n − 1) (Some((tl, n − 1), k), Runnable)
        else eoi (tl, n − 1) (cp, state)

    | Untilv(s, e2, k) :: tl →

        if ispst s then eoi (tl, n − 1) (Some((tl, n − 1), fun () → e2 (s.gather s.value); k ()), Runnable)
        else eoi (tl, n − 1) (cp, state)
```

```
  | When(s, k) :: tl →
      seen_susp := Some(s);
      if ispst s then
        let newstate = match state with
        | Unactive _ → state
        | _ → Suspended(s)
        in
        si a été réactivé, devient suspendu – le plus profond
        eoi (tl, n) (cp, newstate)
      else
        suspension absente empêche quoi que ce soit de se passer "au dessus"
        eoi (tl, n − 1) (None, Unactive(sl))
  | Control(s, k) :: tl →
      annule une préemption précédente, si présent réactive le processus
      if ispst s then
        let newstate = match state with
        si je suis le seul signal qui désactivait, on passe
        en Runnable sauf si une suspension se trouve au dessus
        | Unactive([fs]) →
            if fs ≡ s then
              match !seen_susp with
                None → Runnable | Some s → Suspended(s)
            else Unactive(s :: [fs])
        si je suis en tête, je m'élimine de la liste des signaux désactivants
        sinon, soit je suis ailleurs dans la liste et il y a un signal
        désactivant plus profond, je ne joue aucun rôle, soit je n'y suis pas
        et alors c'est que je suis un nouveau signal de désactivation je m'ajoute en tête
        si il y a un contexte valide dessous, j'ai tort mais il m'annulera
        | Unactive(fs :: others) → if fs ≡ s then Unactive(others) else
          if List.mem s others then state else Unactive(s :: fs :: others)
        | _ → state
        in
        eoi (tl, n − 1) (None, newstate)
      le signal est absent. Si il correspond à un contexte valide
      annule tout ce qui s'est passé au dessus, sinon ne joue aucun rôle
      else eoi (tl, n − 1) (None, if List.mem s sl then Unactive(sl) else state)
  | [] → cp, state
in
par défaut, le processus n'est pas préempté et reste inactif
eoi (cl, n) (None, Unactive(sl))
```

6.6 Placement des processus pour l'instant suivant

Les types de processus présents dans les listes en fin d'instant sont donnés tableau 1 (rappelons que *old_unact* et *suspended* ne sont utilisées que pendant les traitements de fin d'instant.

Ainsi, des processus se trouvent dans les listes *unactive*, *pauseq*, *pause2*, dans les listes *await* des signaux utilisés, plus les listes *await_im* et *present* des signaux utilisés absents.

Les destinations possibles de ces processus sont *runq*, *unactive*, les *await_im* pour les suspendus. Nous procéderons de la façon suivante :

liste	signal	processus contenus
unactive		inactifs pour l'instant courant
pauseq		libres en pause
pause2		non libres actifs en pause
await		quelconques
await_im	présent	
	absent	suspendus et quelconques
present	présent	
	absent	quelconques

TAB. 1 – Contenu des listes en fin d'instant

1. copie de *pauseq* dans *runq*.
2. copie *unactive* (qui sera une destination) dans *old_unact* (qui est une source),
3. traitement de toutes les listes des signaux utilisés. Les processus suspendus pour le prochain instant sont placés provisoirement dans la liste *suspended*, les désactivés dans *unactive*.
4. traitement de *pause2*. Les suspendus sont placés directement dans la file *await_im* correspondante. En effet ces files ont été traitées et sont donc maintenant des destinations.
5. traitement similaire de *old_unact*.
6. distribution du contenu de *suspended* dans les *await_im*.

6.7 Mise en œuvre

Nous avons maintenant tout ce qu'il nous faut pour rédiger les fonctions de traitement des listes. Commençons par les processus en attente sur des signaux.

Signaux présents Voici le cas de la liste *await* pour un signal présent. On note qu'une préemption annule la lecture de la valeur du signal attendu alors qu'une désactivation s'applique après cette lecture.

```
let do_await_queue_present s v ov =
  List.iter
    (fun (c, wp) → let cp, st = active_eoi c in
    let c, p = match cp with
    | Some(c, p) → c, p
    | None → match wp with
      | One p' → c, (fun () → p' ov)
      | All p' → c, (fun () → p' v )
    in
    match st with
    | Unactive(sl) → put_in_unact (sl, c, p)
    | Suspended(s') → put_in_susp s' (c, (No p))
    | Runnable → put_in_runq (c, p)
    )
    s.await;
  s.await ← []
```

Signaux absents Le traitement de la liste *await* dans le cas d'un signal absent apporte quelques subtilités. Le processus à placer en file pourra être :

– sa continuation s'il a été préempté,
– lui même avec réinsertion de l'instruction d'attente s'il est désactivé ou suspensible (car il est sorti de la file mais doit rester en attente du signal),

– lui même s'il est *Runnable* et doit rester en place.

Le signal est utilisé si au moins un des processus qui restent dans la liste (ou sont suspendus) est préemptible ou désactivable. Les processus inactifs ne sont pas pris en compte puisqu'ils ne joueront aucun rôle à l'instant suivant. Quand ils seront réactivés ils se remettront en attente du signal et le rendront alors utilisé.

```
let do_await_queue_absent s =
  let new_list, used = List.fold_left
      (fun (acc, used) ((cl, n), wp) →
        let c = cl, n in
        let cp, st = active_eoi c in
        let (cl, n), p, pt = match cp with
        | Some((cl, n), p) → (cl, n), p, true
        | None → match wp with
            | One p → c, (fun () → await_one s p), false
            | All p → c, (fun () → await s p), false
        in
        let nused = used ∨ n > 0 in
        match st with
        | Unactive(sl) → put_in_unact (sl, (cl, n), p); acc, used
        | Suspended(s′) → put_in_susp s′ ((cl, n), (No p)); acc, nused
        | Runnable →
            if pt then (put_in_runq ((cl, n), p); acc, used)
            else ((cl, n), wp) :: acc, nused
      )
      ([], false)
      s.await
  in
  s.await ← new_list; used
```

Les choses sont quasi-identiques pour *await_im*. Seules les instructions réinsérées et les deux cas pour le *wp* diffèrent.

```
let do_await_im_queue_absent s =
  let new_list, used = List.fold_left
      (fun (acc, used) ((cl, n), wp) →
        let c = cl, n in
        let cp, st = active_eoi c in
        let (cl, n), p, pt = match cp with
        | Some((cl, n), p) → (cl, n), p, true
        | None → match wp with
            | One p → c, (fun () → await_immediate_one s p), false
            | No p → c, (fun () → await_immediate s p), false
        in
        let nused = used ∨ n > 0 in
        match st with
        | Unactive(sl) → put_in_unact (sl, (cl, n), p); acc, used
        | Suspended(s′) → put_in_susp s′ ((cl, n), (No p)); acc, nused
        | Runnable →
            if pt then (put_in_runq ((cl, n), p); acc, used)
            else ((cl, n), wp) :: acc, nused
      )
      ([], false)
      s.await_im
  in
```

```
  s.await_im ← new_list; used
```

Le dernier cas est celui de la liste *present* qui reste assez simple. La continuation est soit celle donnée par la préemption soit celle du cas else.

```
let do_present_queue_absent s =
  List.iter
    (fun (c, p1, p2) → let cp, st = active_eoi c in
    let c, p = match cp with
    | Some(c, p) → c, p
    | None → c, p2
    in
    match st with
    | Unactive(sl) → put_in_unact (sl, c, p)
    | Suspended(s') → put_in_susp s' (c, No p)
    | Runnable → put_in_runq (c, p)
    )
    s.present;
  s.present ← []
```

Autres listes Voyons maintenant le cas des listes *pause2*, *old_unact* et *suspended*.

```
let next_instant_others () =
  List.iter
    (fun (c, p) → let cp, st = active_eoi c in
    let (c, p) = match cp with Some(c, p) → c, p | None → c, p
    in
    match st with
    | Unactive(sl) → put_in_unact (sl, c, p)
    | Suspended(s) → signal_is_used s; put_in_await_im s (c, No p)
    | _ → put_in_runq (c, p))
    !pause2;
  List.iter
    (fun (s, c, p) → let cp, st = unactive_eoi s c p in
    let (c, p) = match cp with Some(c, p) → c, p | None → c, p
    in
    match st with
    | Unactive(sl) → put_in_unact (sl, c, p)
    | Suspended(s) → signal_is_used s; put_in_await_im s (c, No p)
    | _ → put_in_runq (c, p))
    !old_unact;
  List.iter
    (fun (s, cp) → signal_is_used s; put_in_await_im s cp)
    !suspended;
  pause2 := [];
  old_unact := [];
  suspended := []
```

Globalement La fonction *next_instant* qui traite toutes les listes n'est que légèrement modifiée.

NEXTINSTANT6

```
let next_instant () =
  runq :=!pauseq; pauseq := [];
  old_unact := !unactive;
  unactive := [];
  let new_used = List.filter
      (fun s →
        let keep =
          if ispst s then (* s présent *)
            let v = s.gather s.value in
            let ov = List.hd s.value
            in
            do_await_queue_present s v ov;
            false
          else begin (* s absent *)
            do_present_queue_absent s;
            let u1 = do_await_queue_absent s in
            let u2 = do_await_im_queue_absent s
            in u1 ∨ u2
          end
        in
        if ¬ keep then s.used ← false;
        keep)
      !used_signals
  in
  used_signals := new_used;
  (* ajoute aussi des signaux dans used_signals *)
  next_instant_others ();
  incr instant
```

...

6.8 Ordonnanceur

Quand un processus termine, sa continuation si elle existe (cas de toutes les étapes de trampoline) est placée immédiatement dans *runq*, le sommet de la pile de contexte est retiré et oublié. Seule la suspension a des effets en cours d'instant, et si le processus termine c'est qu'il vient de s'exécuter donc qu'il n'est pas suspendu par l'absence d'un signal. Le compteur d'éléments de contexte de préemption et de désactivation est mis à jour.

Nous ajoutons de plus la possibilité de régler la rapidité de l'exécution. La variable *sampling* indiquera la durée souhaitée d'un instant en secondes.

```
let sampling = ref 0.

let start_timer it =
  if it > 0. then begin
    Unix.setitimer Unix.ITIMER_REAL { Unix.it_interval = it; Unix.it_value = it };
    Sys.set_signal Sys.sigalrm (Sys.Signal_handle (fun n → ()))
  end

let rec sched () =
  match !runq with
  | [] →
      if !sampling > 0. then ignore (Unix.select [] [] [] (−1.));
      next_instant ();
      start_timer !sampling;
      if !runq = [] then () else sched ()
```

```
  | ((cl, n) as c, p) :: t → begin runq := t; current_context := c; match p () with
    | Next → sched ()
    | Done → begin match cl with
      | [] → sched ()
      | Until(s, k) :: tl
      | Untilv(s, _, k) :: tl
      | Control(s, k) :: tl → put_in_runq ((tl, n − 1), k); sched ()
      | When(s, No k) :: tl → put_in_runq ((tl, n), k); sched ()
    end
    | Pause p → begin match cl with
      | [] → pauseq := (c, p) :: !pauseq; sched ()
      | _ → pause2 := (c, p) :: !pause2; sched ()
    end
    | Wait(s, p) →
        s.await ← (c, p) :: s.await;
        signal_is_used s;
        sched ()
    | Wait_im(s, p) →
        s.await_im ← (c, p) :: s.await_im;
        signal_is_used s;
        sched ()
    | Present(s, p1, p2) →
        s.present ← (c, p1, p2) :: s.present;
        signal_is_used s;
        sched ()
  end
```

6.9 Composition parallèle

Seule la fonction de démarrage *start* est modifiée pour lancer les processus initiaux dans un contexte vide, au lieu de *Free* précédemment, et prendre la valeur d'échantillonage souhaitée.

```
let spawn = fun p → runq := (!current_context, p) :: !runq

let start pl =
  let spl = 0. in
  runq := List.map (fun p → ([], 0), p) pl;
  sampling := spl;
  start_timer spl;
  sched ()
```

→ PAR, sec 1.2, page 2.

7 Ajouts possibles

La façon dont nous gérons les contextes imbriqués est loin d'être optimale. Un arbre des contextes permettrait de traiter par groupe les processus s'exécutant dans des contextes en partie communs (les processus héritent du contexte de leur créateur). Cela impliquerait cependant des changements importants et n'est peut être pas très utile pour la plupart des applications.

Il serait relativement facile d'ajouter à la bibliothèque des fonctionnalités de collecte d'information sur l'exécution. On pourrait par exemple comptabiliser à chaque instant le nombre de processus en pause, de signaux utilisés ou le nombre total d'émissions d'un certain signal etc.

Références

[1] Deleuze (Christophe). – *Programmation réactive en OCaml*. – Rapport technique, LCIS, 2009.

[2] Mandel (Louis). – *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. – Thèse de PhD, Université Paris 6, mai 2006.

[3] Filliâtre (Jean-Christophe) et Marché (Claude). – ocamlweb : a literate programming tool for objective caml. `http://www.lri.fr/~filliatr/ocamlweb/`

A Réalisation de ce document

La typographie des extraits de code source a été produite par l'outil `Ocamlweb` [3], un outil de "litterate programming" pour Ocaml. `Ocamlweb` requiert que le texte soit inclu comme commentaire du code source OCaml. Ceci ne nous convenait pas ici, d'une part parce que nous décrivons plusieurs réalisations de la bibliothèque dans le même document, d'autre part parce que nous mêlons des extraits de code Caml mais aussi ReactiveML.

Le document prend donc la forme d'un source LaTeX contenant des extraits de code OCaml apparaissant dans des environnements ou commandes bien identifiées. Un script :
– extrait les fragments de code,
– rassemble les fragments de code pour produire les sources Caml complets,
– remplace dans le document les fragments par des commandes d'inclusion de fichiers `.tex`,
– lance ocamlweb sur chaque fragment pour produire un fichier `.tex` correspondant.
L'outil `Ocamlweb` a de plus été légèrement modifié pour traiter également les sources ReactiveML.