UNIVERSITÉ PARIS 6 – PIERRE ET MARIE CURIE

THÈSE

pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PARIS 6

Spécialité : INFORMATIQUE

présentée par :

Louis MANDEL

Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive

Thèse dirigée par Marc POUZET

Soutenue le 22 mai 2006 devant le jury composé de :

Rapporteurs		
M. Frédéric BOUSSINOT	Maître de Recherche	Ecole des Mines de Paris
M. Jean-Jacques LÉVY	Directeur de Recherche	INRIA Rocquencourt
Examinateurs		
M. Gérard BERRY	Directeur Scientifique	Esterel Technologies
Mme. Florence MARANINCHI	Professeur	ENSIMAG-INPG
M. Jean-Charles POMEROL	Professeur	Université Paris 6
Directeur de thèse		
M. Marc POUZET	Professeur	Université Paris-Sud 11

Remerciements

En premier lieu, je tiens à remercier sincèrement Jean-Charles Pomerol, Président de l'université Pierre et Marie Curie, de m'avoir fait l'honneur de bien vouloir présider le jury de ma thèse.

Je remercie Frédéric Boussinot et Jean-Jacques Lévy d'avoir accepté d'être rapporteurs de ma thèse.

De par sa grande culture scientifique et sa rigueur, c'est un privilège d'avoir eu le regard critique de Jean-Jacques Lévy sur mon travail.

L'approche originale des langages synchrones de Frédéric Boussinot est à la base de mes recherches. Je souhaite le remercier pour sa gentillesse et pour les discussions très enrichissantes que nous avons eues.

Je tiens à exprimer ma profonde gratitude à Gérard Berry et Florence Maraninchi qui me font l'honneur de participer à mon jury de thèse.

Marc Pouzet m'a conduit sur ce sujet qui, après plus de trois ans de recherche, me passionne toujours autant. Sa rigueur scientifique et son goût pour le travail sont des exemples pour moi. Je tiens à lui exprimer toute ma reconnaissance pour sa direction attentive et exigeante, pour sa disponibilité et son soutien constant qui m'ont permis de mener ce travail à son terme.

Les premiers utilisateurs de ReactiveML ont beaucoup contribué à l'évolution du langage par leurs expériences ; merci à Farid Benbadis, Matthieu Carlier, Thomas Doumenc, Alain Girault, Sarah Maarek, Lionel Morel, Ludovic Samper.

Je remercie Thérèse Hardin de m'avoir accueilli dans le thème SPI du LIP6, il a été très agréable et stimulant de travailler avec cette équipe. Merci à Renaud Rioboo qui m'a formé à la démarche scientifique pendant les stages précédant ma thèse ; à Damien Doligez et Pierre Weis qui ont toujours répondu à mes questions sur OCaml lors de leurs visites hebdomadaires ; à David Massot, Thierry Lanfroy et Catherine Abiven pour leur aide dans toutes les tâches administratives ; à Pascal Cuoq et Grégoire Hamon qui m'ont fait découvrir les langages synchrones et m'ont vanté les qualités de leur directeur de thèse.

J'ai eu le plaisir de partager mon bureau au LIP6 avec des membres du thème CALFOR. Je les remercie pour leur bonne humeur pendant les moments de détente et particulièrement mes collègues de bureau : Abdolali Basiri, Magali Bardet, Gwénolé Ars, et Guénaël Renault sur qui j'ai toujours pu compter pour me tenir compagnie pendant les week-ends au labo.

Je remercie vivement Nicolas Halbwachs, Florence Maraninchi et Joseph Sifakis qui m'ont accueilli au laboratoire Verimag où je poursuis actuellement mes recherches en pré-post-doc et qui m'ont permis de terminer ma thèse dans de bonnes conditions. Merci aussi à l'ensemble des membres du laboratoire pour ce qu'ils m'apprennent au quotidien sur les systèmes embarqués et sur la montagne.

Je souhaite également remercier les membres de l'équipe MIMOSA de l'INRIA Sophia-Antipolis et de l'équipe voisine AOSTE qui m'ont reçu chaleureusement pendant deux séjours et auprès desquels j'ai beaucoup appris. Je tiens à remercier particulièrement Jean-Ferdinand Susini pour nos discussions passionnées sur le réactif.

Merci aux membres de l'UFR d'informatique de l'université Paris 7 avec qui j'ai eu l'occasion de travailler pendant mon monitorat, en particulier à Olivier Carton qui m'a encadré et dont j'ai retiré beaucoup d'enseignements lorsque je l'assistais en TD. Je tiens également à exprimer ma gratitude à Michel Lévy de m'avoir accepté dans son équipe pédagogique de logique et programmation logique à l'université Grenoble 1.

La réalisation de ce travail n'aurait pas été possible sans des moments de détente, ainsi je dois remercier Vladimir, Yann, Olivier, Lætitia(s), Vincent, Myriam, Manuel, Thérèse, Xavier, Elvire, Farid, Cletol, Ludovic, Mathias, Amélie, Hugo, Lionel, ...

Enfin je remercie naturellement ma famille pour son soutien et particulièrement mon frère Charles qui a eu le courage de relire intégralement ce document et ma mère qui s'est occupée de la préparation de la soutenance.

Table des matières

Première partie

Contexte et présentation du langage

Chapitre 1

Introduction

La programmation de systèmes réactifs tels que les jeux vidéo, les interfaces graphiques ou les problèmes de simulation est une tâche complexe. La difficulté provient d'une part des problèmes algorithmiques, d'autre part de la gestion de la concurrence entre les différents composants du système. Ainsi, les outils pour programmer ce type de systèmes doivent permettre de décrire et manipuler de structures de données complexes et proposer des mécanismes de synchronisation et de communications entre processus parallèles.

1.1 Les systèmes réactifs

En reprenant la terminologie introduite par Harel et Pnueli [53], les systèmes *transformationnels* lisent une entrée, effectuent un calcul et produisent un résultat. Des exemples typiques de systèmes transformationnels sont les compilateurs ou les logiciels de calcul scientifique. Dans ces systèmes, les communications avec l'environnement se font au début et à la fin de l'exécution. Les langages de programmations généralistes comme C [60], Java [46] ou Ocaml [63] sont adaptés à la programmation de ces systèmes.

Par opposition, les systèmes *réactifs* interagissent en continu avec l'environnement. L'environnement peut être constitué de capteurs, d'opérateurs humains ou d'autres programmes. Ces systèmes reçoivent des entrées et émettent des sorties tout au long de leur exécution. Nous distinguons parmi ces systèmes deux familles : les systèmes *temps-réel* et les systèmes *interactifs*.

Un système est dit *temps-réel* si le temps de réaction est inférieur au temps compris entre deux occurrences des événements d'entrée. Ainsi, le système est toujours prêt à répondre aux sollicitations de son environnement. Ces systèmes sont utilisés dans le cadre d'applications critiques (ferroviaire, avionique). Pour garantir qu'un système est temps-réel, il faut pouvoir vérifier statiquement qu'il est exécuté en temps et en mémoire bornés. Parmi les outils pour la programmation des systèmes temps-réel, nous nous intéresserons aux langages synchrones. Ces langages ont été introduits dans les années 1980 avec Lustre [51], Signal [49] et Esterel [12].

Dans les systèmes *interactifs*, la vitesse de réaction est déterminée par la vitesse de réaction du système. C'est par exemple le cas d'un système de fenêtrage dans un système d'exploitation. Lorsqu'une commande telle que l'ouverture d'une fenêtre est demandée par l'utilisateur, le système essaye de répondre le plus vite possible mais il n'y a pas de garantie sur le temps de réponse. Ce n'est donc pas un système temps-réel.

Les systèmes interactifs sont plus expressifs que les systèmes temps-réel car ils peuvent évoluer dynamiquement en fonction des sollicitations de l'environnement. De nouveaux processus peuvent être créés ou supprimés au cours de l'exécution du système.

Pour la programmation des systèmes interactifs, les deux modèles de programmation les plus utilisés sont les *boucles événementielles* et les *processus légers* (plus couramment appelés *threads*).

Les threads [61, 37] proposent un modèle de concurrence où les processus sont exécutés indépendamment et les communications entre les processus se font par mémoire partagée. Dans ce modèle, l'ordonnancement est préemptif et les synchronisations se font avec des primitives de synchronisation comme les *verrous* ou les *sémaphores*. Les threads sont adaptés à la programmation de systèmes où il y a peu de communications et de synchronisations entre les processus.

Comme il est expliqué dans [82], il est difficile de programmer avec des threads. L'accès à la mémoire partagée doit être protégé sinon les données peuvent devenir incohérentes. Ces mécanismes de protection des données peuvent introduire des interblocages. De plus, le non déterminisme rend le débogage difficile.

La programmation événementielle [44] (*event-driven programming*) est une autre approche pour la programmation de systèmes interactifs. Ce modèle de programmation est basé sur un ordonnancement coopératif où chaque processus contrôle le moment où il peut être interrompu.

Dans ce modèle, des actions sont attachées à des événements. Une boucle événementielle récupère les événements et déclenche successivement les actions qui leur sont associées. Chaque action doit être de courte durée afin de ne pas bloquer la boucle d'événements. C'est un des inconvénients de ce modèle.

La programmation événementielle est très utilisée pour la programmation d'interfaces graphiques. Par exemple la bibliothèque Gtk [48] utilisé pour Gimp ou Gnome[1] est basée sur une boucle d'événements. Dans cette bibliothèque, à chaque bouton de l'interface graphique est associé un événement et des actions comme fermer une fenêtre ou effacer un fichier peuvent être attachées à ces événements.

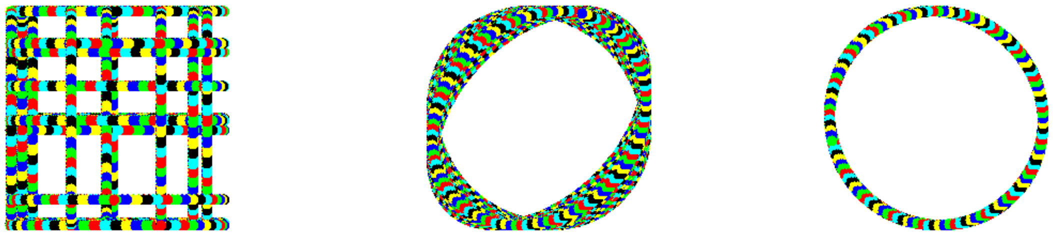
1.2 Le modèle synchrone

La programmation synchrone [50, 9] se base sur un modèle idéal où les calculs et les communications sont supposés instantanés. Dans ce modèle, le temps est défini logiquement comme une succession de réactions à des signaux d'entrée. La réaction du système à son environnement est ainsi supposée instantanée. Dans le cadre des systèmes temps-réel, cela signifie que le temps de réaction du système est suffisamment rapide pour être toujours prêt à traiter un événement venant de l'environnement. Pour les systèmes interactifs cela signifie que l'unité de temps la plus rapide est le temps de réaction du système. Chaque réaction du système définit alors un *instant*.

Plusieurs familles de langages sont fondées sur le modèle synchrone. Signal [49], Lustre [51] et Lucid Synchrone [30] sont des langages flots de données. Ils manipulent des suites infinies de valeurs comme objets de base. Il existe également des formalismes graphiques basés sur la composition d'automates. Argos [70] en est un exemple. Enfin, un langage comme Esterel [12] a un style de programmation impératif. Un comportement est défini par une succession d'actions à exécuter et plusieurs comportements peuvent être exécutés en parallèle pour construire de nouveaux comportements.

Dans ce document, nous nous intéressons à l'approche réactive introduite par Frédéric Boussinot [24]. L'approche réactive étant dérivée de Esterel, nous présentons dans la suite le modèle synchrone sous cet angle.

[1] `http://www.gimp.org`, `http://www.gnome.org`



(a) Parallèle asynchrone. (b) Parallèle asynchrone (`yield`). (c) Parallèle synchrone.

FIG. 1.1 – Comparaison des compositions parallèles synchrones et asynchrones.

1.2.1 La composition parallèle

La notion d'instant permet de définir la *composition parallèle synchrone* de processus. Si e_1 et e_2 désignent deux expressions, la réaction de la composition parallèle $e_1 || e_2$ est définie par la réaction de e_1 *et* la réaction de e_2. Ainsi, par définition, e_1 et e_2 avancent au même rythme.

Reprenons l'exemple donné dans [98] qui compare la composition asynchrone des threads avec la composition parallèle des langages synchrones. Le but du programme est de décrire un mouvement circulaire par composition parallèle de deux mouvement sinusoïdaux. Un processus calcule l'abscisse du point et un autre son ordonnée. En utilisant la syntaxe de OCAML et en notant `||` la composition parallèle, ce programme s'écrit :

```
let alpha = ref 0.0 in              let alpha = ref 0.0 in
while (true) do                     while (true) do
  x := cos(!alpha);           ||      y := sin(!alpha);
  alpha := !alpha +. 0.05;            alpha := !alpha +. 0.05;
done                                done
```

Les deux processus déclarent une variable locale `alpha` qui représente un angle. Puis ils entrent dans une boucle infinie. Le processus de gauche modifie l'abscisse du point en calculant la valeur du cosinus de `alpha` et augmente la valeur de l'angle `alpha`. De même, le processus de droite modifie l'ordonnée du point à chaque itération de la boucle.

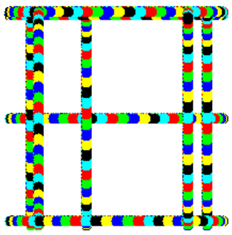
La figure 1.1(a) représente les positions successives du point lorsque l'opérateur `||` est implanté avec les threads de la machine virtuelle de OCAML. L'ordonnanceur exécute alternativement plusieurs itérations du processus qui calcule l'abscisse du point puis il exécute plusieurs itération de celui qui calcule l'ordonnée. Donc, une ligne horizontale est affichée lorsque le premier processus est exécuté et une ligne verticale lorsque l'ordonnanceur exécute le second.

Pour obtenir un cercle, il faut que l'ordonnanceur change de processus à chaque itération. Afin de donner des indications à l'ordonnanceur pour le changement de thread, la primitive `yield` est utilisée :

```
let alpha = ref 0.0 in              let alpha = ref 0.0 in
while (true) do                     while (true) do
  x := cos(!alpha);           ||      y := sin(!alpha);
  alpha := !alpha +. 0.05;            alpha := !alpha +. 0.05;
  Thread.yield();                     Thread.yield();
done                                done
```

Le résultat de l'exécution de ce programme est donné figure 1.1(b). Nous observons que le contrôle change plus régulièrement entre les deux processus. Le mouvement est *presque* circulaire. Mais ce n'est pas le résultat attendu. La primitive `yield` suggère à l'ordonnanceur de changer de thread, mais ne garantit pas qu'il va le faire.

Remarque :

> Si le même programme est exécuté en utilisant les threads du système (linux 2.6.6) à la place des threads de la machine virtuelle de OCAML, il donne le résultat suivant :
>
> 
>
> Les deux exécutions sont ici très différentes, illustrant ainsi le problème de portabilité des threads.

Maintenant si l'opérateur `||` représente la composition parallèle synchrone et si `pause` est l'instruction qui suspend l'exécution jusqu'à l'instant suivant, les positions successives du point lorsque le programme suivant est exécuté sont données figure 1.1(c).

```
let alpha = ref 0.0 in                let alpha = ref 0.0 in
while (true) do                       while (true) do
  x := cos(!alpha);           ||        y := sin(!alpha);
  alpha := !alpha +. 0.05;              alpha := !alpha +. 0.05;
  pause;                                pause;
done                                  done
```

Ici, nous obtenons un cercle car à chaque instant les deux processus sont exécutés. Par définition de la composition parallèle synchrone, il ne peut pas y avoir un processus qui avance plus vite que l'autre. Nous observons que l'ordonnancement est coopératif : chaque processus marque sa fin d'instant.

1.2.2 Les communications et les problèmes de causalité

Étudions maintenant le mode de communication entre processus utilisé dans le modèle synchrone. Les communications se font par diffusion instantanée d'événements. Les instants étant de durée nulle, un événement est soit présent, soit absent pendant un instant, mais il ne peut jamais avoir les deux statuts. De plus, les événements étant diffusés, tous les processus en ont une vision identique.

Dans le modèle synchrone, la vérification de ces propriétés demande une analyse statique qui rejette les programmes incohérents. Cette analyse s'appelle l'analyse de *causalité* [13]. Illustrons avec des programmes ESTEREL, quelques exemples de problèmes de causalité.

Le programme suivant est incohérent :

```
signal s in
  present s then nothing else emit s end;
end
```

La construction `signal/in` déclare un nouveau signal `s`. Puis l'expression `present` teste si ce signal est présent. S'il est présent, l'instruction `nothing` qui ne calcule rien est exécutée, sinon le signal `s` est généré.

Dans ce programme, il n'existe pas de statut pour `s` qui soit cohérent. Si nous supposons `s` absent, la branche `else` est exécutée et donc `s` est émis. Dans ce cas, `s` est absent et présent dans le même instant. Nous supposons alors que `s` est présent. Mais dans ce cas, la branche `then` est exécutée et `s` n'est pas émis. Par conséquent, l'hypothèse "`s` présent" est fausse. Dans cet exemple le signal `s` ne peut être ni présent, ni absent, le programme est donc incohérent.

L'exemple suivant doit être rejeté car les deux hypothèses "`s` présent" et "`s` absent" sont possibles. Le comportement est donc non déterministe :

```
signal s in
  present s then emit s else nothing end;
end
```

Un programme n'est pas non plus causal lorsqu'un signal est émis après que sa présence ait été testée :

```
signal s in
  present s then emit t else nothing end;
  emit s;
end
```

Ici, l'émission de `t` dépend de `s` qui est émis après son test de présence durant la même réaction.

Il n'y a pas seulement le test de présence qui peut introduire des problèmes de causalité. Par exemple, la construction `abort/when` est problématique. C'est une construction de préemption forte. Elle interrompt l'exécution de son corps au début de l'instant où le signal est émis (premier instant exclu). La préemption est dite *forte* par opposition à la préemption *faible* qui laisse exécuter son corps pendant l'instant de la préemption.

```
signal s in
  abort
    pause;
    emit s
  when s;
end
```

Ici, au deuxième instant, si `s` est supposé absent, le corps du `abort/when` est exécuté. Donc `s` est émis ce qui contredit l'hypothèse d'absence de `s`. Si `s` est supposé présent, le corps du `abort/when` n'est pas exécuté et donc `s` n'est pas émis. Ceci contredit l'hypothèse "`s` est présent". Donc le programme est incohérent.

La préemption faible `weak/abort/when` exécute son corps avant d'effectuer la préemption et d'exécuter le traitement de l'échappement. Dans l'exemple suivant, il y a également un problème de causalité. L'émission du signal `s2` dépend de l'émission du signal `s1` qui est effectuée après la préemption :

```
signal s1, s2, k in
  weak abort
    await s1;
    emit s2
  when k do emit s1; end weak abort;
end
```

Les signaux transportant des valeurs peuvent également introduire des boucles de causalité. Dans l'exemple suivant, le signal `s` transporte la somme des valeurs émises pendant l'instant. La valeur `1` est émise sur ce signal. La variable `x` récupère la valeur du signal. Puis, la valeur de `x` est émise sur `s`. Donc `s` émet sa propre valeur.

```
signal s := 0 : combine integer with + in
  emit s(1);
  var x := ?s: integer in
    emit s(x)
  end
end
```

Nous voyons ici apparaître une boucle de causalité car si `x` vaut `1` alors l'instruction `emit s(x)` émet `1` et donc `x` vaut `2` !

Il faut enfin remarquer que la composition parallèle de deux programmes qui sont causaux n'est pas nécessairement causale.

```
signal s1, s2 in
  present s1 then emit s2 else nothing end
  ||
  present s2 then nothing else emit s1 end
end
```

Ici, chaque `present` pris individuellement est causal, mais leur composition parallèle introduit une incohérence.

1.3 Le modèle réactif de Frédéric Boussinot

Le modèle réactif [18, 92] a été introduit par Frédéric Boussinot au début des années 1990. Le langage ReactiveC [16], une extension de C proposant un opérateur `merge` de composition parallèle, en est la première implantation. Le modèle réactif est adapté à la programmation de systèmes interactifs, plus particulièrement ceux où un grand nombre de processus sont exécutés en parallèle et où les processus communiquent et sont fortement synchronisés.

L'idée de la programmation réactive est d'utiliser un langage généraliste pour la programmation des parties algorithmiques du système et d'ajouter à ce langage un modèle de la concurrence qui soit simple, déterministe et modulaire. Ce modèle de la concurrence doit pouvoir se marier harmonieusement avec un langage généraliste.

Le principe du modèle réactif est de reprendre le modèle de concurrence des langages synchrones et de supprimer les problèmes de causalité qui sont à la fois : durs à comprendre, un obstacle à la modularité et difficiles à analyser dans le contexte d'un langage généraliste. La première formalisation de ce modèle a été faite dans SL [24].

Nous présentons maintenant deux visions de l'approche réactive. La première est un affaiblissement de l'approche synchrone afin de supprimer les problèmes de causalité. La seconde détaille l'exécution d'un instant pour donner une vision constructive du modèle.

1.3.1 Ajout de retard au modèle synchrone

Nous allons reprendre les exemples de problèmes de causalité précédents avec une interprétation réactive et voir comment l'insertion d'un retard peut résoudre ces problèmes.

La première caractéristique du modèle réactif est d'ajouter *un retard pour la réaction à l'absence d'un signal*. Donc dans la construction `present/then/else`, si le signal est absent, la branche `else` est exécutée *à l'instant suivant*.

Reprenons le premier exemple de programme incohérent. Nous ajoutons ici une apostrophe aux constructions de ESTEREL dont la sémantique change :

```
signal s in
  present' s then nothing else emit s end;
end
```

Si l'hypothèse que `s` est absent est faite, alors à l'instant suivant le programme émettra `s`. Ceci ne contredit pas l'hypothèse que `s` était absent au premier instant. Ce programme devient donc causal. Autrement dit ce programme correspond au programme ESTEREL suivant :

```
signal s in
  present s then nothing else pause; emit s end;
end
```

La seconde caractéristique du modèle est *de toujours privilégier l'hypothèse d'absence d'un signal*. Ainsi, sur l'exemple suivant où les deux hypothèses `s` présent et absent sont possibles, l'hypothèse `s` absent est prioritaire.

```
signal s in
  present' s then emit s else nothing end;
end
```

La réaction devient alors déterministe.

La combinaison de ces deux caractéristiques rend causal l'exemple suivant :

```
signal s in
  present' s then emit t else nothing end;
  emit s;
end
```

Contrairement à ESTEREL, si `s` est absent, la branche `else` est exécutée à l'instant suivant ; l'émission du signal `s` est donc décalée d'un instant. Ainsi, dans cet exemple, les deux hypothèses "`s` présent" et "`s` absent" sont possibles. C'est donc le choix `s` absent qui est fait.

Dans le modèle réactif, *la préemption forte est interdite*. De plus, *le traitement d'échappement d'une préemption faible est reporté à l'instant suivant la préemption*. Par conséquent, la construction `abort/when` n'existe pas dans l'approche réactive et le problème de causalité de l'exemple du `weak/abort/when` est supprimé car le signal `s1` n'est pas émis dans l'instant de la préemption, mais uniquement à l'instant d'après.

```
signal s1, s2, k in
  weak abort'
    await s1;
    emit s2
  when k do
    emit s1;
  end weak abort;
end
```

Cet exemple correspond au programme ESTEREL suivant où une pause est ajoutée lors du traitement de l'échappement :

```
signal s1, s2, k in
  weak abort
    await s1;
    emit s2
  when k do
    pause;
    emit s1;
  end weak abort;
end
```

La dernière caractéristique du modèle réactif est d'ajouter *un retard pour la récupération de la valeur d'un signal*. Donc dans l'exemple suivant, l'instruction `emit s(x)` est exécutée au deuxième instant lorsque la valeur de `x` est déjà fixée.

```
signal s := 0 : combine integer with + in
  emit s(1);
  var x := ?'s: integer in
    emit s(x)
  end
end
```

Ainsi, cet exemple correspond au programme Esterel suivant :

```
signal s := 0 : combine integer with + in
  emit s(1);
  var x := ?s: integer in
    pause;
    emit s(x)
  end
end
```

Enfin, dans ce modèle, comme tous les programmes sont causaux, la composition de deux programmes causaux reste un programme causal. Sur l'exemple suivant, l'hypothèse où `s1` et `s2` sont absents est valide.

```
signal s1, s2 in
  present' s1 then emit s2 else nothing end
  ||
  present' s2 then nothing else emit s1 end
end
```

En effet dans ce cas, les deux branches `else` sont exécutées à l'instant suivant le test. Donc l'émission de `s1` ne contredit pas les hypothèses de l'instant précédent.

1.3.2 Absence d'hypothèse sur les signaux

Une autre vision du modèle réactif est présentée ici. Les mêmes caractéristiques qu'avec l'ajout de retards dans le modèle synchrone vont être retrouvées.

Avec cette approche, toute hypothèse sur la présence ou l'absence des signaux est interdite et la réaction d'un instant est construite comme une succession de micro-réactions jusqu'à l'obtention d'un point fixe.

Au début d'un instant tous les signaux ont un statut qui est considéré comme indéterminé. Au cours de la réaction, lorsqu'un signal est émis, son statut est alors fixé à présent. Le test d'un signal dont le statut est indéterminé ne peut pas être résolu. Le composant parallèle effectuant le test est alors suspendu. Lorsque tous les composants du système ont terminé leur exécution ou sont suspendus sur le test de signaux dont le statut est indéterminé, alors la fin d'instant est déclarée. À partir de ce moment, il ne peut plus y avoir d'émission de signaux donc tous les signaux qui n'ont pas été émis sont absents.

Il y a alors une seconde étape dans la réaction qui s'appelle la réaction de fin d'instant. Dans cette étape le statut de tous les signaux est fixé, les tests d'absence peuvent donc être résolus et la réaction de l'instant suivant être préparée.

Nous voyons maintenant intuitivement la justifications des cinq caractéristiques du modèle réactif :

– Il ne peut pas y avoir de réaction instantanée à l'absence d'un signal car c'est seulement à la fin d'instant, que le statut d'absence peut être fixé. Sinon cela correspondrait à faire une hypothèse sur l'absence qui pourrait être contredite par l'émission du signal.
– L'hypothèse d'absence d'un signal est privilégiée par le fait que les signaux sont considérés présents seulement une fois qu'ils sont émis. De plus, le test d'un signal dont le statut est indéterminé suspend l'exécution ce qui ne peut jamais engendrer d'émission de signaux.
– La préemption forte est équivalente à faire un test d'absence d'un signal car le corps de la construction `abort/when` est exécuté instantanément uniquement si le signal de préemption est absent. Comme il ne peut pas y avoir de réaction instantanée à l'absence, la préemption forte est interdite.
– Il y a un délai avant l'exécution de la continuation d'une préemption faible car le corps peut être suspendu sur l'attente d'un signal dont le statut sera fixé seulement à la fin d'instant.
– Il faut attendre la fin d'instant pour récupérer la valeur d'un signal afin d'être sûr de ne pas manquer une valeur qui serait émise plus tard dans l'instant.

Le modèle réactif fournit toujours un ordonnancement cohérent et sans échec. En contrepartie, la suppression des problèmes de causalité entraîne une diminution de l'expressivité par rapport au modèle synchrone. Par exemple, la réaction instantanée à l'absence d'un signal devient impossible. Ainsi, il y a une dissymétrie entre le test de présence et celui de l'absence. Cette dissymétrie peut rendre les programmes difficiles à comprendre.

1.4 État de l'art

1.4.1 Les différents travaux sur l'approche réactive

Nous présentons ici les principaux travaux qui ont été réalisés autour de l'approche réactive. Nous reviendrons au chapitre 10 sur les différences avec notre proposition.

Esterel

Même si Esterel [12, 13, 11] n'est pas un langage réactif (au sens de F. Boussinot), il a sa place dans cette liste car il est à l'origine des travaux sur le réactif.

Esterel est un langage synchrone destiné à la programmation de systèmes temps-réel et de circuits. Il dispose de compilateurs destinés à la génération de logiciel [32, 89, 38] ou de circuits [45, 11].

Le modèle réactif reprend de Esterel la vision du temps en instants logiques, la composition parallèle synchrone et la communication par diffusion instantanée. De plus, la plupart des constructions utilisées dans le modèle réactif sont issues de Esterel.

ReactiveC

ReactiveC [16, 18] est la première implantation du modèle réactif faite par Frédéric Boussinot. C'est une extension du langage C avec des procédures réactives (`rproc`). Ces procédures sont des fonctions C qui ont un comportement temporel. Dans ces procédures, des instructions réactives peuvent être utilisées. Parmi ces instructions, il y a l'instruction `stop` (équivalente à `pause`) qui marque la fin d'instant et l'opérateur de composition parallèle `merge`. L'exécution d'une procédure se fait instant par instant en utilisant l'opérateur d'application `react`.

Les procédures réactives peuvent être définies récursivement. La récursion n'étant pas limitée à la récursion terminale, cela permet de créer dynamiquement des processus réactifs. Dans l'exemple suivant, la procédure `Split` se duplique à chaque instant (l'opérateur `exec` exécute une procédure réactive) :

```
rproc void Split(){
  stop;
  merge
    exec Split();
    exec Split();
}
```

ReactiveC propose des constructions de bas niveau où les communications se font par mémoire partagée. Il n'y a pas de communication par événements définie dans le langage.

Une des particularités de ce langage est de proposer des instructions qui permettent de contrôler l'ordonnancement à l'intérieur d'un instant. Ainsi, le programmeur a un contrôle très fin de l'exécution du programme.

Ce langage a été utilisé comme noyau pour l'implantation de plusieurs formalismes réactifs comme les réseaux de processus réactifs [17], SL [24] ou les Reactive Scripts [25].

SL

SL [24, 18] a formalisé l'approche réactive en proposant une variante de Esterel sans problèmes de causalité. Dans ce langage aucune hypothèse sur les signaux n'est autorisée : un signal est présent une fois qu'il a été émis, et il *était* absent à l'instant précédent si personne ne l'avait émis.

À partir de cette définition intuitive, une sémantique opérationnelle décrit la réaction d'un instant comme une succession de micro-réactions construisant l'ensemble des signaux émis pendant l'instant. Cette sémantique décrit le noyau réactif du langage sans prendre en compte les signaux valués.

Junior et les SugarCubes

Junior [57, 56] et les SugarCubes [26] sont deux bibliothèques pour la programmation réactive en Java. Junior est une proposition de Laurent Hazard, Jean-Ferdinand Susini et Frédéric Boussinot sur le noyau de primitives réactives à ajouter à Java. Les SugarCubes sont principalement développés par Jean-Ferdinand Susini et proposent une interface plus riche que celle de JUNIOR. Ainsi, JUNIOR peut être vu comme le noyau des SUGARCUBES.

Ces langages se présentant sous forme de bibliothèques JAVA, il faut écrire ainsi les parties réactives en syntaxe abstraite. Par exemple, le programme JUNIOR qui affiche les numéros d'instant s'écrit :

```
import junior.*;
import jre.*;

public class Example {
  static int cpt = 0;

  public static void main(String[] args) {
      Program P =
          Jre.Loop
            (Jre.Seq
               (Jre.Atom(new Action(){
                       public void execute(Environment env){
                           cpt++;
                           System.out.println("Instant␣"+cpt);
                       }}),
                Jre.Stop()));

      Machine M = Jre.Machine();
      M.add(P);
      for(int i=1; i<=10; i++) {
          M.react();
      }
  }
}
```

La sémantique de référence de JUNIOR est donnée par REWRITE [56] et par son implantation directe. De nombreuses autres implantations existent. La plus efficace est SIMPLE [55] mais elle n'est pas formalisée et contrairement à ce que son nom pourrait laisser croire, elle repose sur une implantation complexe. Parmi les autres implantations nous pouvons citer REPLACE, STORM [100], REFLEX [28] et l'implantation que nous avons réalisée GLOUTON [66].

Il faut noter qu'il existe également des implantations de JUNIOR dans d'autres langages : SENIOR [35] en SCHEME et JRC [83] en C.

Rejo

REJO [1] est une extension de JAVA avec la définition de méthodes réactives. Ce langage produit du JAVA utilisant la bibliothèque JUNIOR. Un des premiers objectifs de ce langage est de fournir une syntaxe simple pour la programmation JUNIOR. Ainsi, l'exemple précédent s'écrit en REJO de la façon suivante :

```
import junior.*;
import ros.kernel.*;
```

```
public class Example {
  static int cpt = 0;

  public reactive rmain(String[] args) {
      loop{
          cpt++;
          System.out.println("Instant␣"+cpt);
          stop;
      }
  }

  public static void main(String[] args) {
      Machine M = Jr.Machine();
      M.add(new Example.rmain(args));
      for(int i=1; i<=10; i++) {
          M.react();
      }
  }
}
```

Rejo permet également de programmer des agents mobiles. Un objet réactif implante le comportement d'un agent qui peut migrer à travers le réseau en utilisant la plate-forme d'exécution Ros développée par Raúl Acosta-Bermejo.

Il existe une version expérimentale de Rejo en C qui s'appelle Raul.

Les Icobjs

Les Icobjs [15, 28] définissent un langage graphique pour la programmation de systèmes réactifs. Des objets graphiques (icobjs) implantant des comportements réactifs peuvent être composés afin de construire de nouveaux icobjs.

Le but de ce langage est de proposer à des non-informaticiens un environnement graphique permettant de construire des comportements complexes à partir d'icobjs primitifs (programmés en Junior). Il existe par exemple un environnement dédié à la modélisation de systèmes en physique [95].

La première version des Icobjs était développée en ReactiveC et Tcl-Tk [81]. Les versions plus récentes se basent sur Junior.

Loft et FairThreads

Les Fair Threads [21, 97, 19] proposent une bibliothèque de threads coopératifs basés sur le modèle réactif. La spécificité de cette bibliothèque est qu'elle permet de mélanger de l'ordonnancement coopératif et préemptif. Plusieurs ordonnanceurs synchrones peuvent être exécutés en parallèle. La composition de ces ordonnanceurs est asynchrone.

Dans ce modèle, un thread peut être exécuté dans une zone synchrone avec un ordonnancement coopératif ou détaché de tout ordonnanceur synchrone et exécuté de façon asynchrone avec le reste de l'environnement. De plus, les threads peuvent changer d'ordonnanceur en cours d'exécution et passer du mode synchrone au mode asynchrone.

Des implantations de Fair Threads sont disponibles en Scheme, Java et C. En Ocaml, la bibliothèque HirondML [104] est une version des Fair Threads intégrant de la mobilité.

Loft [20] propose un langage au-dessus des Fair Threads. C'est une extension de C qui comme Rejo pour Junior, facilite le développement d'applications en Fair Threads. Par exemple, une implantation des automates cellulaires [22, 23] a été réalisée en Loft.

Reactive Programming in Standard ML

La première expérience d'intégration de la programmation réactive dans des langages fonctionnels a été faite en Standard ML [76] par Riccardo Pucella [91]. Par la suite, il y a eu d'autres implantations du modèle réactif dans cette famille de langages comme Senior [35], Fair Threads [97], ...

Dans [91], l'auteur présente une sémantique des constructions de ReactiveC dans un langage ML. La particularité de cette sémantique est de prendre en compte l'ensemble du langage. Elle présente à la fois la partie fonctionnelle et la partie réactive.

De ce travail est issu une bibliothèque pour la programmation réactive qui fait partie de la distribution de Standard ML of New Jersey [99].

ULM

ULM [14, 40] est une extension de Scheme conçu pour la programmation de systèmes distribués où les processus sont mobiles. ULM s'inspire de Fair Threads auquel il ajoute des constructions dédiées à la mobilité.

ULM suit le modèle des GALS (Globally Asynchronous Locally Synchronous) où chaque site est un système réactif déterministe et où les communications entre sites sont asynchrones.

Une attention particulière est portée au traitement des références dans la sémantique. Dans le cadre de processus mobiles, la localisation de la mémoire est une question fondamentale pour la sûreté de fonctionnement. En particulier, si deux processus exécutés en parallèle partagent de la mémoire et si un des deux processus migre, il faut pouvoir spécifier le comportement des processus lorsqu'ils vont essayer d'accéder à la mémoire.

Il existe une version allégée de ULM en C. Cette version (Lurc [64]) se présente sous forme de bibliothèque proposant une alternative à l'utilisation de la bibliothèque des PThreads [44].

SL Revisited

Des travaux récents [3, 2] définissent un noyau réactif à base de threads coopératifs. Ce noyau est proche du cœur réactif de ULM ou des Fair Threads. Ce modèle est utilisé en particulier pour formaliser l'équivalence de programmes réactifs.

1.4.2 Autres approches pour la programmation des systèmes réactifs

Nous étudions maintenant d'autres langages pour la programmation de systèmes réactifs. Nous ne cherchons pas à être exhaustifs et nous portons une attention particulière aux langages fonctionnels car notre proposition est une extension de ML.

Langages flots de données

Lucid Synchrone [30, 90] est un langage flots de données synchrone à la ML. Ce langage est une extension de Lustre avec des traits de ML comme l'inférence de type et l'ordre supérieur. Contrairement aux langages synchrones classiques, la définition récursive de processus et donc la création dynamique sont autorisées dans LUCID SYNCHRONE.

Par rapport à l'approche réactive qui part d'un langage généraliste pour lui ajouter des constructions synchrones, LUCID SYNCHRONE fait le chemin inverse : à la base c'est un langage synchrone à la LUSTRE qui est étendu avec des traits venant des langages généralistes. Ainsi, il n'a pas toute l'expressivité d'un langage généraliste. Mais en contrepartie, l'ordonnancement des processus parallèles est fait statiquement.

FRAN [39] (FUNCTIONAL REACTIVE ANIMATION) et FRP [105] (FUNCTIONAL REACTIVE PROGRAMMING) sont deux autres exemples de langages flots de données avec des aspects dynamiques. Ces langages se présentent sous forme de bibliothèques HASKELL [85]. La particularité de ces langages est de proposer un modèle continu du temps.

MGS [33] est un langage déclaratif dédié à la simulation de systèmes dynamiques en particulier dans le domaine de la biologie. Le langage repose sur la notion de collections topologiques [2] et de transformations sur ces collections. Ce langage n'est pas purement flots de données, mais les simulations se présentent souvent comme des flots de collections.

Mondes virtuels

Les mondes virtuels sont des exemples de systèmes interactifs. Dans ce domaine, des langages à base d'agents sont développés. Ces langages doivent en particulier permettre l'exécution concurrente de plusieurs agents. Il proposent ainsi une autre alternative à la programmation de systèmes interactifs.

Parmi ces langages, nous pouvons par exemple citer ORIS [54] et INVIWO [94]. La particularité de l'atelier INVIWO est de proposer le langage de programmation MARVIN qui a une syntaxe proche de ESTEREL. Dans ce langage, un agent est un programme synchrone, mais la composition entre agents est asynchrone.

Calculs de processus

Les calculs de processus comme CCS/SCCS [74] ou le π-calcul [75] ont pour but de construire un noyau minimal pour exprimer la concurrence. Mais il existe également des implantations de ces calculs. Par exemple JOCAML [43] est un langage à la ML proposant des constructions concurrentes basées sur le join-calcul [42].

Langages fonctionnels concurrents

CONCURRENTML [93] est une extension de STANDARD ML avec de la programmation concurrente. CONCURRENTML est basé sur le modèle des threads et privilégie la communication par messages. Le modèle de communication proposé par le langage sépare la description des opérations de synchronisation (émission, réception de message) de la synchronisation elle même. Cette approche permet de combiner simplement les opérations faites lors des communications.

Il existe de nombreux autres langages fonctionnels pour la concurrences comme CONCURRENT HASKELL [86] ou ERLANG [5].

[2] Une collection topologique est un ensemble d'éléments avec une relation de voisinage.

1.5 ReactiveML

1.5.1 Les traits du langage

Ce document décrit la conception, la sémantique et l'implantation de ReactiveML, un langage de programmation intégrant le modèle réactif à Ocaml.

- ReactiveML est une extension de Ocaml. Il reprend de Ocaml sa syntaxe et sa sémantique. Ce langage hôte permet de définir les structures de données et les opérations pour les manipuler. ReactiveML ajoute à ce langage une notion de temps logique discret global et des constructions permettant de décrire des comportements temporels.
- ReactiveML conserve des traits de ML comme le typage et l'ordre supérieur. Le langage est fortement typé et les types sont inférés par le compilateur. Le système de type est une extension du système de type de Ocaml de sorte que les programmes Ocaml existant gardent le même type.
- La sémantique du langage est formalisée avec une sémantique comportementale et une sémantique opérationnelle. La sémantique comportementale est la première sémantique à grands pas proposée pour le modèle réactif. La sémantique opérationnelle est à petits pas offrant ainsi un formalisme plus adapté à la preuve de certaines propriétés comme la sûreté du typage. Ces deux sémantiques ont pour particularité par rapport aux sémantiques de Junior ou SL de prendre en compte l'intégralité du langage, c'est-à-dire à la fois les parties réactives et les parties algorithmiques. Cet aspect permet de clarifier les interactions entre ces deux niveaux du langage.
- Les programmes ReactiveML sont traduits dans des programmes Ocaml séquentiels (sans threads). L'opération de composition parallèle est une composition logique. Elle ne correspond pas à une exécution parallèle. Nous n'avons pas traité l'exécution sur machines parallèles. En contrepartie, les programmes peuvent être exécutés sur des systèmes ne proposant pas de threads.
- Contrairement aux langages synchrones classiques, l'ordonnancement des calculs est fait à l'exécution et non à la compilation. Cependant l'exécution de programmes ReactiveML repose sur l'utilisation de techniques d'ordonnancement efficaces permettant d'exécuter un grand nombre de processus concurrents.

1.5.2 L'approche suivie dans ReactiveML

De nombreuses implantations du modèle réactif ont été réalisées. Parmi elles, un nombre important sont des librairies dans des langages de programmation généralistes. Cette approche semble très attirante car elle donne accès à toute l'expressivité du langage hôte et est assez légère à mettre en œuvre. Néanmoins, elle peut conduire à des confusions entre les valeurs du langage hôte et les constructions réactives. De plus avec cette approche, il est difficile d'introduire des techniques de compilation spécifiques, des optimisations et des analyses de programmes.

L'approche que nous avons choisie est de proposer la concurrence au niveau du langage. Nous enrichissons donc un langage ML strict avec de nouvelles primitives de programmation réactive. Un programme ReactiveML est un ensemble de définitions de valeurs ML et de valeurs réactives que nous appelons des *processus*. Les processus sont composés d'expressions ML et de constructions réactives. En comparaison avec l'approche sous forme de librairies, nous pensons que l'approche "langage" est plus sûre et conduit à une programmation plus naturelle.

ReactiveML est construit au dessus de Ocaml car c'est un langage généraliste qui est à la fois expressif, efficace et dont la sémantique est simple. Ce dernier point nous permet de formaliser l'ensemble de ReactiveML et ainsi de clarifier les communications entre les parties réactives et algorithmiques du langage. C'est une des originalités de nos sémantiques de prendre en compte les signaux valués. Cela permet par exemple de rendre explicite le phénomène d'échappement de portée des signaux.

Pour proposer une alternative aux threads ou à la programmation événementielle, la rapidité d'exécution est une question importante. L'efficacité de ReactiveML est due au langage hôte mais également aux stratégies d'ordonnancement des programmes réactifs. Les deux techniques d'ordonnancement que nous présentons dans ce document s'inspirent des travaux de Laurent Hazard. L'implantation est basée sur l'utilisation de files d'attentes de sorte qu'une expression est activée uniquement lorsque le signal dont elle dépend est émis.

Pour l'implantation, une attention particulière a été portée à l'intégration au langage Ocaml. D'une part, pour l'implantation des constructions réactives nous avons fait attention à ce que l'implantation soit efficace et puisse utiliser directement le glaneur de cellules (GC) de Ocaml. D'autre part pour les expression ML, nous avons réalisé une analyse statique permettant de les identifier et donc de les laisser inchangées par le compilateur ReactiveML. Ainsi, elles peuvent être exécutées sans surcoût par rapport à du code écrit directement en Ocaml.

1.6 Plan de la thèse

Le document se décompose en cinq parties. La première présente le modèle réactif et le langage ReactiveML avec un tutoriel au chapitre 2.

Dans la seconde partie, la sémantique formelle du langage est présentée. Cette partie commence au chapitre 3 par donner une sémantique comportementale à grands pas au langage. Dans cette sémantique chaque réaction correspond à l'exécution complète d'un instant et permet ainsi de s'abstraire de l'ordonnancement à l'intérieur de l'instant. Puis nous présentons au chapitre 4 une sémantique opérationnelle à petits pas. Contrairement à la sémantique comportementale, la sémantique opérationnelle décrit la construction d'un instant comme une succession de micro-réactions et permet de réaliser un premier interprète. Enfin au chapitre 5, nous présentons le système de type du langage et la preuve de sûreté du typage.

Dans la troisième partie, nous nous intéressons à l'implantation du modèle réactif et en particulier aux techniques d'ordonnancement efficaces. Le chapitre 6 présente Glouton, la première sémantique décrivant une implantation efficace du modèle réactif que nous avons réalisée. Ce travail a été fait à l'origine sur Junior en s'inspirant de Simple, l'implantation de Laurent Hazard. Dans le chapitre 7, nous présentons L_k le langage vers lequel nous compilons ReactiveML. La sémantique que nous donnons à L_k décrit la technique d'ordonnancement utilisée en ReactiveML. Nous terminons cette partie au chapitre 8 par l'étude de l'implantation en Ocaml de L_k. Dans ce chapitre, nous présentons également une bibliothèque Ocaml pour la programmation réactive.

La quatrième partie porte sur les applications écrites en ReactiveML et sur les mesures de performances. Nous détaillons au chapitre 9 l'implantation d'un simulateur de protocoles de routage dans les réseaux mobiles ad hoc. Ce simulateur a été réalisé en collaboration avec l'équipe Réseaux et Performances du Laboratoire d'Informatique de Paris 6 (LIP6).

La dernière partie est consacrée à des discussions. Dans le chapitre 10, nous étudions les choix faits dans la conception du langage par rapport aux autres implantations du modèle réactif. Le chapitre 11 présente des modifications en cours et à venir du langage. Le chapitre 12 conclut le document et présente des perspectives d'évolution de ReactiveML.

Chapitre 2

Présentation de ReactiveML

ReactiveML étend Ocaml avec des constructions de programmations concurrentes basées sur le modèle réactif. Il ajoute la notion de processus au langage hôte. Les processus sont des fonctions dont l'exécution peut prendre du temps : elles peuvent être exécutées sur plusieurs instants. Par opposition, les fonctions Ocaml sont considérées comme instantanées. En faisant l'analogie avec les circuits, les fonctions instantanées correspondent aux circuits combinatoires et les processus correspondent aux circuits séquentiels [72].

Ce chapitre présente le langage ReactiveML à travers des exemples. La présentation s'appuie sur une bonne connaissance de Ocaml.

2.1 Présentation à travers des exemples simples

ReactiveML est une extension de Ocaml ce qui signifie que tout programme Ocaml est un programme ReactiveML valide (dans l'implantation actuelle, les objets, les labels et les foncteurs ne sont pas encore intégrés). Le langage étant une extension de Ocaml, cette présentation détaille les constructions ajoutées au langage hôte.

2.1.1 Définition de processus et composition parallèle

Commençons avec comme premier exemple le processus `hello_world`. Ce processus affiche `hello` au premier instant et `world` au second. L'instruction `pause` suspend l'exécution du programme jusqu'à l'instant suivant :

```
let process hello_world =
  print_string "hello␣";
  pause;
  print_string "world"
val hello_world : unit process
```

Les lignes en *italique* sont données par le compilateur. Ici, le compilateur infère que `hello_world` est un processus dont la valeur de retour est la valeur unité (`()`).

Un programme ReactiveML définit un ensemble de valeurs et de processus. Le processus principal à exécuter est déterminé au moment de la compilation. Si le processus précédent est défini dans le fichier `hello.rml`, la ligne de compilation est alors :

```
rmlc -s hello_world hello.rml
```

Le compilateur `rmlc` génère le fichier `hello.ml` qu'il faut compiler avec un compilateur OCAML :

```
ocamlc -I ‘rmlc -where‘ unix.cma rml_interpreter.cma hello.ml
```

Il est possible d'appeler un autre processus à l'intérieur d'une définition de processus. L'application de processus se fait avec l'opérateur `run`. Par exemple le processus qui exécute deux instances de `hello_world` en parallèle s'écrit :

```
let process hello_world2 =
  run hello_world || run hello_world
val hello_world2 : unit process
```

Le symbole `||` est l'opérateur de composition parallèle synchrone. Donc l'exécution du processus `hello_world2` affiche toujours : `hello hello worldworld`. Il ne peut jamais y avoir `world` affiché avant `hello` car les deux branches parallèles partagent une échelle de temps commune et les deux `hello` sont affichés au premier instant alors que les `world` sont affichés au second. Par ailleurs, les lettres de deux mots ne peuvent jamais être intercalées car l'exécution d'une expression OCAML (comme `print_string "hello"`) est toujours atomique : il n'y a pas de concurrence au niveau des expressions OCAML. Ainsi, il n'y a pas de sections critiques à définir contrairement au modèle des threads.

Remarque :

> La composition parallèle est commutative. Donc dans l'exemple suivant, même si les deux affectations sont des opérations atomiques, la valeur de `x` n'est pas déterministe :
>
> ```
> let x = ref 0 in (x := 4) || (x := 2)
> ```

Les processus peuvent être paramétrés par un ensemble de valeurs. Le processus `pause_n` suspend l'exécution d'une branche parallèle pendant `n` instants :

```
let process pause_n n =
  for i = 1 to n do pause done
val pause_n : int -> unit process
```

Lorsqu'un processus termine son exécution, il retourne une valeur. Ainsi, le processus `fact` faisant une multiplication par instant s'écrit :

```
let rec process fact n =
  pause;
  if n <= 1 then
    1
  else
    let v = run (fact (n-1)) in
    n * v
val fact : int -> int process
```

De même, le processus `fibo` s'écrit :

```
let rec process fibo n =
  pause;
  if n < 2 then 1
  else
    let f1 = run (fibo (n-1)) and f2 = run (fibo (n-2)) in
    f1 + f2
val fibo : int -> int process
```

`let/and/in` est une construction de mise en parallèle. Ici, chaque instance de `fibo` exécute deux instances du processus en parallèle jusqu'à ce que `n` soit inférieur à `2`. Ainsi, si `n` est supérieur à `1`, le calcul de `fibo n` prend `n + 1` instants.

Si la construction `let/and/in` est remplacée par deux `let/in`, les calculs de `f1` et `f2` ne sont pas exécutés en parallèle mais en séquence. Dans ce cas, le nombre d'instants nécessaires pour le calcul de `fibo n` avec `n` supérieur à `1` est la somme des nombres d'instants nécessaires pour le calcul de `f1` et `f2` plus un.

Remarque :

> Les constructions `let/in` et `let/and/in` sont fondamentalement différentes. Le `let/in` est une construction de mise en séquence alors que `let/and/in` est une construction de mise en parallèle.
>
> $$e_1\texttt{;}e_2 \equiv \texttt{let \_ = } e_1 \texttt{ in let x = } e_2 \texttt{ in x}$$
> $$e_1\texttt{||}e_2 \equiv \texttt{let \_ = } e_1 \texttt{ and \_ = } e_2 \texttt{ in ()}$$

2.1.2 Les communications

Les communications entre processus parallèles se font par diffusion instantanée d'événements. Un signal peut être émis (`emit`), attendu (`await`) et on peut tester sa présence (`present`). Commençons avec le processus `is_present` qui affiche `Present` si le signal `x` est présent (les paramètres du type `event` seront expliqués plus tard) :

```
let process is_present x =
  present x then print_string "Present"
val is_present : ('a, 'b) event -> unit process
```

La notation `present` *<signal>* `then` *<expr>* est un raccourci pour `present` *<signal>* `then` *<expr>* `else ()`.

Le programme dual a une sémantique un peu différente :

```
let process is_absent x =
  present x else print_string "Absent"
val is_absent : ('a, 'b) event -> unit process
```

Si le signal `x` est présent, rien n'est affiché. S'il est absent, l'affichage de `Absent` est retardé d'un instant car ReactiveML suit les restriction du modèle réactif [18].

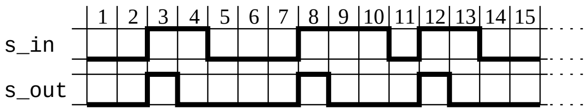
Écrivons maintenant un processus qui émet le signal `z` à chaque fois que `x` et `y` sont synchrones (présents au même instant) :

```
let process together x y z =
  loop
    present x then present y then (emit z; pause)
  end
val together :
  ('a, 'b) event -> ('c, 'd) event -> (unit, 'e) event -> unit process
```

Ici, si `x` ou `y` sont absents, la branche `else` implicite du `present` est empruntée. Ceci est équivalent à exécuter une instruction `pause`. Ainsi nous pouvons remarquer que quelque soit le statut de `x` et `y`, l'exécution du corps de la boucle n'est pas instantané. Cela garantie la réactivité du programme. En effet, s'il y a une boucle instantanée dans un programme, l'exécution d'un instant ne termine jamais, et donc les instants peuvent pas passer : le temps "s'arrête".

Voyons enfin l'exemple du détecteur de front haut. Le comportement du processus `edge` est d'émettre le signal `s_out` quand `s_in` est présent et qu'il était absent à l'instant précédant :

```
let process edge s_in s_out =
  loop
    present s_in then pause
    else (await immediate s_in;
          emit s_out)
  end
val edge : ('a, 'b) event -> (unit, 'c) event -> unit process
```



Tant que `s_in` est présent, `s_out` n'est pas émis. Quand `s_in` devient absent, `s_out` n'est toujours pas émis, mais le contrôle passe par la branche `else`. À l'instant suivant, le processus se met en attente de l'émission de `s_in`. Maintenant, quand `s_in` est présent, `s_out` est émis (`s_in` était nécessairement absent à l'instant précédant). Le mot clé `immediate` indique que `s_out` est émis à l'instant où `s_in` est présent.

Remarque :

> La construction `await` exécute sa continuation à l'instant suivant l'émission du signal alors que la version immédiate de cette construction qui exécute sa continuation à l'instant où le signal est émis.
>
> Il faut faire attention : la construction `await/immediate` est la même que celle d'ESTEREL, mais la version non immédiate est différente. En ESTEREL, nous avons :
>
> `await s` $\equiv$ `pause; await immediate s`
>
> alors qu'en REACTIVEML, la pause est après le `await` :
>
> `await s` $\equiv$ `await immediate s; pause`
>
> Nous discuterons de ce choix chapitre 10.3.5.

2.1.3 Les structures de contrôle

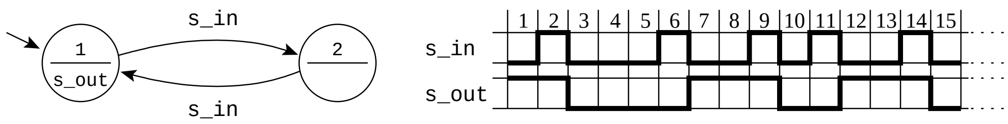
On introduit maintenant les deux principales structures de contrôle du langage : la construction `do` e `when` s suspend l'exécution de e lorsque s est absent et `do` e `until` s `done` interrompt définitivement l'exécution de e quand s est présent. On illustre ces deux constructions avec un processus `suspend_resume` qui contrôle les instants où un processus est exécuté. Pour reprendre l'analogie donnée dans [10], la construction `do/until` correspond à la commande UNIX `Ctrl-c` et le processus `suspend_resume` a le comportement du couple de commande `Ctrl-z` et `fg`.

On commence par définir le processus `sustain` paramétré par un signal `s`. `sustain` émet `s` à chaque instant :

```
let process sustain s = loop emit s; pause end
val sustain : (unit, 'a) event -> unit process
```

Nous définissons un autre opérateur typique : `switch`. Ce processus est un automate de Moore [77] à deux états. Il est paramétré par deux signaux, `s_in` et `s_out`. Son comportement est de maintenir l'émission de `s_out` tant que `s_in` est absent. Puis, lorsque `s_in` est présent, le processus suspend l'émission de `s_out` jusqu'à ce que `s_in` soit émis à nouveau. Le processus retourne alors dans son état initial.

```
let process switch s_in s_out =
  loop
    do run (sustain s_out) until s_in done;
    await s_in
  end
val switch : ('a, 'b) event -> (unit, 'c) event -> unit process
```



On définit maintenant le processus d'ordre supérieur `suspend_resume`. Il est paramétré par un signal `s` et un processus `p`. Ce processus commence l'exécution de `p`. Puis, à chaque émission de `s`, il suspend puis reprend l'exécution de `p` alternativement. On implante ce processus par la composition parallèle (1) d'un `do/when` qui exécute le processus `p` seulement quand le signal `active` est présent et (2) l'exécution d'un processus `switch` qui contrôle l'émission de `active` avec le signal `s`.

```
let process suspend_resume s p =
  signal active in
  do run p when active
  ||
  run (switch s active)
val suspend_resume : ('a, 'b) event -> 'c process -> unit process
```

On peut constater que même si `p` termine le processus `suspend_resume` ne termine pas car le processus `switch` ne termine jamais. De plus, on ne peut pas récupérer la valeur calculée par `p`. Afin de corriger ces deux points, on peut réécrire le processus `suspend_resume` de la façon suivante :

```
let process suspend_resume s p =
  signal active, kill in
  let v =
    let x = do run p when active in
    emit kill;
    x
  and _ = do run (switch s active) until kill done in
  v
val suspend_resume : ('a, 'b) event -> 'c process -> 'c process
```

La construction `let/and/in` permet de récupérer des valeurs calculées en parallèle.

Ce processus étant très utile, la construction `control/with` qui a le même comportement a été ajoutée au langage. Le processus `suspend_resume` peut donc s'écrire simplement :

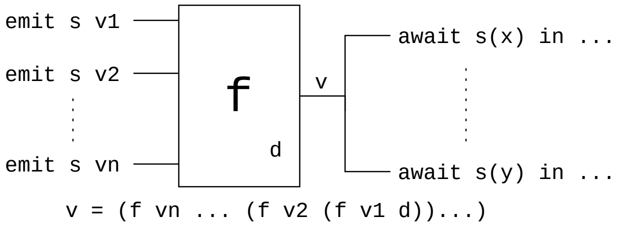


FIG. 2.1 – Multi-émission sur le signal `s`, combinée avec la fonction `f`, donne la valeur `v` à l'instant suivant.

```
let process suspend_resume s p =
  control
   run p
  with s
val suspend_resume : ('a, 'b) event -> 'c process -> 'c process
```

2.1.4 Signaux valués et multi-émission

Nous nous intéressons maintenant aux signaux valués. Ils peuvent être émis (`emit` *<signal>* *<value>*) et la valeur qui leur est associée peut être récupérée (`await` *<signal>* (*<pattern>*) `in` *<expression>*). La construction `await/in` (comme un `let/in`) introduit un nouvel identificateur dans sa partie droite. Ici, ce nom est associé à la valeur du signal.

Plusieurs valeurs peuvent être émises pendant un instant sur un signal, cela s'appelle de la multi-émission. REACTIVEML adopte une solution originale pour traiter la multi-émission. À la déclaration du signal, le programmeur doit définir comment combiner les valeurs émises pendant un même instant. Pour cela, il y a la construction suivante :

`signal` *<name>* `default` *<value>* `gather` *<function>* `in` *<expression>*

Le comportement en cas de multi-émission est illustré figure 2.1. On suppose que le signal `s` est déclaré avec la valeur par défaut `d` et la fonction de combinaison `f`. Si les valeurs `v1`, ..., `vn` sont émises pendant un instant, tous les `await` reçoivent la valeur `v` à l'instant suivant[1]. Récupérer la valeur d'un signal est retardé d'un instant pour éviter les problèmes de causalité. En effet, contrairement à ESTEREL, le programme suivant est causal :

```
await s(x) in emit s(x+1)
```

La valeur `x` du signal `s` est disponible seulement à la fin d'instant. Donc, si `x = 41` pendant l'instant courant, le programme émet `s(42)` à l'instant suivant.

Le type des valeurs émises sur un signal et le type du résultat de la fonction de combinaison peuvent être différents. Cette information est contenue dans le type du signal. Si `t1` est le type de la valeur émise sur le signal `s` et `t2` celui de la combinaison, alors `s` a le type `(t1,t2) event`.

Si on veut définir le signal `sum` qui calcule la somme de toutes les valeurs émises, alors le signal doit être déclaré de la façon suivante :

```
signal sum default 0 gather (+) in ...
sum : (int,int) event
```

[1] `v = (f vn ... (f v2 (f v1 d))...)`

Dans ce cas, le programme `await sum(x) in print_int x` attend le premier instant où `x` est présent. Puis, à l'instant suivant, il affiche la somme des valeurs émises sur `x`.

Un signal peut collecter toutes les valeurs émises pendant un instant. Pour cela, il y a la syntaxe spéciale :

```
signal s in ...
```

qui est un raccourci pour :

```
signal s default [] gather fun x y -> x :: y in ...
s : ('a, 'a list) event
```

Ici, la valeur par défaut est la liste vide et la fonction de combinaison ajoute une valeur émise à la liste des valeurs déjà émises.

Donnons un exemple qui utilise les signaux valués. On définit le processus `run_n` qui exécute un processus `p` pendant au plus `n` instants. Le processus affiche `Execution aborted` si l'exécution de `p` prend plus de `n` instants et `Execution finished` si `p` termine normalement. Ce processus utilise la construction `do/until/->` où la partie droite de la flèche est le traitement à exécuter en cas préemption.

```
let process run_n n p =
  signal finished default false gather (or) in
  do
    run p;
    emit finished true;
    pause;
    ||
    run (pause_n (n-1));
    emit finished false;
    pause;
  until finished(b) ->
    if b then
      print_string "Execution␣finished"
    else
      print_string "Execution␣aborted"
  done
val run_n : int -> 'a process -> unit process
```

Ici, `p` est exécuté en parallèle avec un processus qui attend `n` instants. Le premier processus qui termine son exécution émet le signal `finished` pour tuer l'autre processus. La valeur du signal `finished` est alors utilisée pour savoir si l'exécution de `p` a été interrompue ou pas. Si l'exécution de `p` prend exactement `n` instants, les deux branches parallèles émettent `finished` dans le même instant avec les valeurs `true` et `false`. Dans ce cas, la fonction de combinaison associe `true` au signal pour indiquer que `p` s'est terminé normalement.

Il existe deux autres constructions pour récupérer la valeur associée à un signal : `pre ?s` et `await/one/in`. L'expression `pre ?s` récupère instantanément la valeur précédente d'un signal. Si le signal n'a jamais été émis, cette expression s'évalue en la valeur par défaut du signal.

L'expression `await one s(x) in ...` ne récupère pas la valeur combinée du signal, mais une des valeurs émises. Comme pour la construction `await/in`, le corps du `await/one/in` est exécuté à l'instant suivant l'émission. Mais il existe également une version immédiate de cette construction. Ainsi l'expression `await immediate one s(x) in ...` exécute son corps à l'instant ou `s` est émis.

Il faut noter que cette construction n'est pas déterministe. Si plusieurs valeurs sont émises sur `s` pendant un instant, la valeur récupérée par `await/one/in` n'est pas spécifiée.

Remarque :

> Dans l'implantation actuelle, `await/one/in` peut être utilisé uniquement si le signal est déclaré sans fonction de combinaison.

2.1.5 Aspects dynamiques, ordre supérieur et échappement de portée

Parmi les aspects dynamiques, il y a la reconfiguration de systèmes réactifs. Une forme simple de reconfiguration est la réinitialisation de processus. Un opérateur de réinitialisation peut être implanté à partir de la construction `do/until`. Le processus suivant réinitialise le processus `p` à tous les instants où le signal `s` est présent :

```
let rec process reset s p =
  do
    run p
  until s(_) -> run (reset s p) done
val reset : ('a, 'b) event -> 'c process -> 'c process
```

Ce processus est récursif. Il appelle une nouvelle instance de `reset` à chaque fois que `s` est présent.

On définit maintenant le processus `replace` qui permet de remplacer un processus `p` par un processus `p'` à chaque fois que `s` est émis. Les processus étant des valeurs (comme les fonctions), la définition du processus qui doit remplacer `p` peut être envoyé sur le signal `s`.

```
let rec process replace s p =
  do
    run p
  until s(p') -> run (replace s p') done
val replace : ('a, 'b process) event -> 'b process -> 'b process
```

Ce processus a la même structure que `reset`. La différence est que la fonction de traitement d'échappement récupère la définition du nouveau processus à exécuter.

Nous encodons maintenant la construction `Jr.Dynapar("add", Jr.Halt())` de Junior. Cette construction a été introduite dans [1] pour la programmation de systèmes d'agents. Ce processus reçoit sur le signal `add` des processus qu'il exécute en parallèle.

```
let rec process dynapar add =
  await add (p) in
  run p
  ||
  run (dynapar add)
val dynapar : ('a, process) event -> unit process
```

Nous définissons le processus `send` qui émet le processus `p1` sur le signal `add` et attend un acquittement de son exécution pour exécuter le processus `p2`. Le processus `p2` utilise le résultat rendu par l'évaluation de `p1`.

```
let process send add p1 p2 =
  signal ack in
  emit add (process (let v = run p1 in emit ack v));
  await immediate one ack(res) in
  run (p2 res)
val send :
  (unit process, 'a) event -> 'b process -> ('b -> 'c process) -> 'c process
```

L'expression `process (let v = run p1 in emit ack v)` est la définition d'un processus anonyme qui exécute `p1` et émet le résultat rendu par `p1` sur le signal `ack`. Dans ce processus, le signal `ack` est libre lorsqu'il est émis sur `add`. Comme la portée de `add` est supérieure à celle de `ack`, ce dernier échappe de sa portée. C'est un phénomène classique du π-calcul [75].

Grâce à l'échappement de portée, un canal de communication privé a été créé entre `send` et le processus émis sur `add`. La liaison statique des noms garantit qu'il ne peut pas y avoir d'autres processus qui émettent ou qui écoutent sur `ack`.

La mobilité du π-calcul peut ainsi être simulée. Reprenons l'exemple du chapitre 9.3 de [75]. Trois processus `p`, `q` et `r` sont exécutés en parallèle. D'une part `p` et `q` peuvent communiquer en utilisant un signal `x` et d'autre part `p` et `r` communiquent par le signal `z`. Les processus `p` et `q` peuvent être définis de telle sorte que `q` et `r` puissent communiquer en utilisant `z`.

```
let process p x z =
  emit x z;
  run (p' x)
val p : ('a, 'b) event -> 'a -> unit process

let process q x =
  await one x(y) in
  run (q' y)
val q : ('a, 'a list) event -> unit process

let process r z = ...
val r : ('a, 'b) event -> unit process

let process mobility x z =
  run (p x z) || run (q x) || run (r z)
val mobility :
  (('a, 'b) event, ('a, 'b) event list) event ->
  ('a, 'b) event -> unit process
```

2.2 Deux exemples complets

2.2.1 Le banc de poissons

Nous décrivons maintenant l'implantation d'un problème de simulation. Nous voulons simuler des poissons qui nagent dans un aquarium.

Nous commençons par définir le type `position` comme un couple d'entiers et le type `frame` comme une paire de positions.

```
type position = int * int
type frame = position * position
```

Puis, nous définissons d'abord un processus `button` (à cliquer avec une souris). Il est paramétré par ses coordonnées (`frame`) et deux signaux (`mevent` et `oclick`). On suppose l'existence d'une fonction `draw` qui dessine un bouton et une fonction booléenne `inside (x,y) frame` qui teste quand le point de coordonnées `(x,y)` est à l'intérieur des coordonnées de `frame`. Le signal d'entrée `mevent` représente la souris. Ce signal est présent quand le bouton de la souris est appuyé et la valeur associée au signal est la position de souris. Le processus `button` émet le signal de sortie `oclick` si la souris clique à l'intérieur du bouton.

```
let process button frame mevent oclick =
  draw frame;
  loop
    await mevent (pos) in
    if inside pos frame then emit oclick;
  end
val button :
  frame -> ('a, position) event -> (unit, 'b) event -> unit process
```

Un poisson est un processus ayant cinq paramètres. `border`, `init_pos` et `size` définissent les limites de l'aquarium, la position initiale du poisson et sa taille. Avec le signal `others` un poisson reçoit en continu la position de tous les poissons présents dans l'aquarium et il reçoit la position de la souris avec `mevent`.

```
let process fish border init_pos size others mevent =
  (* start from an initial position *)
  signal kill in
  let pos = ref init_pos in
  do
    loop
      (* moves according to some law *)
      emit others (!pos);
      await others (o) in
      clear_fish !pos size;
      pos := law !pos border o;
      draw_fish !pos size;
    end
    ||
    loop
      await mevent (mouse_pos) in
      if inside_fish mouse_pos !pos size then emit kill;
    end
  until kill done;
  clear_fish !pos size
val fish :
  frame -> position -> int ->
  (position, position list) event -> ('a, position) event -> unit process
```

Le comportement d'un poisson est le suivant : il définit d'abord un signal local `kill` permettant de supprimer le poisson et une variable impérative `pos` représentant sa position puis il effectue deux actions en parallèle de manière cyclique. La première consiste à (1) émettre sa position, (2) récupérer la position des autres poissons, (3) calculer sa nouvelle position selon la loi `law`. L'autre comportement cyclique surveille le signal `mevent` et si la souris clique sur le poisson, alors le signal `kill` est émis. Ces deux actions sont incluses dans une construction de préemption `do/until` de sorte que le processus `fish` termine quand `kill` est présent.

Nous définissons maintenant un banc de poissons. Ce banc (`shoal`) est agrandi avec un nouveau poisson chaque fois que le signal `add` est émis.

```
let rec process shoal frame add others mevent =
  await add;
  (* the initial position is taken randomly *)
  let init_pos = random_pos frame in
  run (fish frame init_pos ((Random.int 10)+1) others mevent)
  ||
  run (shoal frame add others mevent)
val shoal :
  frame -> ('a, 'b) event ->
  (position, position list) event -> ('c, position) event -> unit process
```

On peut remarquer que `shoal` est un processus récursif. Lorsque `add` est émis, le processus crée un poisson et se rappelle récursivement pour attendre de nouveau le signal `add`. Il n'y a pas de récursion instantanée car `await add` exécute sa continuation un instant après l'émission du signal.

Maintenant, un aquarium est un processus qui contient un banc de `n` poissons et un bouton pour en ajouter. Tous les poissons utilisent le signal `others` pour communiquer leur position. Ce signal doit donc collecter toutes les positions émises.

```
let process aquarium ((x_min,y_min),(x_max,y_max)) mevent n =
  let x_sep = x_min + (x_max-x_min) / 10 in
  (* compute the border for the water *)
  let shoal_frame = (x_sep,y_min),(x_max,y_max) in
  (* compute the border for the create button *)
  let but_frame = (x_min,y_min),(x_sep,y_max) in
  signal others default [] gather fun x y -> x :: y in
  signal add in
  run (button but_frame mevent add)
  ||
  run (shoal shoal_frame add others mevent)
  ||
  for i = 1 to n dopar
    run (fish shoal_frame (random_pos shoal_frame) (random_size())
            others mevent)
  done
val aquarium : frame -> ('a, position) event -> int -> unit process
```

Ce processus fait donc la mise en parallèle d'un processus `button`, d'un processus `shoal` et de `n` processus `fish`.

Finalement, le programme principal est un processus sans paramètre. Ce processus définit d'abord quelques constantes puis appelle le processus `mouse` (qui définit les signaux de souris) et le processus `aquarium` en parallèle.

```
let process main =
  Graphics.open_graph "";
  let border = ((0, 0), (Graphics.size_x(), Graphics.size_y())) in
  signal mevent default (0,0) gather fun x y -> x in
  run (mouse mevent)
  ||
  run (aquarium border mevent 10)
val main : unit process
```

2.2.2 Le crible d'Ératosthène

Terminons avec l'exemple du crible d'Ératosthène tel qu'on peut le trouver dans [59]. C'est un exemple classique de l'approche réactive (cf. [20] par exemple). Il reprend l'utilisation des signaux, de la composition parallèle et de la création dynamique.

On définit d'abord le processus `integers` qui génère la suite des entiers naturels à partir de la valeur de `n`.

```
let rec process integers n s_out =
  emit s_out n;
  pause;
  run (integers (n+1) s_out)
val integers : int -> (int, 'a) event -> 'b process
```

Un entier est émis à chaque instant sur le signal `s_out`. Il n'y a pas de récursion instantanée grâce à l'instruction `pause`.

Nous définissons maintenant le processus `filter` qui supprime les multiples d'un nombre premier.

```
let process filter prime s_in s_out =
  loop
    await one s_in(n) in
    if n mod prime <> 0 then emit s_out n
  end
val filter : int -> (int, int list) event -> (int, 'a) event -> unit process
```

Maintenant, le processus `shift` crée un nouveau processus `filter` à chaque fois qu'un nouveau nombre premier est découvert.

```
let rec process shift s_in s_out =
  await one s_in(prime) in
  emit s_out prime;
  signal s default 0 gather fun x y -> x in
  run (filter prime s_in s) || run (shift s s_out)
val shift : (int, int list) event -> (int, 'a) event -> unit process
```

Enfin, nous définissons le processus `output` qui affiche les nombres premiers et le processus principal `sieve`.

```
let process output s_in =
  loop
    await one s_in (prime) in
    print_int prime
  end
val output : (int, int list) event -> unit process

let process sieve =
  signal nat, prime in
  run (integers 2 nat)
  ||
  run (shift nat prime)
  ||
  run (output prime)
val sieve : unit process
```

Les fonctions de combinaison de `nat` et `prime` ne gardent qu'une seule des valeurs émises pendant l'instant.

Résumé

Nous reprenons ici l'ensemble des constructions qui ont été ajoutées à OCAML.

Définition de processus

```
let process <id> = <expr> in <expr>
let process <id> <pattern> ... <pattern> = <expr> in <expr>
process <expr>
```

Les définitions de processus sont introduites par le mot clé `process`. Un processus peut être nommé avec la construction `let process` ou on peut définir des processus anonymes avec la notation `process` *<expr>*.

Instructions de base

```
nothing
pause
halt
run <process>
```

`nothing` est équivalent à `()`. `pause` suspend l'exécution pour un instant, `halt` suspend l'exécution pour toujours et `run` exécute un processus.

Compositions

```
<expr> ; <expr>
<expr> || <expr>
let <pattern> = <expr> and <pattern> = <expr> in <expr>
```

En REACTIVEML, on peut exécuter deux expressions en séquence ou en parallèle. La construction `let/and/in` calcule deux expressions en parallèle et récupère leurs valeurs pour ensuite calculer la troisième expression.

Déclarations de signaux

```
signal <id> in <expr>
signal <id>, ..., <id> in <expr>
signal <id> default <value> gather <function> in <expr>
```

Ces constructions servent à déclarer de nouveaux signaux. Lors de la déclaration, on peut définir comment combiner les valeurs émises pendant un instant sur un signal avec la construction `signal/gather`. Si on ne donne pas de fonction de combinaisons, le comportement du signal est de collecter toutes les valeurs émises.

Émission de signaux

```
emit <signal>
emit <signal> <value>
```

Les émissions se font par diffusion instantanée. Ainsi, un signal est présent ou absent pendant un instant mais il ne peut pas avoir les deux statuts. La notation `emit <signal>` est un raccourci pour émettre un signal avec la valeur `()` (unit).

Statut d'un signal

```
present <signal> then <expr> else <expr>
await <signal>
await immediate <signal>
pre <signal>
```

L'expression `present` teste le statut du signal. Si le signal est présent, la branche `then` est exécutée instantanément, sinon la branche `else` est exécutée à l'instant suivant.

L'expression `await s` attend que `s` soit émis et termine à l'instant suivant. Alors que l'expression `await immediate s` attend que `s` soit émis et termine instantanément.

Comme ESTEREL, la version non immédiate a été choisi comme version par défaut pour que la séquence

```
await s; await s
```

attende deux occurrences de `s`. Alors que `await immediate s; await immediate s` est équivalent à `await immediate s`.

L'expression `pre s` s'évalue en `true` si le signal `s` a été émis à l'instant précédent ou en `false` sinon.

Valeur d'un signal

```
await one <signal> (<pattern>) in <expr>
await immediate one <signal> (<pattern>) in <expr>
await <signal> (<pattern>) in <expr>
pre ?<signal>
```

La construction `await/one/in` attend l'émission d'un signal pour lier le pattern avec une des valeurs émises. En cas d'émission multiple sur un signal dans un instant, le choix de la valeur n'est pas spécifié. Le pattern peut être alors utilisé dans la partie droite du `await one/in`. Comme pour `await`, la partie droite est exécutée un instant après la réception du signal, sauf si on a ajouté le mot clé `immediate`.

Le `await/in` attend l'émission d'un signal et récupère à la fin de l'instant la combinaison des valeurs émises pendant l'instant sur le signal. Le corps du `await/in` est exécuté à l'instant suivant l'émission du signal. Pour des raisons de causalité, il n'y a pas de version immédiate de cette construction.

L'expression `pre ?s` est la dernière valeur associée à `s`. Tant que `s` n'a pas été émis, `pre ?s` est égal à la valeur par défaut donnée à la déclaration du signal.

Itérateurs

```
loop <expr> end
while <expr> do <expr> done
for <id> = <expr> [ to | downto ] <expr> do <expr> done
for <id> = <expr> [ to | downto ] <expr> dopar <expr> done
```

`loop` est la boucle infinie. `while` et `for/do` sont les boucles classiques. Ces constructions exécutent plusieurs instances de leur corps en séquence. Au contraire, la construction `for/dopar` exécute les instances de son corps en parallèle.

Structures de contrôle

```
do <expr> when <signal>
control <expr> with <signal>
do <expr> until <signal> done
do <expr> until <signal>(<pattern>) -> <expr> done
```

`do/when` et `control/with` sont des constructions de suspension. Le `do/when` exécute son corps seulement quand le signal est présent. `control/with` alterne entre le mode actif et suspendu chaque fois que le signal est émis.

La construction de préemption `do/until` arrête l'exécution de son corps à la fin d'instant quand le signal est émis. La seconde forme de `do/until` exécute un traitement d'échappement lorsqu'il y a préemption.

Deuxième partie

La sémantique de ReactiveML

Chapitre 3

Sémantique comportementale

Dans cette partie, nous présentons le calcul réactif sur lequel est fondé ReactiveML. Nous nous intéressons à la fois à ses sémantiques statiques et dynamiques. Nous partons d'un noyau déterministe du langage que l'on augmente avec les références et la construction `await/one/in`. Dans ce chapitre, nous présentons la syntaxe abstraite du langage, l'analyse permettant d'identifier les expressions instantanées et une sémantique dite *comportementale à grand pas*. Dans le chapitre 4 nous verrons une sémantique *opérationnelle à petit pas* et l'équivalence entre les deux sémantiques dynamiques. Le chapitre 5 présentera le système de type du langage.

Une version préliminaire de ce chapitre est publié dans [68].

3.1 Le noyau du langage

Nous considérons le noyau du langage. Il étend un noyau fonctionnel de type ML avec des constructions de programmation réactive.

3.1.1 Définitions

La syntaxe des expressions (e) est définie figure 3.1. Nous pouvons remarquer que la construction `let/and/in` n'est pas récursive ; pour les définitions récursives, il y a la construction `rec`. La construction `let/and/in` est utilisée pour la composition parallèle, elle ne peut pas être récursive car nous ne savons pas garantir que le résultat de l'évaluation d'une branche du `let` sera disponible pour être utilisé dans l'autre branche. Nous pouvons noter également que dans la construction

$$\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e$$

e_1 désigne une valeur par défaut et e_2 une fonction de composition.

Les constantes (c) sont soit des valeurs de base comme des entiers, des booléens ou des flottants, soit des opérateurs.

$$c \quad ::= \quad \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \texttt{0} \mid \ldots \mid \texttt{+} \mid \texttt{-} \mid \ldots$$

Les valeurs (v) sont définies par :

$$v \quad ::= \quad c \mid n \mid (v, v) \mid \lambda x.e \mid \texttt{process}\ e$$

Une valeur peut être une constante (c), une valeur de signal n, une paire de valeurs (v, v), une abstraction ($\lambda x.e$) ou une définition de processus ($\texttt{process}\ e$).

$$
\begin{array}{lll}
e ::= & x & \text{variable} \\
& \mid c & \text{constante} \\
& \mid (e, e) & \text{paire} \\
& \mid \lambda x.e & \text{définition de fonction} \\
& \mid e\ e & \text{application} \\
& \mid \texttt{rec}\ x \texttt{=} e & \text{définition récursive} \\
& \mid \texttt{process}\ e & \text{définition de processus} \\
& \mid e \texttt{;} e & \text{séquence} \\
& \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{and}\ x\ \texttt{=}\ e\ \texttt{in}\ e & \text{définitions parallèles} \\
& \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e & \text{déclaration de signal} \\
& \mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e & \text{test de présence} \\
& \mid \texttt{emit}\ e\ e & \text{émission} \\
& \mid \texttt{pause} & \text{attente de la fin d'instant} \\
& \mid \texttt{run}\ e & \text{exécution d'une définition de processus} \\
& \mid \texttt{pre}\ e & \text{statut précédent d'un signal} \\
& \mid \texttt{pre ?}e & \text{valeur précédente d'un signal} \\
& \mid \texttt{do}\ e\ \texttt{until}\ e(x)\ \texttt{->}\ e\ \texttt{done} & \text{préemption} \\
& \mid \texttt{do}\ e\ \texttt{when}\ e & \text{suspension}
\end{array}
$$

FIG. 3.1 – Syntaxe abstraite du noyau de REACTIVEML.

3.1.2 Traduction de ReactiveML dans le langage noyau

Les programmes REACTIVEML peuvent se traduire facilement dans ce noyau. Par exemple, la définition de processus

$$\texttt{let process}\ f\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

est un raccourci pour

$$\texttt{let}\ f\ \texttt{=}\ \lambda x.\texttt{process}\ e_1\ \texttt{in}\ e_2$$

Un certain nombre de constructions réactives de REACTIVEML sont des formes dérivées des primitives de base :

$$
\begin{array}{lcl}
\texttt{emit}\ e & \stackrel{def}{=} & \texttt{emit}\ e\ \texttt{()} \\
\texttt{present}\ e_1\ \texttt{then}\ e_2 & \stackrel{def}{=} & \texttt{present}\ e_1\ \texttt{then}\ e_2\ \texttt{else ()} \\
\texttt{present}\ e_1\ \texttt{else}\ e_2 & \stackrel{def}{=} & \texttt{present}\ e_1\ \texttt{then ()}\ \texttt{else}\ e_2 \\
\texttt{signal}\ s\ \texttt{in}\ e & \stackrel{def}{=} & \texttt{signal}\ s\ \texttt{default}\ \emptyset\ \texttt{gather}\ \lambda x.\lambda y.\{x\} \uplus y\ \texttt{in}\ e \\
\texttt{do}\ e\ \texttt{until}\ e_1\ \texttt{done} & \stackrel{def}{=} & \texttt{do}\ e_1\ \texttt{until}\ e(x)\ \texttt{-> () done} \\
\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e & \stackrel{def}{=} & \texttt{let}\ x_1\ \texttt{=}\ e_1\ \texttt{and}\ x_2\ \texttt{= () in}\ e \quad x_2 \notin fv(e)
\end{array}
$$

Un signal pur (sans valeur) est implanté avec un signal valué qui émet la valeur `()`.
Omettre une branche dans un `present` est équivalent à ne rien faire dans cette branche. Remarquons que ne pas mettre de `else` dans un `present` ne signifie pas la suppression du délai si le signal est absent.

La construction `signal` s `in` e est un raccourci pour la déclaration de signal qui collecte dans un multi-ensemble toutes les valeurs émises pendant un instant. $\emptyset$ est le multi-ensemble vide et $\uplus$ est l'union de multi-ensembles (si $m_1 = \{v_1, ..., v_n\}$ et $m_2 = \{v'_1, ..., v'_k\}$ alors $m_1 \uplus m_2 = \{v_1, ..., v_n, v'_1, ..., v'_k\}$). Remarquons que dans l'implantation, les valeurs sont collectées dans une liste et non dans un multi-ensemble. Cela permet d'avoir une syntaxe plus légère pour le filtrage et d'utiliser les fonctions de la bibliothèque standard.
Le `do/until` sans récupérateur d'échappement n'applique pas un traitement particulier en cas de préemption.
Le `let/in` à une branche est un `let/and/in` où la seconde branche ne calcule rien et termine instantanément.

De même, nous pouvons remarquer que le `let/and/in` à n branches se programme avec le `let/and/in` binaire. Les deux expressions suivantes sont équivalentes :

$$\begin{array}{l} \texttt{let}\ x_1\ \texttt{=}\ e_1 \\ \texttt{and}\ x_2\ \texttt{=}\ e_2 \\ \texttt{and}\ x_3\ \texttt{=}\ e_3 \\ \texttt{in}\ e \end{array} \quad \stackrel{def}{=} \quad \begin{array}{l} \texttt{let}\ (x_1, x_2)\ \texttt{=} \\ \quad \texttt{let}\ v_1\ \texttt{=}\ e_1 \\ \quad \texttt{and}\ v_2\ \texttt{=}\ e_2 \\ \quad \texttt{in}\ (v_1, v_2) \\ \texttt{and}\ x_3\ \texttt{=}\ e_3 \\ \texttt{in}\ e \end{array}$$

Enfin, les autres constructions peuvent se programmer ainsi :

$$\begin{array}{lcl} \texttt{nothing} & \stackrel{def}{=} & \texttt{()} \\ \texttt{halt} & \stackrel{def}{=} & \texttt{loop pause end} \\ \texttt{await immediate}\ s & \stackrel{def}{=} & \texttt{do () when}\ s \\ \texttt{await}\ s & \stackrel{def}{=} & \texttt{await immediate}\ s\texttt{;pause} \\ \texttt{await}\ s(x)\ \texttt{in}\ e & \stackrel{def}{=} & \texttt{do halt until}\ s(x)\ \texttt{->}\ e\ \texttt{done} \\ e_1 \texttt{||} e_2 & \stackrel{def}{=} & \texttt{let}\ x_1\ \texttt{=}\ e_1\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{in ()} \\ \texttt{loop}\ e\ \texttt{end} & \stackrel{def}{=} & \texttt{run}\ ((\texttt{rec}\ loop = \lambda x.\texttt{process}\ (\texttt{run}\ x\texttt{;run}\ (loop\ x)))\ \ e) \end{array}$$

`nothing` est une construction de base d'Esterel. Elle est un synonyme pour la valeur `()` en ReactiveML. L'exécution du processus `halt` ne termine jamais.
La construction `await/immediate` attend la présence d'un signal. On la programme avec un `do/when` qui termine instantanément quand le signal est présent. Il faut noter que la sémantique de la version non instantanée de `await` a une sémantique différente de celle de Esterel.
Attendre un signal pour récupérer la valeur associée peut se programmer avec un `do/until`. Pour cela, le processus `halt` est utilisé. `halt` est exécuté dans un `do/until` qui le préempte quand s est émis. L'expression e dans laquelle x a été substitué par la valeur du signal est alors exécutée.
La composition parallèle se code avec un `let/and/in` où on ne récupère pas les valeurs calculées en parallèle. Le `loop` se programme par un processus récursif.

Nous n'avons pas pris ici le noyau minimal du langage. Par exemple, les expressions suivantes sont équivalentes :

$$\begin{array}{lcl} e_1\texttt{;}e_2 & \equiv & \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \quad x \notin fv(e) \\ \texttt{pause} & \equiv & \texttt{signal}\ x\ \texttt{in present}\ x\ \texttt{else ()} \end{array}$$

La séquence se programme avec la dépendance introduite dans le `let/in`. L'instruction `pause` peut utiliser le retard pour la réaction à l'absence. Comme x est absent, l'exécution de la branche `else` de l'instruction `present` a lieu à l'instant suivant.

$$k \vdash x \qquad k \vdash c \qquad \frac{0 \vdash e}{k \vdash \lambda x.e} \qquad \frac{1 \vdash e}{k \vdash \texttt{process}\ e} \qquad \frac{0 \vdash e}{k \vdash \texttt{rec}\ x \texttt{=} e} \qquad 1 \vdash \texttt{pause}$$

$$\frac{0 \vdash e_1 \quad 0 \vdash e_2}{k \vdash e_1\ e_2} \qquad \frac{0 \vdash e_1 \quad 0 \vdash e_2}{k \vdash (e_1, e_2)} \qquad \frac{k \vdash e_1 \quad k \vdash e_2 \quad k \vdash e}{k \vdash \texttt{let}\ x_1\ \texttt{=}\ e_1\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ e}$$

$$\frac{0 \vdash e_1 \quad 0 \vdash e_2}{k \vdash \texttt{emit}\ e_1\ e_2} \qquad \frac{k \vdash e_1 \quad k \vdash e_2}{k \vdash e_1 \texttt{;} e_2} \qquad \frac{0 \vdash e \quad 1 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

$$\frac{0 \vdash e_1 \quad 0 \vdash e_2 \quad k \vdash e}{k \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e} \qquad \frac{0 \vdash e_1 \quad 1 \vdash e_2 \quad 1 \vdash e_3}{1 \vdash \texttt{do}\ e_2\ \texttt{until}\ e_1(x)\ \texttt{->}\ e_3\ \texttt{done}}$$

$$\frac{0 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{do}\ e_2\ \texttt{when}\ e_1} \qquad \frac{0 \vdash e}{k \vdash \texttt{pre}\ e} \qquad \frac{0 \vdash e}{k \vdash \texttt{pre ?}e} \qquad \frac{0 \vdash e}{1 \vdash \texttt{run}\ e}$$

FIG. 3.2 – Séparation des expressions instantanées

Ces constructions ont été gardées dans le noyau afin donner une meilleure intuition pour comprendre le formalisme des règles de sémantique.

Le `await/one` et les références qui peuvent introduire de l'indéterminisme seront traités séparément dans le chapitre 4.3.

3.2 Identification des expressions instantanées

Afin de séparer clairement les expressions *réactives* (ou séquentielles en utilisant la terminologie des circuits) dont l'exécution peut prendre plusieurs instants, des expressions *instantanées* (ou combinatoires), les expressions doivent suivre des règles de bonne formation. Cette séparation discrimine les expressions à état de celles sans état. De plus, elle permet à la fois une plus grande clarté des programmes et également d'appliquer des techniques de compilation différentes pour les expressions combinatoires et les expressions séquentielles. En particulier, nous voulons que les expressions OCAML soient exécutées sans surcoût.

Définition 1 (Bonne formation)
Une expression e bien formée doit vérifier le prédicat $k \vdash e$ défini figure 3.2 où $k \in \{0, 1\}$.

Ce prédicat se lit “dans un contexte k l'expression e est bien formée”. 0 représente le contexte des expressions instantanées et 1 le contexte réactif. Lorsque $0 \vdash e$, alors e est une expression *instantanée*. Sinon e est une expression *réactive*.

Dans la figure 3.2, une règle donnée dans un contexte k ($k \vdash e$) est un raccourci pour les deux règles $0 \vdash e$ et $1 \vdash e$. Par exemple, cela signifie que les variables et les constantes peuvent être utilisées dans tous les contextes. Une abstraction ($\lambda x.e$) peut également être utilisée dans tous les contextes alors que son corps doit nécessairement être instantané. Pour la définition d'un processus (`process` e), le corps peut être réactif. Toutes les expressions ML sont bien formées dans tous les contextes, mais les expressions comme `pause`, `run` ou `present` dont l'exécution peut se faire sur plusieurs instants ne peuvent être utilisées que dans des processus. Nous pouvons remarquer qu'aucune règle ne permet de conclure qu'une expression est typée uniquement dans un contexte 0 et pas dans un contexte 1. Par conséquent, toutes les expressions combinatoires peuvent être utilisées dans des processus.

Remarque :

> Avec ces règles de bonne formation, nous avons fait des choix dans le design du langage. Par exemple pour la paire nous aurions pu autoriser l'utilisation d'expressions réactives :
>
> $$\frac{k \vdash e_1 \quad k \vdash e_2}{k \vdash (e_1, e_2)}$$
>
> Mais dans ce cas, le programme `((emit s; 1), (pause; 2))` a plusieurs sémantiques possibles. Si l'ordre d'évaluation d'une paire est de gauche à droite, `s` est émis dès le premier instant alors que si l'ordre d'évaluation est de droite à gauche, `s` est émis au second instant. Nous pouvons aussi faire le choix d'exécuter les deux expressions en parallèle. Nous trouvons plus naturel d'interdire ces possibilités dans une paire. Elles peuvent être exprimées avec la séquence et la composition parallèle. Par exemple, on préférera écrire `(emit s || pause); (1, 2)`.
>
> Cet exemple illustre un choix dans le design du langage qui doit éviter autant que possible les écritures troublantes.

3.3 Sémantique à grands pas

La sémantique comportementale décrit la réaction d'un programme comme une succession de réductions où chaque réduction représente l'exécution d'un instant. Cette sémantique s'inspire de la *sémantique logique comportementale* de Esterel [11].

La sémantique intègre la description du comportement des parties réactives et combinatoires. Ceci permet d'obtenir une formalisation complète du langage en rendant explicite les interactions entre les deux mondes à travers un formalisme commun. Ce n'était pas le cas des sémantiques précédentes du modèle réactif, ni de la sémantique comportementale de Esterel. De plus, c'est la première sémantique à grand pas du modèle réactif.

3.3.1 Environnement de signaux

La réaction d'un programme se définit par rapport à un ensemble de signaux. Dans cette section, ces ensembles et les opérations permettant de les manipuler sont définis formellement.

Les noms de signaux sont notés n et appartiennent à un ensemble dénombrable $\mathcal{N}$. Si $N_1 \subseteq \mathcal{N}$ et $N_2 \subseteq \mathcal{N}$, nous notons $N_1 \cdot N_2$ l'union de ces deux ensembles qui est définie uniquement si $N_1 \cap N_2 = \emptyset$.

Définition 2 (Environnement de signaux)
Un environnement de signaux S *est une fonction :*

$$S \quad ::= \quad [(d_1, g_1, p_1, m_1)/n_1, ..., (d_k, g_k, p_k, m_k)/n_k]$$

qui, à un nom de signal n_i, associe un quadruplet (d_i, g_i, p_i, m_i) où d_i est la valeur par défaut de n_i, g_i est une fonction de combinaison, p_i est une paire représentant le statut du signal à l'instant précédent et sa dernière valeur et m_i le multi-ensemble des valeurs émises pendant la réaction.

Si le signal n_i est de type (τ_1, τ_2) `event` *alors les valeurs associées au signal ont les types suivants :*

$$\begin{array}{ll} d_i : \tau_2 & p_i : \texttt{bool} \times \tau_2 \\ g_i : \tau_1 \to \tau_2 \to \tau_2 & m_i : \tau_2\,\texttt{multiset} \end{array}$$

on note $S^d(n_i) = d_i$, $S^g(n_i) = g_i$, $S^p(n_i) = p_i$ et $S^v(n_i) = m_i$.

On utilise des multi-ensembles pour représenter les valeurs émises pendant un instant car ils ne sont pas ordonnés et permettent de garder plusieurs occurrences d'une même valeur. Par exemple, pour le programme :

```
emit s 1 || emit s 2 || emit s 1
```

le multi-ensemble des valeurs associées à `s` sera $\{1, 1, 2\}$.

Comme tous les signaux sont valués, si le multi-ensemble associé à un signal n dans un environnement S est vide ($S^v(n) = \emptyset$), cela signifie que n n'a pas été émis. On le note $n \notin S$. Réciproquement, si $S^v(n) \neq \emptyset$ alors n est présent dans l'environnement S (noté $n \in S$).

Définition 3 (Événement)
Un événement *E est une fonction des noms de signaux dans les multi-ensembles de valeurs :*

$$E \quad ::= \quad [m_1/n_1, ..., m_k/n_k]$$

Les événements servent à représenter des ensembles de valeurs qui sont émises sur des signaux. S^v est l'événement associé à l'environnement S.

L'union de deux événements E_1 et E_2 définit un nouvel événement $E = E_1 \sqcup E_2$ où pour chaque signal n le multi-ensemble associé est l'union des valeurs associées à n dans E_1 et E_2 :

$$\forall n \in Dom(E_1) \cup Dom(E_2) : \; E(n) = E_1(n) \uplus E_2(n)$$

avec la convention que pour tout événement E, si $n \notin Dom(E)$ alors $E(n) = \emptyset$.
De la même façon, nous définissons $E = E_1 \sqcap E_2$, l'intersection de E_1 et E_2 qui garde uniquement les valeurs qui sont à la fois dans E_1 et E_2 :

$$\forall n \in Dom(E_1) \cup Dom(E_2) : \; E(n) = E_1(n) \sqcap E_2(n)$$

Ajouter une valeur v au multi-ensemble associé à un signal n dans un environnement de signaux S se note avec l'opérateur $+$:

$$(S + [v/n])(n') = \begin{cases} S(n') & \text{si } n' \neq n \\ (S^d(n), S^g(n), S^p(n), S^v(n) \uplus \{v\}) & \text{si } n' = n \end{cases}$$

On définit la relation d'ordre $\sqsubseteq$ sur les événements que l'on étend aux environnements de signaux :

$$\begin{array}{lll} E_1 \sqsubseteq E_2 & \text{ssi} & \forall n \in Dom(E_1) : E_1(n) \subseteq E_2(n) \\ S_1 \sqsubseteq S_2 & \text{ssi} & S_1^v \sqsubseteq S_2^v \end{array}$$

Enfin on notera $\subseteq$, l'ordre sur les fonctions défini par :

$$f_1 \subseteq f_2 \text{ ssi } \forall x \in Dom(f_1) : f_1(x) = f_2(x)$$

3.3.2 Sémantique

La réaction en un instant d'une expression e en une expression e' est définie par une relation de transition de la forme :

$$N \vdash e \xrightarrow[S]{E, b} e'$$

où

- N est l'ensemble des noms de signaux créés par la réaction.
- S est l'environnement de signaux dans lequel e doit réagir. Il contient les signaux d'entrée, de sortie et les signaux locaux.
- L'événement E représente les signaux émis pendant la réaction.
- b est le statut de terminaison. C'est une valeur booléenne indiquant si l'expression e' doit être activée à l'instant suivant ou si elle a terminé sa réaction.

L'exécution d'un programme est une succession potentiellement infinie de réactions. Elle est terminée lorsque que le statut b est vrai. À chaque instant, un programme lit des entrées (I_i) et émet des sorties (O_i). L'exécution d'un instant est définie par le plus petit environnement de signaux S_i (pour l'ordre $\sqsubseteq$) tel que :

$$N_i \vdash e_i \xrightarrow[S_i]{E_i, b} e'_i$$

où

$$\begin{array}{ll} (I_i \sqcup E_i) \sqsubseteq S_i^v & (1) \\ O_i = next(S_i) & (2) \\ \forall n \in N_{i+1}.\ n \notin Dom(S_i) & (3) \\ S_i^d \subseteq S_{i+1}^d \text{ et } S_i^g \subseteq S_{i+1}^g & (4) \\ O_i \subseteq S_{i+1}^p & (5) \end{array}$$

(1) L'environnement S_i^v doit contenir les signaux d'entrée et ceux émis pendant la réaction, cela garantit la propriété de diffusion instantanée des événements. (2) La sortie O_i associe à chaque signal son statut (présent/absent) et une valeur qui est la combinaison des valeurs émises. Donc la fonction *next* qui calcule la sortie est définie par :

$$\forall n \in Dom(S).\ next(S)(n) = \begin{cases} (false, v) & \text{si } n \notin S \text{ et } S^p(n) = (b, v) \\ (true, fold\ g\ m\ d) & \text{si } n \in S \text{ et } S(n) = (d, g, p, m) \end{cases}$$

avec *fold* tel que :

$$\begin{array}{lcl} fold\ f\ (\{v_1\} \uplus m)\ v_2 & = & fold\ f\ m\ (f\ v_1\ v_2) \\ fold\ f\ \emptyset\ v & = & v \end{array}$$

La sortie associée à un signal n est donc *false* et la valeur précédente de n, s'il est absent. Sinon, c'est *true* et la combinaison des valeurs émises pendant l'instant.

(3) La condition $\forall n \in N_{i+1}.n \notin Dom(S_i)$ garantit que les noms introduits lors de la réaction sont des noms frais. (4) Les conditions $S_i^d \subseteq S_{i+1}^d$ et $S_i^g \subseteq S_{i+1}^g$ indiquent que les valeurs par défaut et les fonctions de combinaison des signaux sont gardées d'un instant à l'autre. (5) Enfin, $O_i \subseteq S_{i+1}^p$ est la transmission des valeurs des signaux pour le calcul du `pre`.

Les contraintes $S_i^d \subseteq S_{i+1}^d$ et $S_i^g \subseteq S_{i+1}^g$ ont pour conséquence de garder dans l'environnement tous les signaux créés par la réaction du programme. Mais on peut remarquer que les signaux de S_i qui ne sont pas des variables libres dans e'_i peuvent être supprimés de l'environnement de signaux S_{i+1}.

$$\emptyset \vdash v \xrightarrow[S]{\emptyset,\, true} v \qquad \frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,\, true} v_1 \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,\, true} v_2}{N_1 \cdot N_2 \vdash (e_1, e_2) \xrightarrow[S]{E_1 \sqcup E_2,\, true} (v_1, v_2)}$$

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,\, true} \lambda x.e \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,\, true} v_2 \quad N_3 \vdash e[x \leftarrow v_2] \xrightarrow[S]{E_3,\, true} v}{N_1 \cdot N_2 \cdot N_3 \vdash e_1\ e_2 \xrightarrow[S]{E_1 \sqcup E_2 \sqcup E_3,\, true} v}$$

$$\frac{N \vdash e[x \leftarrow \texttt{rec}\, x = e] \xrightarrow[S]{E,\, true} v}{N \vdash \texttt{rec}\, x = e \xrightarrow[S]{E,\, true} v} \qquad \frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,\, true} n \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,\, true} v}{N_1 \cdot N_2 \vdash \texttt{emit}\ e_1\ e_2 \xrightarrow[S]{E_1 \sqcup E_2 \sqcup [\{v\}/n],\, true} \texttt{()}}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\, true} n \quad (b, v) = S^p(n)}{N \vdash \texttt{pre}\ e \xrightarrow[S]{E,\, true} b} \qquad \frac{N \vdash e \xrightarrow[S]{E,\, true} n \quad (b, v) = S^p(n)}{N \vdash \texttt{pre ?}e \xrightarrow[S]{E,\, true} v}$$

FIG. 3.3 – Sémantique comportementale (1)

La relation de transition est définie figure 3.3 en donnant les règles pour les expressions instantanées (celles pour lesquelles $0 \vdash e$).

Nous pouvons remarquer que dans toutes les règles, le statut de terminaison b est vrai, montrant bien que l'exécution est instantanée. Pour les expressions ML, les règles sont basées sur la sémantique classique de ML à "grands pas". Nous avons seulement ajouté les événements (E) pour pouvoir exprimer l'émission de signaux dans les expressions instantanées, les environnements de signaux (S) pour pouvoir calculer les `pre` et les ensembles de noms frais (N) pour pouvoir déclarer de nouveaux signaux dans les parties instantanées.

Détaillons quelque règles de cette figure 3.3 :

- Lors de l'évaluation d'une valeur aucun signal n'est émis.
- Pour évaluer (e_1, e_2), on évalue les deux branches dans le même instant. Ces deux branches réagissent dans le même environnement de signaux S pour avoir une vision globale et cohérente des signaux présents pendant un instant. E_1 et E_2 sont les signaux émis par la réaction de e_1 et e_2, donc la réaction de (e_1, e_2) émet l'union de E_1 et E_2. Enfin, les signaux créés par cette réaction sont ceux créés par la réaction de e_1 et e_2. Les noms de ces signaux sont pris dans N_1 et N_2.
- L'expression `emit` e_1 e_2 évalue e_1 en un signal n et ajoute le résultat de l'évaluation de e_2 dans l'ensemble des valeurs émises pendant la réaction.
- Pour l'évaluation des `pre`, le paramètre doit être évalué en un signal puis on va chercher dans l'environnement de signaux le statut ou la valeur précédente du signal.

On commente maintenant les règles de la figure 3.4 qui présentent les règles de sémantique du noyau réactif.

$$\frac{N_1 \vdash e \xrightarrow[S]{E,\,true} \texttt{process}\ e_1 \quad N_2 \vdash e_1 \xrightarrow[S]{E_1,\,b} e_1'}{N_1 \cdot N_2 \vdash \texttt{run}\ e \xrightarrow[S]{E \sqcup E_1,\,b} e_1'} \qquad \emptyset \vdash \texttt{pause} \xrightarrow[S]{\emptyset,\,false} \texttt{()}$$

$$\frac{N \vdash e_1 \xrightarrow[S]{E_1,\,false} e_1'}{N \vdash e_1 \,\texttt{;}\, e_2 \xrightarrow[S]{E_1,\,false} e_1' \,\texttt{;}\, e_2} \qquad \frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,\,true} e_1' \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,\,b} e_2'}{N_1 \cdot N_2 \vdash e_1 \,\texttt{;}\, e_2 \xrightarrow[S]{E_1 \sqcup E_2,\,b} e_2'}$$

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,\,b_1} e_1' \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,\,b_2} e_2' \quad b_1 \wedge b_2 = false}{N_1 \cdot N_2 \vdash \texttt{let}\ x_1 \texttt{ = } e_1\ \texttt{and}\ x_2 \texttt{ = } e_2\ \texttt{in}\ e \xrightarrow[S]{E_1 \sqcup E_2,\,false} \texttt{let}\ x_1 \texttt{ = } e_1'\ \texttt{and}\ x_2 \texttt{ = } e_2'\ \texttt{in}\ e}$$

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,\,true} v_1 \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,\,true} v_2 \quad N_3 \vdash e[x_1 \leftarrow v_1, x_2 \leftarrow v_2] \xrightarrow[S]{E,\,b} e'}{N_1 \cdot N_2 \cdot N_3 \vdash \texttt{let}\ x_1 \texttt{ = } e_1\ \texttt{and}\ x_2 \texttt{ = } e_2\ \texttt{in}\ e \xrightarrow[S]{E_1 \sqcup E_2 \sqcup E,\,b} e'}$$

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,\,true} v_1 \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,\,true} v_2 \quad S(n) = (v_1, v_2, (false, v_1), m) \quad N_3 \vdash e[x \leftarrow n] \xrightarrow[S]{E,\,b} e'}{N_1 \cdot N_2 \cdot N_3 \cdot \{n\} \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e \xrightarrow[S]{E_1 \sqcup E_2 \sqcup E,\,b} e'}$$

$$\frac{N_1 \vdash e \xrightarrow[S]{E,\,true} n \quad n \in S \quad N_2 \vdash e_1 \xrightarrow[S]{E_1,\,b} e_1'}{N_1 \cdot N_2 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E \sqcup E_1,\,b} e_1'}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\,true} n \quad n \notin S}{N \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E,\,false} e_2}$$

FIG. 3.4 – Sémantique comportementale (2)

- `run` e évalue e en une définition de processus et l'exécute.
- `pause` se réduit en `()`. Son statut de terminaison est égal à *false* pour marquer l'attente de la fin de l'instant.
- Les règles pour la séquence illustrent l'utilisation du statut de terminaison b. L'expression e_2 est exécutée seulement si e_1 termine dans l'instant ($b = \mathit{true}$).
- Le comportement du `let/and/in` est d'exécuter e_1 et e_2 en parallèle. Quand ces expressions sont réduites en des valeurs v_1 et v_2, alors x_1 et x_2 sont substitués respectivement par v_1 et v_2 dans e.
- `signal` x `default` e_1 `gather` e_2 `in` e déclare un nouveau signal x. La valeur par défaut (e_1) et la fonction de combinaison (e_2) sont évaluées au moment de la déclaration. Le nom x est substitué par un nom frais n dans e. Dans l'environnement des signaux, le `pre` du signal est initialisé avec le statut absent et avec la valeur par défaut. Pour le multi-ensemble m, les valeurs émises pendant l'instant doivent être *devinées*. Cela veut dire que la dérivation complète du programme doit vérifier que m contient bien les valeurs émises pendant l'instant.
- Dans le test de présence d'un signal, si le signal est présent, la branche `then` est exécutée instantanément. Sinon, comme le statut de terminaison est $b = \mathit{false}$, la branche `else` est exécutée à l'instant suivant.

Enfin, nous présentons la sémantique des instructions de suspension et préemption figure 3.5.

- Le `do/when` exécute son corps uniquement quand le signal qui le contrôle est présent. Quand le corps est actif et termine son exécution, le `do/when` termine aussi instantanément. Dans chaque règle on évalue le signal mais à la première activation l'expression e est évaluée en une valeur n. Donc pour les activations aux instants suivant le `do/when` sera toujours contrôlé par le même signal n.
- Enfin, le `do/until` active toujours son corps. Si le corps termine, le `do/until` se réécrit en la valeur de son corps. En cas de préemption, il récupère la valeur associée au signal pour pouvoir exécuter le code de traitement de préemption à la prochaine activation.

Remarque :

> L'ensemble N est utilisé comme comme générateur de noms frais. Pour simplifier la présentation l'ensemble N pourrait être supprimé de toutes les règles et la fonction *gensym* qui génère des noms frais serait alors utilisée dans la règle de `signal/in`. Ainsi les règles auraient la forme des règles de la sémantique comportementale de ESTEREL :
>
> $$e \xrightarrow[S]{E,b} e'$$

Nous terminons cette description de la sémantique par la définition de la relation d'équivalence entre deux expressions.

Définition 4 (Équivalence)
Deux expressions e_1 et e_2 sont équivalentes si pour tout environnement de signaux S tel que

$$N_1 \vdash e_1 \xrightarrow[S]{E_1,b_1} e'_1 \quad \text{et} \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,b_2} e'_2$$

alors $E_1 = E_2$, $b_1 = b_2$ et $e'_1 = e'_2$.

$$\frac{N \vdash e \xrightarrow[S]{E,\, true} n \quad n \notin S}{N \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{E,\, false} \texttt{do}\ e_1\ \texttt{when}\ n}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\, true} n \quad n \in S \quad N_1 \vdash e_1 \xrightarrow[S]{E_1,\, false} e_1'}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{E \sqcup E_1,\, false} \texttt{do}\ e_1'\ \texttt{when}\ n}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\, true} n \quad n \in S \quad N_1 \vdash e_1 \xrightarrow[S]{E_1,\, true} v}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{E \sqcup E_1,\, true} v}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\, true} n \quad N_1 \vdash e_1 \xrightarrow[S]{E_1,\, true} v}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e(x)\ \texttt{->}\ e_2\ \texttt{done} \xrightarrow[S]{E \sqcup E_1,\, true} v}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\, true} n \quad n \in S \quad N_1 \vdash e_1 \xrightarrow[S]{E_1,\, false} e_1' \quad S(n) = (d, g, p, m) \quad v = \mathit{fold}\ g\ m\ d}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e(x)\ \texttt{->}\ e_2\ \texttt{done} \xrightarrow[S]{E \sqcup E_1,\, false} e_2[x \leftarrow v]}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\, true} n \quad n \notin S \quad N_1 \vdash e_1 \xrightarrow[S]{E_1,\, false} e_1'}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e(x)\ \texttt{->}\ e_2\ \texttt{done} \xrightarrow[S]{E \sqcup E_1,\, false} \texttt{do}\ e_1'\ \texttt{until}\ n(x)\ \texttt{->}\ e_2\ \texttt{done}}$$

FIG. 3.5 – Sémantique comportementale (3)

3.3.3 Exemples de dérivations

Sémantique des expressions dérivées

À partir de la sémantique des expressions du noyau du langage, les règles pour les expressions dérivées peuvent être construites.

Section 3.1.2, nous avons défini e_1`||`e_2 de la façon suivante : `let` x_1 `=` e_1 `and` x_2 `=` e_2 `in ()`. Les dérivations possibles pour cette expression sont :

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, b_1} e_1' \quad N_2 \vdash e_2 \xrightarrow[S]{E_2, b_2} e_2' \quad b_1 \wedge b_2 = \mathit{false}}{N_1 \cdot N_2 \vdash \texttt{let } x_1 \texttt{ = } e_1 \texttt{ and } x_2 \texttt{ = } e_2 \texttt{ in ()} \xrightarrow[S]{E_1 \sqcup E_2, \mathit{false}} \texttt{let } x_1 \texttt{ = } e_1' \texttt{ and } x_2 \texttt{ = } e_2' \texttt{ in ()}}$$

et

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, \mathit{true}} v_1 \quad N_2 \vdash e_2 \xrightarrow[S]{E_2, \mathit{true}} v_2 \quad \emptyset \vdash \texttt{()} \xrightarrow[S]{\emptyset, \mathit{true}} \texttt{()}}{N_1 \cdot N_2 \vdash \texttt{let } x_1 \texttt{ = } e_1 \texttt{ and } x_2 \texttt{ = } e_2 \texttt{ in ()} \xrightarrow[S]{E_1 \sqcup E_2, \mathit{true}} \texttt{()}}$$

Donc la composition parallèle exécute e_1 et e_2 et termine instantanément quand les deux branches ont terminé leur exécution.

L'expression `await immediate` e est définie par : `do () when` e. Si e s'évalue en un signal n, on a les dérivations suivantes :

$$\frac{N \vdash e \xrightarrow[S]{E, \mathit{true}} n \quad n \notin S}{N \vdash \texttt{do () when } e \xrightarrow[S]{E, \mathit{false}} \texttt{do () when } n} \qquad \frac{N \vdash e \xrightarrow[S]{E, \mathit{true}} n \quad n \in S \quad \emptyset \vdash \texttt{()} \xrightarrow[S]{\emptyset, \mathit{true}} \texttt{()}}{N \vdash \texttt{do () when } e \xrightarrow[S]{E, \mathit{true}} \texttt{()}}$$

Donc les règles pour `await/immediate` sont :

$$\frac{N \vdash e \xrightarrow[S]{E, \mathit{true}} n \quad n \notin S}{N \vdash \texttt{await immediate } e \xrightarrow[S]{E, \mathit{false}} \texttt{await immediate } n} \qquad \frac{N \vdash e \xrightarrow[S]{E, \mathit{true}} n \quad n \in S}{N \vdash \texttt{await immediate } e \xrightarrow[S]{E, \mathit{false}} \texttt{()}}$$

De même, on peut vérifier l'équivalence entre `pause` et l'expression suivante :

$$\texttt{signal } x \texttt{ in present } x \texttt{ then () else ()}$$

La dérivation de cette expression est :

$$\frac{\dfrac{N \vdash n \xrightarrow[S]{\emptyset, \mathit{true}} n \quad n \notin S}{N \vdash \texttt{present } n \texttt{ then () else ()} \xrightarrow[S]{\emptyset, \mathit{false}} \texttt{()}}}{N \cdot \{n\} \vdash \texttt{signal } x \texttt{ in present } x \texttt{ then () else ()} \xrightarrow[S]{\emptyset, \mathit{false}} \texttt{()}}$$

Comme n ne peut pas échapper de sa portée dans `present` n `then () else ()`, on peut faire l'hypothèse $S^v(n) = \emptyset$ et donc $n \notin S$. Ceci donne la règle attendue pour `pause` :

$$N \vdash \texttt{pause} \xrightarrow[S]{\emptyset, \mathit{false}} \texttt{()}$$

Exemple de dialogue instantané

Nous illustrons maintenant la sémantique sur des petits exemples. Commençons avec un exemple de composition parallèle avec dialogue instantané entre les branches.

$$(\texttt{emit}\ n_1\texttt{;present}\ n_2\ \texttt{then emit}\ n_3)\texttt{||}(\texttt{present}\ n_1\ \texttt{then emit}\ n_2)$$

Dans cet exemple, la branche gauche émet n_1 qui déclenche l'émission de n_2 dans la branche droite. Comme n_2 est présent, n_3 peut être émis par la branche gauche.

Écrivons la dérivation de cette expression dans l'environnement S défini comme suit :

$$\begin{array}{llcl} \forall i \in \{1,2,3\} & S^d(n_i) & = & \texttt{()} \\ & S^g(n_i) & = & \lambda x.\lambda y.\texttt{()} \\ & S^p(n_i) & = & (\mathit{false}, \emptyset) \\ \text{et} & S^v & = & [\{\texttt{()}\}/n_1; \{\texttt{()}\}/n_2; \{\texttt{()}\}/n_3] \end{array}$$

$$\frac{D_1 \qquad D_2}{\emptyset \vdash (\texttt{emit}\ n_1\texttt{;present}\ n_2\ \texttt{then emit}\ n_3)\texttt{||}(\texttt{present}\ n_1\ \texttt{then emit}\ n_2) \xrightarrow[S]{E,\, true} \texttt{()}}$$

avec $E = [\{\texttt{()}\}/n_1; \{\texttt{()}\}/n_2; \{\texttt{()}\}/n_3]$

D_1 :

$$\frac{\dfrac{}{\emptyset \vdash \texttt{emit}\ n_1 \xrightarrow[S]{[\{\texttt{()}\}/n_1],\, true} \texttt{()}} \qquad \dfrac{n_2 \in S \quad \emptyset \vdash \texttt{emit}\ n_3 \xrightarrow[S]{[\{\texttt{()}\}/n_3],\, true} \texttt{()}}{\emptyset \vdash \texttt{present}\ n_2\ \texttt{then emit}\ n_3 \xrightarrow[S]{[\{\texttt{()}\}/n_3],\, true} \texttt{()}}}{\emptyset \vdash \texttt{emit}\ n_1\texttt{;present}\ n_2\ \texttt{then emit}\ n_3 \xrightarrow[S]{[\{\texttt{()}\}/n_1;\{\texttt{()}\}/n_3],\, true} \texttt{()}}$$

D_2 :

$$\frac{n_1 \in S \quad \emptyset \vdash \texttt{emit}\ n_2 \xrightarrow[S]{[\{\texttt{()}\}/n_2],\, true} \texttt{()}}{\emptyset \vdash \texttt{present}\ n_1\ \texttt{then emit}\ n_2 \xrightarrow[S]{[\{\texttt{()}\}/n_1],\, true} \texttt{()}}$$

Nous observons dans cet exemple que la propriété de diffusion ($E \in S^v$) oblige à avoir n_1, n_2 et n_3 présents dans l'environnement et donc permet la communication instantanée entre les deux branches.

Illustration du modèle réactif

Regardons maintenant le comportement de la sémantique sur deux exemples qui ne sont pas causaux en ESTEREL. Le premier n'est pas causal en ESTEREL car on ne peut pas lui donner de sémantique.

$$\texttt{present}\ n\ \texttt{then () else emit}\ n$$

En REACTIVEML, comme la réaction à l'absence est retardée, on a la dérivation suivante si on suppose qu'il n'y a pas de signaux d'entrée :

$$\frac{n \notin S}{\emptyset \vdash \texttt{present}\ n\ \texttt{then () else emit}\ n \xrightarrow[S]{\emptyset,\, false} \texttt{emit}\ n}$$

Le second exemple n'est pas causal en ESTEREL car on peut lui donner deux sémantique.

$$\texttt{present}\ n\ \texttt{then emit}\ n\ \texttt{else ()}$$

Ici aussi, en fonction des hypothèses faites sur l'environnement des signaux, on peut obtenir deux dérivations différentes. Si on suppose n absent de l'environnement S_1, on a :

$$\frac{n \notin S_1}{\emptyset \vdash \texttt{present}\ n\ \texttt{then emit}\ n\ \texttt{else ()} \xrightarrow[S_1]{\emptyset, \mathit{false}} \texttt{()}}$$

Et si on suppose que n est présent dans S_2 :

$$\frac{n \in S_2 \quad \emptyset \vdash \texttt{emit}\ n \xrightarrow[S_2]{[\{\texttt{()}\}/n], \mathit{true}} \texttt{()}}{\emptyset \vdash \texttt{present}\ n\ \texttt{then emit}\ n\ \texttt{else ()} \xrightarrow[S_2]{[\{\texttt{()}\}/n], \mathit{true}} \texttt{()}}$$

Ces deux dérivations sont valides. Mais contrairement à ESTEREL, il y a une contrainte supplémentaire : la sémantique est donnée par le plus petit environnement dans lequel le programme peut réagir. On sait donc que si n n'est pas donné en entrée, alors il n'est pas émis.

3.3.4 Discussion sur les différences avec la sémantique d'Esterel

L'idée de décrire la réaction du programme dans un environnement qui contient toute l'information sur les signaux qui vont être émis pendant la réaction est reprise de la sémantique logique comportementale d'ESTEREL [11]. Pour adapter la sémantique d'ESTEREL au modèle réactif, nous avons dû ajouter du retard au test à l'absence, mais nous avons fait également des modifications plus profondes. La principale difficulté vient du traitement des signaux valués alors que la sémantique d'ESTEREL ne traite que les signaux purs.

La première conséquence de cet ajout concerne la définition des environnements de signaux. Notre environnement S associe à chaque signal une valeur par défaut, une fonction de combinaison, la valeur précédente du signal et le multi-ensemble des valeurs émises. Dans la sémantique de ESTEREL, comme il n'y a que des signaux purs, l'environnement des signaux peut être représenté simplement comme l'ensemble des signaux émis.

La seconde conséquence concerne la définition des signaux locaux. Contrairement à ESTEREL, des signaux (ou des processus avec des signaux libres) peuvent être émis sur les signaux. Ceci peut conduire à des phénomènes d'échappement de portée. Nous ne pouvons donc pas avoir les règles d'ESTEREL suivantes pour la déclaration de signaux :

$$\frac{e \xrightarrow[S+s]{E,b} e' \quad s \in E}{\texttt{signal}\ s\ \texttt{in}\ e \xrightarrow[S]{E\backslash s, b} \texttt{signal}\ s\ \texttt{in}\ e'} \qquad \frac{e \xrightarrow[S\backslash s]{E,b} e' \quad s \notin E}{\texttt{signal}\ s\ \texttt{in}\ e \xrightarrow[S]{E, b} \texttt{signal}\ s\ \texttt{in}\ e'}$$

En effet, celles-ci ne permettent pas d'utiliser le signal s en dehors de sa portée car il n'est pas défini dans l'environnement S.

La première solution proposée pour résoudre ce problème a été de donner la sémantique modulo une relation d'équivalence comme pour le π-calcul [75]. L'idée est de définir la commutativité de `signal` avec les autres constructions et d'ajouter une règle d'équivalence dans la sémantique. Par exemple `signal` commute avec `do/when` ou la composition parallèle mais ne commute pas avec `loop` :

$$\begin{array}{lcll} \texttt{do signal } s \texttt{ in } e_1 \texttt{ when } e & \equiv & \texttt{signal } s \texttt{ in do } e_1 \texttt{ when } e & \text{si } s \notin fv(e) \\ e_1 \texttt{||signal } s \texttt{ in } e_2 & \equiv & \texttt{signal } s \texttt{ in } e_1 \texttt{||} e_2 & \text{si } s \notin fv(e_1) \\ & \ldots & & \end{array}$$

La règle d'équivalence ajoutée dans la sémantique est la suivante :

$$\frac{e_1 \equiv e_2 \quad e_2 \xrightarrow[S]{E,b} e_2' \quad e_2' \equiv e_1'}{e_1 \xrightarrow[S]{E,b} e_1'}$$

Cette approche permet de faire remonter la déclaration du signal aussi haut que nécessaire.

Nous avons préféré une solution plus simple qui s'inspire de la sémantique des langages avec références. L'idée est de rendre l'environnement S global. Ainsi, il n'y a plus de problèmes de portée car tous les signaux sont accessibles. Afin d'éviter les problèmes de collision entre les noms de signaux, un nom unique leur est attribué au moment de leur déclaration.

Enfin, l'originalité de notre sémantique par rapport à celle de Esterel est de prendre en compte à la fois la partie réactive et la partie combinatoire.

Une variation très intéressante de la sémantique comportementale d'Esterel a été présentée dans [102]. Elle permet de rendre déterministe la sémantique d'Esterel. L'idée principale de cette sémantique est d'intégrer l'analyse de causalité et la contrainte $E \sqsubseteq S$ dans les règles de `signal`. Ceci est réalisé en dérivant le corps de `signal` s `in` e en faisant les deux hypothèses "s présent" et "s absent". En fonction des signaux qui sont émis par ces deux dérivations, la règle de `signal` détermine quelle était la bonne hypothèse. S'il n'y avait pas le problème d'échappement de portée en ReactiveML, nous aurions pu utiliser une approche similaire. Les règles auraient la forme suivante :

$$\frac{e \xrightarrow[S\backslash s]{E^-,b^-} e^- \quad s \in E^- \quad e \xrightarrow[S+s]{E^+,b^+} e^+ \quad s \in E^+}{\texttt{signal } s \texttt{ in } e \xrightarrow[S]{E^+\backslash s,b^+} \texttt{signal } s \texttt{ in } e^+} \qquad \frac{e \xrightarrow[S\backslash s]{E^-,b^-} e^- \quad s \notin E^-}{\texttt{signal } s \texttt{ in } e \xrightarrow[S]{E^-,b^-} \texttt{signal } s \texttt{ in } e^-}$$

Ces règles expriment que l'hypothèse "s est absent" est *plus prioritaire* que "s est présent". L'hypothèse "s est présent" est sélectionnée uniquement lorsque l'hypothèse "s est absent" est fausse ($s \in E^-$).

3.4 Déterminisme et unicité

Nous présentons maintenant les propriétés principales de la sémantique comportementale. La première est le *déterminisme* : dans un environnement de signaux donné, un programme ne peut réagir que d'une seule façon. La seconde propriété que nous nommons *unicité* dit que si un programme est réactif, alors il existe un unique plus petit environnement de signaux dans lequel il peut réagir. Nous reprenons la définition de réactif donnée dans [11] : il existe au moins un environnement de signaux S tel que $N \vdash e \xrightarrow[S]{E,b} e'$.

La combinaison de ces deux propriétés garantit que tous les programmes réactifs sont corrects. Cette propriété n'est pas vraie en Esterel. Elle montre qu'il n'y a pas besoin d'analyse de causalité en ReactiveML, i.e, tous les programmes sont causaux.

Propriété 1 (Déterminisme)
Pour toute expression e, la sémantique comportementale est déterministe. C'est à dire :
$\forall e, \forall S, \forall N.$
si $\forall n \in Dom(S).\ S^g(n) = f$ *et* $f(x, f(y, z)) = f(y, f(x, z))$
et $N \vdash e \xrightarrow[S]{E_1, b_1} e'_1$ *et* $N \vdash e \xrightarrow[S]{E_2, b_2} e'_2$
alors $(E_1 = E_2 \wedge b_1 = b_2 \wedge e'_1 = e'_2)$

Démonstration :

Nous ne présentons que le cas le plus intéressant de la preuve.
Cas `do` e_1 `until` $e(x)$ `->` e_2 `done` **avec préemption :** Supposons que l'on ait les deux dérivations suivantes :

$$\frac{\begin{array}{c} N \vdash e \xrightarrow[S]{E_1,\, true} n_1 \quad n_1 \in S \quad N_1 \vdash e_1 \xrightarrow[S]{E_{1_1},\, true} e'_{1_1} \\ S(n_1) = (d_1, g_1, p_1, m_1) \quad v_1 = fold\ g_1\ m_1\ d_1 \end{array}}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e(x)\ \texttt{->}\ e_2\ \texttt{done} \xrightarrow[S]{E_1 \sqcup E_{1_1},\, false} e_2[x \leftarrow v_1]}$$

$$\frac{\begin{array}{c} N \vdash e \xrightarrow[S]{E_2,\, true} n_2 \quad n_2 \in S \quad N_1 \vdash e_1 \xrightarrow[S]{E_{1_2},\, true} e'_{1_2} \\ S(n_2) = (d_2, g_2, p_2, m_2) \quad v_2 = fold\ g_2\ m_2\ d_2 \end{array}}{N \cdot N_1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e(x)\ \texttt{->}\ e_2\ \texttt{done} \xrightarrow[S]{E_2 \sqcup E_{1_2},\, false} e_2[x \leftarrow v_2]}$$

Par induction on a $E_1 = E_2$ et $n_1 = n_2$ et également $E_{1_1} = E_{1_2}$ et $e'_{1_1} = e'_{1_2}$ donc $S(n_1) = S(n_2) = (d, g, p, m)$. Avec la propriété d'associativité et de commutativité de la fonction de combinaison g, nous sommes sûrs que *fold* est déterministe, donc $v_1 = v_2$. Par conséquence, $E_1 \sqcup E_{1_1} = E_2 \sqcup E_{1_2}$ et $e_2[x \leftarrow v_1] = e_2[x \leftarrow v_2]$. □

L'associativité et la commutativité des fonctions de combinaison expriment le fait qu'elles ne doivent pas dépendre de l'ordre des émissions pendant l'instant. C'est une contrainte assez forte mais même si elle n'est pas satisfaite le programme peut être déterministe. Par exemple, s'il n'y a pas de multi-émission, la fonction de combinaison n'a pas à être associative et commutative. Ou si la fonction de combinaison construit la liste des valeurs émises et que toutes les opérations faites sur cette liste ne dépendent pas de l'ordre des éléments, alors le programme reste déterministe.

Nous donnons maintenant la propriété d'unicité.

Propriété 2 (Unicité)
Pour toute expression e, soit $\mathcal{S}$ l'ensemble des environnements de signaux tel que

$$\mathcal{S} = \left\{ S \mid \exists N, E, b.\ N \vdash e \xrightarrow[S]{E, b} e' \right\}$$

alors il existe un unique plus petit environnement $(\sqcap \mathcal{S})$ tel que

$$\exists N, E, b.\ N \vdash e \xrightarrow[\sqcap \mathcal{S}]{E, b} e'$$

Démonstration :

La preuve de cette propriété est basée sur le lemme suivant qui dit que si un programme peut réagir dans deux environnements de signaux différents, alors il peut réagir dans l'intersection de ces deux environnements.

De plus, $(S, \sqsubseteq, \sqcup, \sqcap)$ définit un treillis avec un minimum. Donc l'intersection de tous les environnements dans lesquels l'expression peut réagir est unique. $\square$

Le lemme suivant est basé sur l'absence de réaction instantanée à l'absence d'événement. Cette caractéristique de la sémantique garantit que l'absence d'un signal ne peut pas générer des émissions.

Remarque :

On peut constater sur l'exemple suivant que cette caractéristique n'est pas présente en Esterel.

$$\texttt{present}\ n_1\ \texttt{then ()}\ \texttt{else emit}\ n_2$$

En Esterel, l'absence de n_1 provoque l'émission de n_2 alors qu'avec le modèle réactif, si n_1 est absent, rien n'est émis dans l'instant.

Lemme 1

Pour toute expression e, soit S_1 et S_2 deux environnements dans lesquels e peut réagir :

$$N_1 \vdash e \xrightarrow[S_1]{E_1, b_1} e_1 \qquad N_2 \vdash e \xrightarrow[S_2]{E_2, b_2} e_2$$

Soit S_3 tel que $S_3^v = S_1^v \sqcap S_2^v$. Alors il existe E_3, b_3 et e_3 tels que

$$N_3 \vdash e \xrightarrow[S_3]{E_3, b_3} e_3 \quad \textit{et} \quad b_3 \Rightarrow (b_1 \wedge b_2) \quad \textit{et} \quad E_3 \sqsubseteq (E_1 \sqcap E_2) \quad \textit{et} \quad N_3 \subseteq (N_1 \cap N_2)$$

Démonstration :

La preuve se fait par induction sur les dérivations. On présente ici seulement le cas de `present`.

Cas `present` e `then` e_1 `else` e_2 : dans les deux règles de `present`, on commence par évaluer l'expression e. Par induction, on a :

$$\forall i \in \{1, 2, 3\} \ : \ N_i \vdash e \xrightarrow[S_i]{E_i,\, true} n \quad \text{et} \quad E_3 \sqsubseteq (E_1 \sqcap E_2) \quad \text{et} \quad N_3 \subseteq (N_1 \cap N_2)$$

On regarde maintenant les dérivations possibles en fonction de la présence de n dans S_3 :

– Cas $n \in S_3$: par définition de S_3, $n \in S_3 \Rightarrow (n \in S_1 \wedge n \in S_2)$. Donc

$$\frac{N_1 \vdash e \xrightarrow[S_1]{E_1,\, true} n \quad n \in S_1 \quad N_{1_1} \vdash e_1 \xrightarrow[S_1]{E_{1_1}, b_1} e'_{1_1}}{N_1 \cdot N_{1_1} \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S_1]{E_1 \sqcup E_{1_1}, b_1} e'_{1_1}}$$

$$\frac{N_2 \vdash e \xrightarrow[S_2]{E_2,\, true} n \quad n \in S_2 \quad N_{1_2} \vdash e_1 \xrightarrow[S_2]{E_{1_2}, b_2} e'_{1_2}}{N_2 \cdot N_{1_2} \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S_2]{E_2 \sqcup E_{1_2}, b_2} e'_{1_2}}$$

Par induction et avec la règle de `present` :

$$\frac{N_3 \vdash e \xrightarrow[S_3]{E_3,\, true} n \quad n \in S_3 \quad N_{1_3} \vdash e_1 \xrightarrow[S_3]{E_{1_3},\, b_3} e'_{1_3}}{N_3 \cdot N_{1_3} \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S_3]{E_3 \sqcup E_{1_3},\, b_3} e'_{1_3}}$$

$$\text{et} \quad b_3 \Rightarrow (b_1 \wedge b_2) \quad \text{et} \quad E_{1_3} \sqsubseteq E_{1_1} \sqcap E_{1_2} \quad \text{et} \quad N_{1_3} \subseteq (N_{1_1} \cap N_{1_2})$$

Avec ces hypothèses, nous pouvons montrer que $(E_3 \sqcup E_{1_3}) \sqsubseteq (E_1 \sqcup E_{1_1}) \sqcap (E_2 \sqcup E_{2_2})$ et que $N_3 \cdot N_{1_3} \subseteq (N_1 \cdot N_{1_1} \cap N_2 \cdot N_{1_2})$

– Cas $n \notin S_3$: dans ce cas, l'expression peut réagir :

$$\frac{N_3 \vdash e \xrightarrow[S]{E_3,\, true} n \quad n \notin S}{N_3 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E_3,\, false} e_2}$$

et on peut montrer trivialement que $false \Rightarrow (b_1 \wedge b_2)$ et $E_3 \sqsubseteq (E_1 \sqcap E_2)$ et $N_3 \subseteq (N_1 \cap N_2)$

□

3.4.1 Propriétés des expressions combinatoires

Nous terminons avec deux remarques sur les expressions instantanées. La première est la correction de l'analyse d'instantanéité.

Propriété 3
Pour toute expression e instantanée ($0 \vdash e$) et pour tout environnement S tel que l'expression e peut réagir ($N \vdash e \xrightarrow[S]{E,b} e'$), alors la réaction de e termine instantanément ($b = true$)

$$\forall e, S \quad \textit{tels que} \quad 0 \vdash e \quad \textit{et} \quad N \vdash e \xrightarrow[S]{E,b} e' \quad \textit{alors} \quad b = true$$

Démonstration :

Cette démonstration est directe par induction sur le nombre de règles de la réaction. □

L'autre remarque est que pour toute expression e, pour tout environnement S tel que l'expression e termine instantanément ($N \vdash e \xrightarrow[S]{E,\, true} e'$) alors e' est une valeur.

3.5 Conclusion

Cette sémantique comportementale est la première sémantique que nous avons définie pour ReactiveML. Il est intéressant d'avoir défini une sémantique à grands pas. Cela permet de définir la réaction d'un instant sans prendre en compte l'ordonnancement à l'intérieur de l'instant. Ainsi c'est un bon formalisme pour prouver le déterminisme et l'unicité de la réaction.

En revanche, cette sémantique ne permet pas de décrire comment construire la réaction d'un instant. De plus, nous ne pouvons pas définir la sémantique des références dans cette sémantique.

Résumé

Ce chapitre commence par la présentation du noyau du langage avec une première analyse permettant d'identifier les expressions dont l'exécution est instantanée. Puis, une sémantique à grands pas du noyau est présentée et illustrée sur des exemples. Enfin, les propriétés de déterminisme et d'unicité de cette sémantique sont démontrées.

Cette sémantique s'inspire de la sémantique comportementale de ESTEREL. L'exécution d'un instant est définie par une relation de la forme :

$$N \vdash e \xrightarrow[S]{E, b} e'$$

La particularité de cette sémantique est qu'il faut connaître les signaux qui vont être émis par la réaction de e avant de faire la dérivation : la dérivation vérifie seulement que les hypothèses faites sur l'environnement ne sont pas fausses. Ainsi cette sémantique s'abstrait de l'ordonnancement intra-instant. En contrepartie de ce haut niveau d'abstraction, un interprète ne peut pas être directement défini à partir de cette sémantique.

Chapitre 4

Sémantique opérationnelle

La sémantique comportementale que nous avons définie dans le chapitre 3 n'est pas opérationnelle : elle présente ce qu'une réaction doit vérifier mais pas comment la calculer. En particulier, l'environnement de signaux doit être *deviné* avant la réaction. Nous présentons ici une sémantique à petits pas où la réaction construit l'environnement de signaux. Ce travail est publié dans [69].

4.1 Sémantique à petits pas

La sémantique opérationnelle est décomposée en deux étapes. La première décrit la réaction pendant l'instant comme une succession de micro-réactions. La seconde étape, appelée *réaction de fin d'instant* prépare la réaction pour l'instant suivant.

4.1.1 Sémantique à réduction

La première étape de la sémantique est une extension de la sémantique à réduction de ML [88, 107]. La réaction d'un instant est représentée par une succession de réactions de la forme $e/S \to e'/S'$ où S est un *environnement de signaux* défini comme dans le chapitre 3.3.1. Ces réductions définissent la réaction du programme tout en construisant l'ensemble des valeurs émises.

Pour définir la réaction $\to$, on commence par se donner des axiomes pour la relation de réduction en tête de terme ($\to_\varepsilon$). Ces axiomes sont donnés dans la figure 4.1.

– Les axiomes pour l'application et le `rec` sont les mêmes que pour ML.
– La règle pour la séquence indique qu'une fois que la branche gauche est réduite en une valeur, alors cette valeur peut être oubliée.
– Le `let/and/in` peut se réduire lorsque les deux expressions calculées en parallèle sont réduites en des valeurs.
– Le `run` s'applique à une définition de processus. Il permet son évaluation.
– `emit` n v se réduit en `()` et ajoute v au multi-ensemble des valeurs émises sur n.
– La construction `present` peut être réduite seulement si le signal est présent dans l'environnement.
– La réduction de la déclaration d'un signal x dans une expression e renomme x en n où n est un nom frais appartenant à $\mathcal{N}$. n est ajouté à l'environnement de signaux avec v_1 comme valeur par défaut, v_2 comme fonction de combinaison et (false, v_1) pour l'initialisation du

$$\lambda x.e\ v/S \rightarrow_\varepsilon e[x \leftarrow v]/S \qquad \texttt{rec}\,x\texttt{=}e/S \rightarrow_\varepsilon e[x \leftarrow \texttt{rec}\,x\texttt{=}e]/S \qquad v\texttt{;}e/S \rightarrow_\varepsilon e/S$$

$$\texttt{run (process } e\texttt{)}/S \rightarrow_\varepsilon e/S \qquad \texttt{let } x_1 \texttt{ = } v_1 \texttt{ and } x_2 \texttt{ = } v_2 \texttt{ in } e/S \rightarrow_\varepsilon e[x_1 \leftarrow v_1, x_2 \leftarrow v_2]/S$$

$$\texttt{emit } n\ v/S \rightarrow_\varepsilon \texttt{()}/S + [v/n] \qquad \texttt{present } n \texttt{ then } e_1 \texttt{ else } e_2/S \rightarrow_\varepsilon e_1/S \text{ si } n \in S$$

$$\texttt{signal } x \texttt{ default } v_1 \texttt{ gather } v_2 \texttt{ in } e/S \rightarrow_\varepsilon e[x \leftarrow n]/S[(v_1, v_2, (\mathit{false}, v_1), \emptyset)/n] \text{ si } n \notin \mathit{Dom}(S)$$

$$\texttt{do } v \texttt{ until } n(x) \texttt{ -> } e \texttt{ done}/S \rightarrow_\varepsilon v/S \qquad \texttt{do } v \texttt{ when } n/S \rightarrow_\varepsilon v/S \text{ si } n \in S$$

$$\texttt{pre } n/S \rightarrow_\varepsilon b/S \text{ si } S^p(n) = (b, v) \qquad \texttt{pre ?}n/S \rightarrow_\varepsilon v/S \text{ si } S^p(n) = (b, v)$$

FIG. 4.1 – Réduction en tête

`pre`. Initialement le multi-ensemble associé à n est vide car les signaux sont absents par défaut.
- Lorsque le corps d'une construction `do/until` est une valeur, cela signifie que la réaction est terminée. Donc le `do/until` retourne la valeur de son corps.
- Et le `do/when` peut rendre une valeur quand son corps est une valeur et le signal est présent.
- Les `pre` vont chercher dans l'environnement leur valeur.

Nous venons de définir les réductions en tête. Nous définissons maintenant la réduction en profondeur ($\rightarrow$) :

$$\frac{e/S \rightarrow_\varepsilon e'/S'}{\Gamma(e)/S \rightarrow \Gamma(e')/S'} \qquad \frac{n \in S \quad e/S \rightarrow e'/S'}{\Gamma(\texttt{do } e \texttt{ when } n)/S \rightarrow \Gamma(\texttt{do } e' \texttt{ when } n)/S'}$$

où Γ est un contexte d'évaluation. Dans la première règle, l'expression e se réduit en tête, donc elle peut se réduire dans n'importe quel contexte. La seconde règle définit la suspension. Elle montre que le corps d'un `do/when` ne peut être évalué que si le signal est présent.

Les contextes d'évaluation sont définis de la façon suivante :

$$\begin{array}{lll}
\Gamma & ::= & [\,] \mid \Gamma\ e \mid e\ \Gamma \mid (\Gamma, e) \mid (e, \Gamma) \mid \Gamma\texttt{;}e \mid \texttt{run } \Gamma \\
& & \mid \texttt{let } x \texttt{ = } \Gamma \texttt{ and } x \texttt{ = } e \texttt{ in } e \mid \texttt{let } x \texttt{ = } e \texttt{ and } x \texttt{ = } \Gamma \texttt{ in } e \\
& & \mid \texttt{emit } \Gamma\ e \mid \texttt{emit } e\ \Gamma \\
& & \mid \texttt{present } \Gamma \texttt{ then } e \texttt{ else } e \\
& & \mid \texttt{signal } x \texttt{ default } \Gamma \texttt{ gather } e \texttt{ in } e \mid \texttt{signal } x \texttt{ default } e \texttt{ gather } \Gamma \texttt{ in } e \\
& & \mid \texttt{do } e \texttt{ until } \Gamma(x) \texttt{ -> } e \texttt{ done} \mid \texttt{do } \Gamma \texttt{ until } n(x) \texttt{ -> } e \texttt{ done} \\
& & \mid \texttt{do } e \texttt{ when } \Gamma \\
& & \mid \texttt{pre } \Gamma \mid \texttt{pre ?}\Gamma
\end{array}$$

Si on reprend la définition de la composition parallèle, les deux contextes possibles sont $\Gamma \texttt{||} e$ et $e \texttt{||} \Gamma$. Ces contextes montrent que l'ordre d'évaluation n'est pas spécifié. Dans l'implantation de REACTIVEML, le choix de l'ordonnancement est fixé de sorte que l'exécution soit toujours

$$O \vdash v \rightarrow_{eoi} v \qquad O \vdash \texttt{pause} \rightarrow_{eoi} \texttt{()} \qquad \frac{O \vdash e_1 \rightarrow_{eoi} e_1'}{O \vdash e_1 \texttt{;} e_2 \rightarrow_{eoi} e_1' \texttt{;} e_2}$$

$$\frac{n \notin O}{O \vdash \texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow_{eoi} e_2}$$

$$\frac{O \vdash e_1 \rightarrow_{eoi} e_1' \quad O \vdash e_2 \rightarrow_{eoi} e_2'}{O \vdash \texttt{let}\ x_1\ \texttt{=}\ e_1\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ e \rightarrow_{eoi} \texttt{let}\ x_1\ \texttt{=}\ e_1'\ \texttt{and}\ x_2\ \texttt{=}\ e_2'\ \texttt{in}\ e}$$

$$\frac{O(n) = (\mathit{true}, v)}{O \vdash \texttt{do}\ e_1\ \texttt{until}\ n(x)\ \texttt{->}\ e_2\ \texttt{done} \rightarrow_{eoi} e_2[x \leftarrow v]}$$

$$\frac{n \notin O \quad O \vdash e_1 \rightarrow_{eoi} e_1'}{O \vdash \texttt{do}\ e_1\ \texttt{until}\ n(x)\ \texttt{->}\ e_2\ \texttt{done} \rightarrow_{eoi} \texttt{do}\ e_1'\ \texttt{until}\ n(x)\ \texttt{->}\ e_2\ \texttt{done}}$$

$$\frac{n \in O \quad O \vdash e \rightarrow_{eoi} e'}{O \vdash \texttt{do}\ e\ \texttt{when}\ n \rightarrow_{eoi} \texttt{do}\ e'\ \texttt{when}\ n} \qquad \frac{n \notin O}{O \vdash \texttt{do}\ e\ \texttt{when}\ n \rightarrow_{eoi} \texttt{do}\ e\ \texttt{when}\ n}$$

FIG. 4.2 – Réaction de fin d'instant

reproductible d'une exécution à l'autre. Nous reviendrons plus tard (chapitre 10) sur l'ordre d'exécution du parallèle et les choix faits dans les autres langages.

On note également que `do` Γ `when` n n'est pas un contexte car on ne veut pas pouvoir évaluer dans un `do/when` lorsque le signal n est absent.

4.1.2 Réaction de fin d'instant

Le modèle réactif est basé sur l'absence de réaction instantanée à l'absence d'un signal. Il faut donc attendre la fin d'instant pour traiter l'absence des signaux et préparer le programme pour l'instant suivant.

La réaction de l'instant s'arrête lorsqu'il n'y a plus de réductions $\rightarrow$ possibles. À partir de ce moment, l'environnement des signaux est figé, il ne peut plus y avoir de nouvelles émissions. Les signaux qui n'ont pas été émis sont supposés absents. On peut alors calculer la sortie O du programme avec la fonction *next* définie dans le chapitre 3.3.2 (page 51).

Les règles pour le traitement de fin d'instant ont la forme suivante : $O \vdash e \rightarrow_{eoi} e'$. Leur définition est donnée figure 4.2. Nous pouvons remarquer que les règles ne sont données que pour un sous-ensemble d'expressions car elles seront appliquées seulement quand l'expression e ne peut plus être réduite par $\rightarrow$. Nous verrons chapitre 5 que les expressions biens typées se réduisent toujours dans une expression de cette forme.

Commentons les règles de la figure 4.2 :

– Les valeurs restent inchangées par la réduction.

- L'instruction `pause` peut se réduire uniquement à la fin de l'instant.
- Le traitement de fin d'instant est appliqué seulement à la blanche gauche de la séquence car la branche droite n'est pas activée pendant l'instant.
- De même, la réduction de fin d'instant ne s'applique qu'aux deux expressions de la partie gauche du `let/and/in`.
- S'il y a une instruction `present` à la fin d'instant, c'est que le signal est absent. Donc à l'instant suivant il faudra exécuter la branche `else`. La notation $n \notin O$ indique que le statut associé à n dans O est *false*.
- Le traitement de la préemption a lieu en fin d'instant. Si le signal qui contrôle le `do/until` est présent, l'expression doit être préemptée. Dans ce cas, le code de traitement de la préemption (e_2) doit être exécuté à l'instant suivant avec x qui prend la valeur du signal.
- Et pour le `do/when`, si le signal est présent, il faut propager le traitement de fin d'instant. Si le signal est absent, le corps du `do/when` est suspendu.

4.1.3 Exécution d'un programme

L'exécution d'un instant est définie par la relation :

$$e_i/S_i \Rightarrow e'_i/S'_i$$

Si on note I_i et O_i les entrées et les sorties de la réaction de l'instant, l'environnement des signaux doit avoir les propriétés suivantes. Tous les signaux sont par défaut absents sauf les signaux donnés en entrée ($I_i = S_i^v$). Les sorties sont calculées à partir de l'environnement à la fin de la réaction ($O_i = next(S'_i)$). Les valeurs par défaut, les fonctions de combinaison des signaux sont conservées d'un instant à l'autre ($S'^d_i = S^d_{i+1}$ et $S'^g_i = S^g_{i+1}$). Les sorties définissent le `pre` de l'instant suivant ($O_i = S^p_{i+1}$).

L'exécution d'un instant se décompose en deux étapes : la réduction de e_i jusqu'à l'obtention d'un point fixe e''_i, puis la préparation à l'instant suivant.

$$\frac{e_i/S_i \hookrightarrow e''_i/S'_i \quad O_i = next(S'_i) \quad O_i \vdash e''_i \rightarrow_{eoi} e'_i}{e_i/S_i \Rightarrow e'_i/S'_i}$$

où $e/S \hookrightarrow e'/S'$ si $e/S \rightarrow^* e'/S'$ et $e'/S' \not\rightarrow$. La relation $\rightarrow^*$ est la fermeture réflexive et transitive de $\rightarrow$.

4.2 Équivalence avec la sémantique comportementale

Nous montrons que la sémantique opérationnelle est équivalente à la sémantique comportementale (chapitre 3). Cette propriété permet de choisir le formalisme le plus adapté en fonction des propriétés à montrer.

Nous commençons par prouver que si une expression e réagit en une expression e' avec la sémantique à petits pas, alors elle peut réagir dans le même environnement de signaux avec la sémantique grands pas.

Théorème 1
Pour tout environnement S_{init} et expression e tels que $e/S_{init} \Rightarrow e'/S$, alors il existe N, E et b tels que $N \vdash e \xrightarrow[S]{E,b} e'$ avec $E = S^v \backslash S^v_{init}$.

$$\begin{tikzcd}
e/S \arrow[r] \arrow[d, equal] & \cdots \arrow[r] & e_{n-2}/S_{n-2} \arrow[r] \arrow[d, equal] & e_{n-1}/S_{n-1} \arrow[r] \arrow[d, equal] & e_n/S' \arrow[r, "eoi"'] \arrow[d, equal] \arrow[dr, phantom, "A"] & e' \arrow[d, equal] \\
e/S \arrow[r] \arrow[d, equal] & \cdots \arrow[r] & e_{n-2}/S_{n-2} \arrow[r] \arrow[d, equal] & e_{n-1}/S_{n-1} \arrow[r] \arrow[d, equal] \arrow[dr, phantom, "B"] & e_n/S' \arrow[r, dashed, "\emptyset{,}b", "S'"'] & e' \arrow[d, equal] \\
e/S \arrow[r] \arrow[d, equal] & \cdots \arrow[r] & e_{n-2}/S_{n-2} \arrow[r] & e_{n-1}/S_{n-1} \arrow[rr, dashed, "E_1{,}b", "S'"'] & & e' \arrow[d, equal] \\
\vdots \arrow[d, equal] & & & & & \vdots \arrow[d, equal] \\
e/S \arrow[rrrrr, dashed, "E{,}b", "S'"'] & & & & & e'
\end{tikzcd}$$

FIG. 4.3 – Structure de la preuve du théorème 1.

Démonstration :

La preuve se fait par induction sur le nombre de réductions $\rightarrow$ dans $e/S_{init} \Rightarrow e'/S$.

- S'il n'y a pas de réductions $\rightarrow$ possibles, il faut montrer que la réduction $\rightarrow_{eoi}$ est équivalente à la sémantique grands pas (lemme 2).
- S'il y a au moins une réduction $\rightarrow$, on doit montrer qu'une réduction $\rightarrow$ suivie d'une réaction grands pas est équivalente à une réaction grands pas (lemme 3).

Le diagramme de la figure 4.3 représente la structure de la preuve. Les flèches pleines sont quantifiées universellement ($\forall$) et les flèches hachurées le sont existentiellement ($\exists$). Les diagrammes A et B correspondent respectivement aux lemmes 2 et 3. □

La preuve de ce lemme est basée sur les propriétés suivantes :

Lemme 2
Si $e/S \not\rightarrow$ et $S \vdash e \rightarrow_{eoi} e'$, alors il existe N et b tel que $N \vdash e \xrightarrow[S]{\emptyset, b} e'$.

Démonstration :

La preuve se fait par induction sur la structure de e. □

Lemme 3
Si $e/S_0 \rightarrow e_1/S_1$ et $N \vdash e_1 \xrightarrow[S]{E', b} e'$ avec $S_1 \sqsubseteq S$
alors $N \vdash e \xrightarrow[S]{E, b} e'$ avec $E = E' \sqcup (S_1^v \backslash S_0^v)$.

Démonstration :

La preuve se fait en deux étapes. On montre d'abord la même propriété pour la réduction $\rightarrow_\varepsilon$. Puis on montre que cette propriété est vraie dans tout contexte Γ. □

On montre maintenant que si un programme peut réagir dans le même environnement avec les deux sémantiques, alors les expressions obtenues à la fin des réactions sont les mêmes. On a donc l'équivalence entre les deux sémantiques.

Théorème 2
Pour chaque S_{init} et e tels que :

- $N \vdash e \xrightarrow[S_1]{E_1, b_1} e_1$ *où* S_1 *est le plus petit environnement tel que* $S_{init} \sqsubseteq S_1$
- $e/S_{init} \Rightarrow e_2/S_2$
- $\forall n \in Dom(S_2) : S_2^g(n) = f$ *et* $f(x, f(y, z)) = f(y, f(x, z))$,

alors $e_1 = e_2$ *et* $S_1 = S_2$

Démonstration :

Avec le théorème 1, il existe E_2 et b_2 tels que $N \vdash e \xrightarrow[S_2]{E_2, b_2} e_2$ et on peut remarquer, par construction, que S_2 est le plus petit environnement tel que $S_{init} \sqsubseteq S_2$.
Avec la propriété d'unicité (propriété 2), nous savons qu'il existe un unique plus petit environnement de signaux dans lequel une expression peut réagir avec la sémantique comportementale donc $S_1 = S_2$. Maintenant, avec le déterminisme de la sémantique (propriété 1) on a $E_1 = E_2$, $b_1 = b_2$ et $e_1 = e_2$.

□

4.3 Ajout des constructions non déterministes

Afin de donner la sémantique complète du langage, nous ajoutons maintenant à la sémantique les références et la possibilité de récupérer instantanément une des valeurs émises sur un signal.

Ces deux constructions font perdre le déterminisme des réactions. Par exemple, le programme :

```
let x = ref 0 in
(x := 1 || x := 2);
print_int !x
```

peut afficher `1` ou `2`. L'indéterminisme vient de l'absence de spécification de l'ordre d'évaluation des branches parallèles. De même, dans le programme suivant :

```
signal s in
emit s 1; emit s 2;
await one s(x) in print_int x
```

comme l'environnement des signaux n'est pas ordonné, `await/one` ne peut donner aucune garantie sur la valeur que va prendre `x`. Le programme peut donc afficher lui aussi `1` ou `2`.

On veut pouvoir refléter ces différentes exécutions possibles dans la sémantique.

Remarque :

Il y a le même problème en OCAML avec les paires par exemple :

```
let x = ref 0 in (x := 1, x := 2)
```

L'ordre d'évaluation n'est pas spécifié, mais l'exécution est déterministe. Si on exécute deux fois le même programme, il donne le même résultat.

4.3.1 Await/one

Nous avons vu chapitre au 2 qu'il existe deux formes de `await/one`. Une seule des deux est primitive. On a

$$\texttt{await one } s(x) \texttt{ in } e \stackrel{def}{=} \texttt{await immediate one } s(x) \texttt{ in pause;} e$$

On ajoute donc la construction `await immediate one` $e(x)$ `in` e aux expressions du noyau.

$$\begin{array}{lcl} e & ::= & \ldots \\ & & \mid \texttt{await immediate one } e(x) \texttt{ in } e \end{array}$$

Afin de décrire le comportement de cette construction, il faut ajouter la règle de réduction en tête de terme suivante :

$$\texttt{await immediate one } n(x) \texttt{ in } e/S \rightarrow_{\varepsilon} e[x \leftarrow one(S^v(n))]/S \quad \text{si} \quad n \in S$$

où *one* est une fonction qui retourne une valeur d'un multi-ensemble. On est sûr que le multi-ensemble n'est pas vide car on a la condition $n \in S$.

Il faut également étendre les contextes :

$$\begin{array}{lcl} \Gamma & ::= & \ldots \\ & & \mid \texttt{await immediate one } \Gamma(x) \texttt{ in } e \end{array}$$

Enfin, la règle de fin d'instant peut être appliquée à `await/one` si le signal est absent. Dans ce cas, il faut continuer à attendre que le signal soit émis.

$$\frac{n \notin O}{O \vdash \texttt{await immediate one } n(x) \texttt{ in } e \rightarrow_{eoi} \texttt{await immediate one } n(x) \texttt{ in } e}$$

On peut aussi facilement étendre la sémantique comportementale du chapitre 3 avec cette construction. Il suffit d'ajouter les règles suivantes au système de transition :

$$\frac{N \vdash e \xrightarrow[S]{E, true} n \qquad n \notin S}{N \vdash \texttt{await immediate one } e(x) \texttt{ in } e_1 \xrightarrow[S]{E, false} \texttt{await immediate one } n(x) \texttt{ in } e_1}$$

$$\frac{N \vdash e \xrightarrow[S]{E, true} n \qquad n \in S \qquad N_1 \vdash e_1[x \leftarrow one(S^v(n))] \xrightarrow[S]{E_1, b} e_1'}{N \cdot N_1 \vdash \texttt{await immediate one } e(x) \texttt{ in } e_1 \xrightarrow[S]{E \sqcup E_1, b} e_1'}$$

On peut remarquer que la sémantique opérationnelle est plus précise que la sémantique comportementale. Dans l'exemple suivant, avec la sémantique opérationnelle la seule valeur possible pour `x` est `1` alors qu'avec la sémantique comportementale `x` peut aussi être égal à `2` :

```
signal s in
emit s 1;
await immediate one s(x) in
emit s 2;
print_int x
```

Enfin, les règles de bonne formation sont étendues par :

$$\frac{0 \vdash e \qquad 1 \vdash e_1}{1 \vdash \texttt{await immediate one } e(x) \texttt{ in } e_1}$$

La règle de bonne formation indique que la construction `await/one` ne peut être utilisée que dans un contexte de processus car son exécution peut prendre plusieurs instants si le signal est absent.

4.3.2 Les références

L'extension du noyau de ReactiveML avec les références se fait comme pour l'extension de ML. On introduit la notion d'adresse mémoire (l) et d'état mémoire (M). Un état mémoire M est une fonction des adresses mémoires dans les valeurs.

$$M ::= [v_1/l_1, ..., v_n/l_n]$$

On ajoute au noyau la définition de références ($\texttt{ref}\, e$), l'affectation ($e \texttt{ :=} e$), le déréférencement ($\texttt{!}e$) et les adresses mémoires (l) :

$$e \quad ::= \quad \ldots \mid \texttt{ref}\, e \mid e \texttt{ :=} e \mid \texttt{!}e \mid l$$

Les adresses mémoires sont également ajoutées à l'ensemble des valeurs.

La sémantique opérationnelle doit maintenant prendre en compte l'état mémoire. La réaction d'un instant devient donc :

$$\frac{e_i/S_i, M_i \hookrightarrow e''_i/S'_i, M'_i \quad O_i = next(S'_i) \quad O_i \vdash e''_i \rightarrow_{eoi} e'_i}{e_i/S_i, M_i \Rightarrow e'_i/S'_i, M'_i}$$

avec les relations $\rightarrow$ et $\rightarrow_\varepsilon$ étendues avec les états mémoires.

Les règles de réduction des nouvelles expressions sont :

$$\texttt{ref}\, v/S, M \rightarrow_\varepsilon l/S, M[v/l] \ \text{ si } \ l \notin Dom(M) \qquad \texttt{!}l/S, M \rightarrow_\varepsilon M(l)/S, M \ \text{ si } \ l \in Dom(M)$$

$$l \texttt{ :=} v/S, M \rightarrow_\varepsilon \texttt{()}/S, M[v/l] \ \text{ si } \ l \in Dom(M)$$

Toutes les règles du noyau laissent la mémoire inchangée, seul les nouvelles règles y accèdent.

Pour finir de décrire la sémantique, on augmente les contextes :

$$\Gamma \quad ::= \quad \ldots \mid \texttt{ref}\, \Gamma \mid \Gamma \texttt{ :=} e \mid e \texttt{ :=} \Gamma \mid \texttt{!}\Gamma$$

Sémantique comportementale

Nous ne pouvons pas donner la sémantique comportementale du langage avec références car l'entrelacement des branches parallèles ne peut pas être décrit. Le problème s'illustre si on essaye de donner la règle du parallèle.

$$\frac{N_1 \vdash e_1/M \xrightarrow[S]{E_1, b_1} e'_1/M_2 \quad N_2 \vdash e_2/M \xrightarrow[S]{E_2, b_2} e'_2/M_2 \quad b_1 \wedge b_2 = false}{N_1 \cdot N_2 \vdash e_1 \,||\, e_2/M \xrightarrow[S]{E_1 \sqcup E_2, false} e'_1 \,||\, e'_2/(M_1 \uplus M_2)}$$

On voit que la réaction de e_1 ne dépend pas de M_2 et réciproquement, e_2 ne dépend pas de M_1. De plus, comment définir $\uplus$ si e_1 et e_2 modifient la même référence ?

Une solution peut être d'interdire les lectures et écritures sur une partie commune de la mémoire par les deux branches parallèles. Le problème de cette approche est qu'elle réduit l'expressivité du langage. De plus, il est difficile de faire une analyse statique qui garantit cette propriété et qui soit à la fois modulaire et suffisamment fine pour le pas rejeter trop de programmes.

On voit ici les limites de la sémantique à grands pas. Comme elle abstrait l'ordonnancement à l'intérieur de l'instant, elle ne peut pas exprimer un dialogue instantané par mémoire partagée.

4.4 Conclusion

Ce chapitre introduit la sémantique opérationnelle de ReactiveML. C'est une sémantique à petits pas décrivant la réaction d'un instant comme une succession de micro-réactions. Même si cette sémantique est à grains fins, elle ne spécifie pas l'ordonnancement des calculs. Ainsi cette sémantique offre un bon formalisme pour vérifier la correction des implantations. En effet, l'ordonnancement n'étant pas spécifié, tous les ordonnancements possibles peuvent être exprimés.

La preuve d'équivalence entre la sémantique opérationnelle et la sémantique comportementale a été réalisée. Nous disposons ainsi d'un cadre sémantique large avec deux formalisations du langage qui sont de nature différente.

Résumé

Nous avons introduit dans ce chapitre une sémantique à petits pas du langage qui permet de définir la construction de l'environnement des signaux au cours de la réaction. Cette sémantique est définie en deux étapes. La première s'inspire de la sémantique à réduction de ML. Elle décrit la réaction pendant l'instant comme une succession de micro-réactions jusqu'à l'obtention d'un point fixe. Une fois le point fixe obtenu, le programme ne peut plus émettre de signaux pendant l'instant. La seconde étape peut alors gérer l'absence des signaux pour préparer la réaction de l'instant suivant. Nous avons montré que cette sémantique est équivalente à la sémantique comportementale du chapitre précédent.

Enfin, nous avons étendu le langage avec la construction `await/one` et les références. Ces constructions font perdre le déterminisme de la sémantique. En voulant conserver le parallèle associatif et commutatif, il ne peut pas y avoir de solution unique pour l'évaluation du programme suivant :

```
let x = ref 0 in (x := 1 || x := 2)
```

Chapitre 5

Typage

Nous terminons la partie sur la sémantique de ReactiveML en présentant le système de type du langage ainsi que sa preuve de sûreté. Ce système de type est une extension conservative du système de type de ML, garantissant que tous les programmes ML bien typés restent bien typés et gardent le même type. Ce point est important dès lors que l'on souhaite plonger le modèle dans un langage existant.

5.1 Extension du typage de ML

Le système de type de ReactiveML est une extension du système de type de ML de Milner [73]. Nous devons traiter les signaux et en particulier les valeurs qui peuvent être transmises sur les signaux. Le langage de type est :

$$\begin{array}{lcl} \sigma & ::= & \forall \alpha_1, \ldots, \alpha_n . \tau \\ \tau & ::= & T \mid \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \tau\, \texttt{process} \mid (\tau, \tau)\, \texttt{event} \\ T & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots \end{array}$$

Un type (τ) peut être un type de base (T), une variable de type (α), le type d'une fonction ($\tau_1 \to \tau_2$), un type produit ($\tau_1 \times \tau_2$), le type d'un processus paramétré par sa valeur de retour ($\tau\, \texttt{process}$) ou le type d'un signal ($(\tau_1, \tau_2)\, \texttt{event}$). Dans le type $(\tau_1, \tau_2)\, \texttt{event}$, τ_1 est le type des valeurs émises sur le signal alors que τ_2 désigne le type de la valeur associée au signal en lecture (obtenu après la combinaison de toutes les valeurs émises pendant un instant). σ est un schéma de type.

Comme pour le typage des références, il ne faut pas généraliser le type des expressions qui créent des signaux. Par exemple, il ne faut pas généraliser le type de `x` dans l'expression suivante :

```
let x = signal s in s in
emit x 1; emit x true
```

Si on donnait le type $\forall \alpha. (\alpha, \alpha\, \texttt{multiset})\, \texttt{event}$ à `x`, on accepterait le programme qui pourtant est incorrect. Ainsi, les expressions *expansives* et les expressions *non-expansives* sont distinguées [103, 106]. Cette séparation est faite sur des critères syntaxiques. Les expressions non-expansives (e_{ne}) sont définies par :

$$\begin{array}{lcl} e_{ne} & ::= & x \mid c \mid (e_{ne}, e_{ne}) \mid \lambda x.e \mid \texttt{process}\ e \\ & & \mid \texttt{let}\ x\ \texttt{=}\ e_{ne}\ \texttt{and}\ x\ \texttt{=}\ e_{ne}\ \texttt{in}\ e_{ne} \mid \texttt{present}\ e_{ne}\ \texttt{then}\ e_{ne}\ \texttt{else}\ e_{ne} \\ & & \mid \texttt{emit}\ e_{ne}\ e_{ne} \mid \texttt{pause} \mid \texttt{pre}\ e_{ne} \mid \texttt{pre ?}e_{ne} \\ & & \mid \texttt{do}\ e_{ne}\ \texttt{until}\ e_{ne}(x)\ \texttt{->}\ e_{ne}\ \texttt{done} \mid \texttt{do}\ e_{ne}\ \texttt{when}\ e_{ne} \end{array}$$

Nous remarquons que les déclarations de signaux et les applications ne font pas partie des expressions non-expansives.

Un environnement de typage H associe un schéma de type aux variables et des types aux noms de signaux :

$$H \quad ::= \quad [x_1 : \sigma_1; \ldots ; x_k : \sigma_k; n_1 : \tau_1; \ldots ; n_{k'} : \tau_{k'}]$$

L'instanciation et la généralisation sont définies comme suit :

$$\tau'[\tau_1/\alpha_1, \ldots , \tau_n/\alpha_n] \leq \forall\alpha_1, \ldots , \alpha_k.\tau$$

$$Gen(H, e, \tau) = \begin{cases} \forall\alpha_1, ..., \alpha_n.\tau \text{ où } \{\alpha_1, ..., \alpha_k\} = fv(\tau) - fv(H) \text{ si } e \text{ non-expansive} \\ \tau \text{ sinon} \end{cases}$$

Les expressions sont typées dans un environnement de typage initial TC tel que :

$$TC = [\texttt{true} : \texttt{bool}; \texttt{fst} : \forall\alpha, \beta.\ \alpha \times \beta \rightarrow \alpha; ...]$$

Les expressions sont typées par un prédicat de la forme $H \vdash e : \tau$ qui se lit "e a le type τ dans l'environnement de typage H". Ce prédicat est défini figure 5.1.

- Les règles de typage des expressions ML ne sont pas modifiées.
- Pour le typage de `let/and/in`, le type des variables qui sont introduites est généralisé uniquement si les expressions e_1 et e_2 sont non-expansives.
- Dans le typage de `signal`, la valeur par défaut (e_1) a le type de la valeur combinée et la fonction de combinaison (e_2) est une fonction qui prend en paramètre une valeur émise et la combinaison des valeurs déjà émises et retourne la nouvelle combinaison.
- La règle pour `emit` vérifie que le premier argument est de type $(\tau_1, \tau_2)\,\texttt{event}$ (le type des signaux) et que le type de la valeur émise est τ_1.
- `do` e_1 `until` $e(x)$ `->` e_2 `done` récupère la valeur associée à un signal en cas de préemption. Ainsi, si e a le type $(\tau_1, \tau_2)\,\texttt{event}$, x doit avoir le type τ_2.
- L'instanciation `run` e est appliquée à un processus.
- Les constructions `present`, `until` et `when` peuvent être appliquées à n'importe quel signal.

Nous définissons maintenant la notion d'environnement de signaux bien typé. Elle est notée $H \vdash S$. Un environnement S est bien typé dans un environnement H si les environnements S et H contiennent les mêmes signaux :

$$n \in Dom(S) \Leftrightarrow n \in Dom(H)$$

et que pour tout signal $n \in Dom(S)$, il existe un type $(\tau_1, \tau_2)\,\texttt{event}$ tel que :

$$\begin{array}{l} H(n) = (\tau_1, \tau_2)\,\texttt{event} \\ H \vdash S^d(n) : \tau_2 \\ H \vdash S^g(n) : \tau_1 \rightarrow \tau_2 \rightarrow \tau_2 \\ H \vdash S^p(n) : \texttt{bool} \times \tau_2 \\ H \vdash S^m(n) : \tau_1\,\texttt{multiset} \end{array}$$

De même, pour l'environnement de fin d'instant, nous définissons le jugement $H \vdash O$. Donc O est bien typé dans H si pour tout signal n :

$$\begin{array}{l} n \in Dom(O) \Leftrightarrow n \in Dom(H) \\ H(n) = (\tau_1, \tau_2)\,\texttt{event} \\ H \vdash O(n) : \texttt{bool} \times \tau_2 \end{array}$$

$$\frac{\tau \leq H(x)}{H \vdash x : \tau} \qquad \frac{\tau \leq TC(c)}{H \vdash c : \tau} \qquad \frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{H[x : \tau_1] \vdash e : \tau_2}{H \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$\frac{H \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad H \vdash e_2 : \tau_1}{H \vdash e_1\, e_2 : \tau_2} \qquad \frac{H[x : \tau] \vdash e : \tau}{H \vdash \texttt{rec}\, x \texttt{=} e : \tau}$$

$$\frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2 \quad H[x_1 : Gen(H, (e_1, e_2), \tau_1); x_2 : Gen(H, (e_1, e_2), \tau_2)] \vdash e : \tau}{H \vdash \texttt{let}\ x_1\ \texttt{=}\ e_1\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ e : \tau}$$

$$\frac{H \vdash e : \tau}{H \vdash \texttt{process}\ e : \tau\, \texttt{process}} \qquad \frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash e_1 \texttt{;} e_2 : \tau_2}$$

$$\frac{H \vdash e : \tau\, \texttt{process}}{H \vdash \texttt{run}\ e : \tau} \qquad \frac{H \vdash e_1 : (\tau_1, \tau_2)\, \texttt{event} \quad H \vdash e_2 : \tau_1}{H \vdash \texttt{emit}\ e_1\ e_2 : \texttt{unit}}$$

$$\frac{H \vdash e : (\tau_1, \tau_2)\, \texttt{event} \quad H \vdash e_1 : \tau \quad H \vdash e_2 : \tau}{H \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\frac{H \vdash e_1 : \tau_2 \quad H \vdash e_2 : \tau_1 \rightarrow \tau_2 \rightarrow \tau_2 \quad H[s : (\tau_1, \tau_2)\, \texttt{event}] \vdash e : \tau}{H \vdash \texttt{signal}\ s\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e : \tau}$$

$$\frac{H \vdash e : (\tau_1, \tau_2)\, \texttt{event} \quad H \vdash e_1 : \tau \quad H[x : \tau_2] \vdash e_2 : \tau}{H \vdash \texttt{do}\ e_1\ \texttt{until}\ e(x)\ \texttt{->}\ e_2\ \texttt{done} : \tau}$$

$$\frac{H \vdash e_1 : (\tau_1, \tau_2)\, \texttt{event} \quad H \vdash e : \tau}{H \vdash \texttt{do}\ e\ \texttt{when}\ e_1 : \tau} \qquad \frac{}{H \vdash \texttt{pause} : \texttt{unit}}$$

$$\frac{H \vdash e : (\tau_1, \tau_2)\, \texttt{event}}{H \vdash \texttt{pre}\ e : \texttt{bool}} \qquad \frac{H \vdash e : (\tau_1, \tau_2)\, \texttt{event}}{H \vdash \texttt{pre ?}e : \tau_2}$$

FIG. 5.1 – Système de type

Nous avons les propriétés suivantes sur les environnements de signaux. Si pour tout H, S et n tels que $O = next(S)$ alors

$$H \vdash S \text{ implique } H \vdash O \quad \text{et} \quad n \in S \text{ implique } n \in O \quad \text{et} \quad n \notin S \text{ implique } n \notin O$$

5.2 Preuve de sûreté du typage pour la réduction $\rightarrow$

Les techniques classiques de preuve de sûreté du typage [87] sont utilisées. La principale différence vient de la définition des expressions en forme normale vis-à-vis de la réduction $\rightarrow$. Ces expressions ne sont pas des valeurs mais des expressions sur lesquelles nous pouvons appliquer la réduction $\rightarrow_{eoi}$. Elles sont appelées *expressions de fin d'instant*.

La présentation suit fidèlement les preuves de [62].

Théorème 3 (Sûreté du typage)
Si $TC \vdash e : \tau$ et $k \vdash e$ et $e/S \rightarrow^* e'/S'$ et e'/S' est en forme normale vis-à-vis de $\rightarrow$, alors e' est une expression de fin d'instant.

Démonstration :

La démonstration repose sur les deux propriétés suivantes :
- la préservation du typage par la réduction $\rightarrow$
- les formes normales bien typées sont des expressions de fin d'instant.

□

5.2.1 Préservation du typage

Pour définir la propriété de préservation du typage, nous commençons par définir la relation *être moins typable*.

Définition 5 (être moins typable)
Nous disons que e_1/S_1 est moins typable que e_2/S_2 ($e_1/S_1 \sqsubseteq e_2/S_2$) si pour tout environnement H et pour tout type τ :

- *Si e_1 est non-expansive alors e_2 est non-expansive et de plus :*

$$H \vdash e_1 : \tau \text{ et } H \vdash S_1 \quad \text{implique} \quad H \vdash e_2 : \tau \text{ et } H \vdash S_2$$

- *Si e_1 est expansive, alors il existe H' étendant H tel que :*

$$H \vdash e_1 : \tau \text{ et } H \vdash S_1 \quad \text{implique} \quad H' \vdash e_2 : \tau \text{ et } H' \vdash S_2$$

L'environnement H' étend H si pour tout $x \in Dom(H)$ et pour tout $n \in Dom(H)$, $H(x) = H'(x)$ et $H(n) = H'(n)$ et il peut exister des noms de signaux $n \in Dom(H')$ qui ne sont pas définis dans H ($n \notin Dom(H)$).

La préservation du typage par la réduction $\rightarrow$ montre que si une expression e est bien typée et elle se réduit en une expression e', alors e' est bien typée et a le même type.

Lemme 4 (Préservation du typage)
Si $e/S \rightarrow e'/S'$, alors $e/S \sqsubseteq e'/S'$

La démonstration utilise le lemme de substitution.

Lemme 5 (Lemme de substitution)
Supposons

$$H \vdash e' : \tau' \quad \text{et} \quad H[x : \forall \alpha_1, \ldots, \alpha_n.\tau'] \vdash e : \tau$$

avec $\alpha_1, \ldots, \alpha_n$ *non libres dans* H*, et aucune des variables liées dans* e *n'est libre dans* e'*. Alors,*

$$H \vdash e[x \leftarrow e'] : \tau$$

Démonstration :

La démonstration de ce lemme est comme pour ML. Elle se fait par récurrence sur la structure des expressions. Pour les expressions qui n'introduisent pas d'identificateur, la preuve se fait directement avec l'hypothèse de récurrence. Pour les constructions `signal/in` et `do/until`, elle utilise le même principe que pour la définition de fonctions. Et le cas `let/and/in` se prouve comme un `let/in`. □

Maintenant, pour prouver la préservation du typage, nous traitons les cas de base en démontrant la préservation du typage pour les réductions en tête de terme.

Lemme 6 (Préservation du typage par réduction de tête)
Si $e/S \rightarrow_\varepsilon e'/S'$*, alors* $e/S \sqsubseteq e'/S'$

Démonstration :

La propriété e non-expansive implique e' non-expansive se montre directement par examen des règles de réduction. Le reste de la démonstration se fait également par cas sur la règle de réduction utilisée.

Cas `run` $($`process` $e)/S \rightarrow_\varepsilon e/S$ **:** Par hypothèse, $H \vdash$ `run` $($`process` $e) : \tau$ et $H \vdash S$. La dérivation de typage de l'expression `run` $($`process` $e)$ est de la forme :

$$\dfrac{\dfrac{H \vdash e : \tau}{H \vdash (\texttt{process } e) : \tau\, \texttt{process}}}{H \vdash \texttt{run } (\texttt{process } e) : \tau}$$

Nous avons donc $H \vdash e : \tau$. Par conséquent, `run` $($`process` $e)/S \sqsubseteq e/S$.

Cas `emit` $n\ v/S \rightarrow_\varepsilon$ `()`$/S + [v/n]$ **:** Puisque `emit` $n\ v$ est bien typée dans H, nous avons la dérivation :

$$\dfrac{H \vdash n : (\tau_1, \tau_2)\, \texttt{event} \quad H \vdash v : \tau_1}{H \vdash \texttt{emit } n\ v : \texttt{unit}}$$

Par ailleurs, $H \vdash$ `()` : `unit`.
Nous avons également $H \vdash S$. Comme le typage de `emit` nous a donné $H \vdash n : (\tau_1, \tau_2)$ `event` et $H \vdash v : \tau_1$, nous avons :

$$H \vdash S + [v/n]$$

Donc `emit` $n\ v/S \sqsubseteq$ `()`$/S + [v/n]$.

Cas `signal` x `default` v_1 `gather` v_2 `in` $e/S \rightarrow_\varepsilon e[x \leftarrow n]/S[(v_1, v_2, (\mathit{false}, v_1), \emptyset)/n]$ **:** La dérivation de `signal/in` est de la forme :

$$\dfrac{H \vdash v_1 : \tau_2 \quad H \vdash v_2 : \tau_1 \rightarrow \tau_2 \rightarrow \tau_2 \quad H[x : (\tau_1, \tau_2)\, \texttt{event}] \vdash e : \tau}{H \vdash \texttt{signal } x \texttt{ default } v_1 \texttt{ gather } v_2 \texttt{ in } e : \tau}$$

Comme $H \vdash S$ et $n \notin Dom(S)$, nous avons $n \notin Dom(H)$. Prenons alors l'environnement $H' = H[n : (\tau_1, \tau_2)\,\texttt{event}]$. En utilisant l'indifférence du typage vis-à-vis des hypothèses inutiles, nous avons $H'[x : (\tau_1, \tau_2)\,\texttt{event}] \vdash e : \tau$. Par ailleurs, $H' \vdash n : (\tau_1, \tau_2)\,\texttt{event}$. Donc avec le lemme de substitution, nous avons :

$$H' \vdash e[x \leftarrow n] : (\tau_1, \tau_2)\,\texttt{event}$$

Nous pouvons montrer simplement que $H \vdash S[(v_1, v_2, (\mathit{false}, v_1), \emptyset)/n]$. Enfin comme l'expression $\texttt{signal}\ x\ \texttt{default}\ v_1\ \texttt{gather}\ v_2\ \texttt{in}\ e$ est expansive, nous avons :

$$\texttt{signal}\ x\ \texttt{default}\ v_1\ \texttt{gather}\ v_2\ \texttt{in}\ e/S \sqsubseteq e[x \leftarrow n]/S[(v_1, v_2, (\mathit{false}, v_1), \emptyset)/n]$$

Pour les autres cas, ou bien ce sont des cas de ML, ou bien ils se prouvent simplement comme le cas `run`.

□

La démonstration de la préservation du typage se termine en passant au contexte avec la propriété de croissance de $\sqsubseteq$.

Lemme 7 (Croissance de $\sqsubseteq$)
Pour tout contexte,

$$e_1/S_1 \sqsubseteq e_2/S_2 \quad \textit{implique} \quad \Gamma[e_1]/S_1 \sqsubseteq \Gamma[e_2]/S_2$$

Démonstration :

Soient H et τ tels que $H \vdash \Gamma[e] : \tau$ et $H \vdash S$. Montrons que $H' \vdash \Gamma[e'] : \tau$ et $H' \vdash S'$ par récurrence sur la structure du contexte Γ. Le cas le plus intéressant est celui du `let/and/in`.

Cas $\texttt{let}\ x_1\ \texttt{=}\ \Gamma[e_1]\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ e$ **:** Une dérivation de typage de cette expression se termine par :

$$\frac{\begin{array}{c} H \vdash \Gamma[e_1] : \tau_1 \quad H \vdash e_2 : \tau_2 \\ H[x_1 : Gen(H, (\Gamma[e_1], e_2), \tau_1); x_2 : Gen(H, (\Gamma[e_1], e_2), \tau_2)] \vdash e : \tau \end{array}}{H \vdash \texttt{let}\ x_1\ \texttt{=}\ \Gamma[e_1]\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ e : \tau}$$

Par l'hypothèse de récurrence, nous avons $e_1/S \sqsubseteq e_1'/S'$.
Si $\Gamma[e_1]$ est non-expansive, alors e_1 est non-expansive et donc e_1' est non-expansive. Ainsi par définition de la relation $\sqsubseteq$, nous avons $H \vdash \Gamma[e_1'] : \tau_1$ et $H \vdash S'$ et $\Gamma[e_1']$ est non-expansive. Par conséquent, nous pouvons construire la dérivation :

$$\frac{\begin{array}{c} H \vdash \Gamma[e_1'] : \tau_1 \quad H \vdash e_2 : \tau_2 \\ H[x_1 : Gen(H, (\Gamma[e_1'], e_2), \tau_1); x_2 : Gen(H, (\Gamma[e_1'], e_2), \tau_2)] \vdash e : \tau \end{array}}{H \vdash \texttt{let}\ x_1\ \texttt{=}\ \Gamma[e_1']\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ e : \tau}$$

Donc nous avons dans le cas où e_1 n'est pas expansive :

$$\texttt{let}\ x_1\ \texttt{=}\ \Gamma[e_1]\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ e/S \sqsubseteq \texttt{let}\ x_1\ \texttt{=}\ \Gamma[e_1']\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ e/S'$$

Maintenant, si e_1 est expansive, l'hypothèse de récurrence nous donne un environnement H' étendant H tel que $H' \vdash \Gamma[e_1'] : \tau_1$ et $H' \vdash S'$. De plus comme e_1 est expansive, nous avons $Gen(H, (\Gamma[e_1], e_2), \tau_1) = \tau_1$ et $Gen(H, (\Gamma[e_1], e_2), \tau_2) = \tau_2$. Ainsi en utilisant l'indifférence du typage vis-à-vis des hypothèses inutiles, nous avons la dérivation :

$$\frac{H' \vdash \Gamma[e_1'] : \tau_1 \quad H' \vdash e_2 : \tau_2 \quad H'[x_1 : \tau_1; x_2 : \tau_2] \vdash e : \tau}{H' \vdash \texttt{let } x_1 \texttt{ = } \Gamma[e_1'] \texttt{ and } x_2 \texttt{ = } e_2 \texttt{ in } e : \tau}$$

Donc, dans le cas où e_1 est expansive, nous avons également :

$$\texttt{let } x_1 \texttt{ = } \Gamma[e_1] \texttt{ and } x_2 \texttt{ = } e_2 \texttt{ in } e/S \sqsubseteq \texttt{let } x_1 \texttt{ = } \Gamma[e_1'] \texttt{ and } x_2 \texttt{ = } e_2 \texttt{ in } e/S'$$

□

5.2.2 Les formes normales bien typées sont des expressions de fin d'instant

Pour démontrer que les formes normales vis-à-vis de la réduction → sont des expressions de fin d'instant, nous utilisons le lemme de progression. Il montre qu'une expression bien typée peut se réduire, sinon c'est une expression de fin d'instant.

Afin de simplifier la preuve de ce lemme, les expressions instantanées ($0 \vdash e$) sont traitées séparément.

Lemme 8 (Lemme de progression des expressions instantanées)
Soit H un environnement qui ne lie aucune variable x mais uniquement des noms de signaux n. Supposons $H \vdash e : \tau$, $H \vdash S$ et $0 \vdash e$. Alors, ou bien e est une valeur, ou bien il existe e' et S' tels que $e/S \rightarrow e'/S'$.

Démonstration :

Les expressions instantanées correspondent aux expressions ML. Par conséquent, la preuve est la même que pour ML. Elle se fait par récurrence sur la structure de e et par cas sur e sachant que nous n'avons pas à traiter les cas où l'expression e est réactive ($1 \vdash e$).

Cette démonstration utilise la propriété de *la forme des valeurs selon leur type* qui montre que le type d'une valeur conditionne sa structure (constante, fonction, paire, processus, ...).

□

Lemme 9 (Lemme de progression)
Soit H un environnement qui ne lie aucune variable x mais uniquement des noms de signaux n. Supposons $H \vdash e : \tau$, $H \vdash S$ et $k \vdash e$. Alors, ou bien e est une expression de fin d'instant, ou bien il existe e' et S' tels que $e/S \rightarrow e'/S'$.

Démonstration :

La démonstration se fait par récurrence sur la structure de e et par cas sur e.

Cas $0 \vdash e$: Avec le lemme précédent de progression des expressions instantanées, nous savons que soit e peut se réduire, soit e est une valeur. Donc si c'est une valeur c'est également une expression de fin d'instant car nous pouvons appliquer la réduction :

$$S \vdash v \rightarrow_{eoi} v$$

Cas `pause` : Cette expression est une expression de fin d'instant car nous pouvons appliquer la règle :

$$O \vdash \texttt{pause} \rightarrow_{eoi} \texttt{()}$$

Cas `run` e : On a alors :

$$\frac{H \vdash e : \tau\,\texttt{process}}{H \vdash \texttt{run}\ e : \tau} \quad \text{et} \quad \frac{0 \vdash e}{1 \vdash \texttt{run}\ e}$$

Si e n'est pas une valeur, nous pouvons appliquer le lemme précédent. Ainsi nous savons que e peut se réduire et en appliquant la règle de contexte, `run` e peut se réduire.

Si e est une valeur, son type garantit que c'est un processus. Donc l'expression e est de la forme `process` e'. La règle de réduction du `run` s'applique :

$$\texttt{run}\ (\texttt{process}\ e')/S \rightarrow_{\varepsilon} e'/S$$

Cas `present` e `then` e_1 `else` e_2 : On a alors :

$$\frac{H \vdash e : (\tau_1, \tau_2)\,\texttt{event} \quad H \vdash e_1 : \tau \quad H \vdash e_2 : \tau}{H \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau} \quad \text{et} \quad \frac{0 \vdash e \quad 1 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

Si e n'est pas une valeur, en appliquant le lemme de réduction des expressions instantanées, il vient que e peut se réduire. Donc avec la règle de contexte, `present` e `then` e_1 `else` e_2 peut se réduire.

Si e est une valeur, comme c'est une expression de type (τ_1, τ_2) `event`, c'est nécessairement un signal n. Il y a alors deux cas :

– Soit $n \in S$, et nous pouvons appliquer la règle de réduction en tête de `present` :

$$\texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2/S \rightarrow_{\varepsilon} e_1/S$$

– Soit $n \notin S$, et dans ce cas si $O = next(S)$, nous savons que $n \notin 0$. Nous avons donc une expression de fin d'instant car la règle suivante s'applique :

$$\frac{n \notin O}{O \vdash \texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow_{eoi} e_2}$$

Cas `do` e_1 `when` e : Les hypothèses nous donnent :

$$\frac{H \vdash e : (\tau_1, \tau_2)\,\texttt{event} \quad H \vdash e_1 : \tau}{H \vdash \texttt{do}\ e_1\ \texttt{when}\ e : \tau} \quad \text{et} \quad \frac{0 \vdash e \quad 1 \vdash e_1}{1 \vdash \texttt{do}\ e_1\ \texttt{when}\ e}$$

Si e n'est pas une valeur, nous pouvons réduire l'expression `do` e_1 `when` e en utilisant le lemme précédent et la règle de contexte.

Si e est une valeur, c'est un signal n. Nous étudions les deux cas en fonction de la présence de n :

– Si $n \in S$, nous appliquons l'hypothèse de récurrence. Donc soit e_1 peut être réduit et dans ce cas en utilisant la règle de contexte du `when`, nous pouvons réduire l'expression `do` e_1 `when` e. Soit e_1 est une expression de fin d'instant et dans ce cas si e_1 est une valeur, nous appliquons la règle :

$$\texttt{do } v \texttt{ when } n/S \rightarrow_{\varepsilon} v/S$$

si e_1 n'est pas une valeur et $O = next(S)$, alors `do` e_1 `when` e est une expression de fin d'instant :

$$\frac{n \in O \quad O \vdash e_1 \rightarrow_{eoi} e_1'}{O \vdash \texttt{do } e_1 \texttt{ when } n \rightarrow_{eoi} \texttt{do } e_1' \texttt{ when } n}$$

– Si $n \notin S$ et $O = next(S)$, alors `do` e_1 `when` e est une expression de fin d'instant :

$$\frac{n \notin O}{O \vdash \texttt{do } e_1 \texttt{ when } n \rightarrow_{eoi} \texttt{do } e_1 \texttt{ when } n}$$

Pour les autres cas, nous pouvons utiliser des raisonnements similaires.

□

Nous pouvons terminer la preuve du théorème de sûreté du typage en énonçant la propriété sur les formes normales. Cette propriété est une conséquence directe du lemme de progression.

Lemme 10 (Les formes normales bien typées sont des expressions de fin d'instant)
Si $TC \vdash e : \tau$ et $k \vdash e$ et e est en forme normale vis-à-vis de $\rightarrow$, alors e est une expression de fin d'instant.

5.3 Préservation du typage pour la réduction $\rightarrow_{eoi}$

Comme nous l'avons vu chapitre 4.1.3 (page 68), la réaction complète d'un instant est définie par :

$$\frac{e/S \hookrightarrow e''/S' \quad O = next(S') \quad O \vdash e'' \rightarrow_{eoi} e'}{e/S \Rightarrow e'/S'}$$

Nous venons de montrer avec la preuve de sûreté du typage de $\rightarrow$ que si $e/S \hookrightarrow e''/S'$ alors il existe un environnement de typage H tel que $H \vdash S'$ et $H \vdash e'' : \tau$ et $O \vdash e'' \rightarrow_{eoi} e'$ avec $O = next(S')$. Par ailleurs, nous savons que si $H \vdash S'$ alors $H \vdash O$. Donc pour prouver que la réaction complète d'un instant préserve le typage, il nous faut démontrer le théorème suivant :

Théorème 4 (Préservation du typage par réduction $\rightarrow_{eoi}$)
Si $H \vdash S$ et $O = next(S)$ et $O \vdash e \rightarrow_{eoi} e'$ alors $e/S \sqsubseteq e'/S$.

Démonstration :

La preuve procède par récurrence sur la taille de la dérivation d'évaluation. On raisonne par cas sur l'expression e.

En examinant chacun des cas, nous pouvons montrer que si $O \vdash e \rightarrow_{eoi} e'$ et que e est non-expansive alors e' est non-expansive. Regardons maintenant ce qui ce passe au niveau des types.

Cas v **:** Par hypothèses, $H \vdash v : \tau$ et $O \vdash v \rightarrow_{eoi} v$. Donc nous avons directement $H \vdash v : \tau$.

Cas `pause` **:** Par hypothèses, $H \vdash$ `pause` : `unit` et $O \vdash$ `pause` $\rightarrow_{eoi}$ `()`. Par ailleurs $H \vdash$ `()` : `unit` ce qui est le résultat attendu.

Cas $e_1 ; e_2$ **:** Par hypothèses,

$$\frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash e_1 ; e_2 : \tau_2} \quad \text{et} \quad \frac{O \vdash e_1 \rightarrow_{eoi} e_1'}{O \vdash e_1 ; e_2 \rightarrow_{eoi} e_1' ; e_2}$$

En appliquant l'hypothèse de récurrence, nous avons $H \vdash e_1' : \tau_1$. Nous pouvons donc construire la dérivation :

$$\frac{H \vdash e_1' : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash e_1' ; e_2 : \tau_2}$$

Cas `present` n `then` e_1 `else` e_2 **:** Par hypothèses,

$$\frac{H \vdash n : (\tau, \tau')\,\texttt{event} \quad H \vdash e_1 : \tau \quad H \vdash e_2 : \tau}{H \vdash \texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\frac{n \notin O}{O \vdash \texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow_{eoi} e_2}$$

Donc nous avons directement $H \vdash e_2 : \tau$.

Cas `let` x_1 `=` e_1 `and` x_2 `=` e_2 `in` e **:** La preuve est directe en utilisant l'hypothèse de récurrence et le lemme de croissance de $\sqsubseteq$.

Cas `do` e_1 `until` $n(x)$ `->` e_2 `done` **:** Nous distinguons deux cas en fonction de la présence de n dans l'environnement :
- Si $n \notin S$, nous utilisons l'hypothèse de récurrence.
- Si $n \in S$, par hypothèses

$$\frac{H \vdash n : (\tau_1, \tau_2)\,\texttt{event} \quad H \vdash e_1 : \tau \quad H[x : \tau_2] \vdash e_2 : \tau}{H \vdash \texttt{do}\ e_1\ \texttt{until}\ n(x)\ \texttt{->}\ e_2\ \texttt{done} : \tau}$$

$$\frac{O(n) = (true, v)}{O \vdash \texttt{do}\ e_1\ \texttt{until}\ n(x)\ \texttt{->}\ e_2\ \texttt{done} \rightarrow_{eoi} e_2[x \leftarrow v]}$$

Avec $H \vdash n : (\tau_1, \tau_2)$ `event` et l'hypothèse $H \vdash O$, nous avons $H \vdash O(n)$: `bool` $\times\ \tau_2$ et donc $H \vdash v : \tau_2$. Maintenant avec $H[x : \tau_2] \vdash e_2 : \tau$, nous pouvons appliquer le lemme de substitution et donc obtenir :

$$H \vdash e_2[x \leftarrow v] : \tau$$

Cas `do` e `when` n **:** Il suffit d'appliquer l'hypothèse de récurrence.

□

Résumé

Dans ce chapitre, nous avons présenté le système de type du langage. Ce système est une extension conservative de celui de ML. La particularité de la preuve de sûreté du typage vient de la définition de la forme normale. En effet, une forme normale ne doit pas être une valeur pour que le typage soit sûr, il faut que ce soit une expression qui peut être réduite par la réduction $\rightarrow_{eoi}$.

Troisième partie

Ordonnancement

Chapitre 6

Une sémantique efficace pour le modèle réactif I : Glouton

Dans la partie précédente, les deux sémantiques décrivent assez simplement la réaction d'un programme, mais ne donnent pas d'indications pour l'implantation d'un interprète efficace. Le but de cette partie est d'effectuer le passage de ces sémantiques intuitives jusqu'à l'implantation.

Ce chapitre présente la sémantique de Glouton, une implantation de Junior que nous avons réalisée avant de travailler sur ReactiveML. Le nom Glouton vient de l'idée de faire un ordonnancement glouton des calculs. Comme dans la programmation événementielle, les expressions en attente d'événements sont postées et un ordonnanceur les traite. Ce travail s'inspire de Simple l'implantation faite par Laurent Hazard [55]. Simple est la première implantation du modèle réactif qui a utilisé des files d'attente associées aux signaux. Ces files d'attente permettent de déclencher l'exécution des actions seulement quand elle peuvent être exécutées. C'est l'idée d'une exécution *paresseuse* sans attente active des signaux.

La particularité de nos interprètes par rapport à d'autres implantations efficaces comme Storm [100] et Reflex [28] est de destructurer l'arbre de syntaxe abstraite du programme pour avoir un accès direct aux instructions à exécuter. Mais cette approche a pour contrepartie de rendre plus difficile l'implantation des constructions de bloc comme la suspension et la préemption.

Les idées et l'implantation ont été faites pour Junior. Ce travail a d'ailleurs été comparé aux autres implantations du modèle réactif dans deux thèses ([1, 28]). Mais pour l'unité du document, nous trouvons plus intéressant de décrire cette formalisation en ReactiveML.

6.1 Les clés d'un interprète efficace

Une partie importante des travaux sur l'approche réactive porte sur l'implantation d'interprètes efficaces [55, 100, 20, 28, 40]. Dans [1], Raúl Acosta-Bermejo récapitule un certain nombre de caractéristiques communes à ces interprètes.

Le point principal est l'attente des signaux. Dans une implantation naïve du modèle réactif comme Rewrite [56], toutes les branches parallèles du programme sont activées successivement de gauche à droite jusqu'à l'obtention d'un point fixe. Ainsi, dans le programme suivant, l'expression `await immediate s1` est activée deux fois avant de pouvoir se réduire à la troisième activation [1].

[1] C'est l'exemple classique de la cascade inverse.

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```

$$\xrightarrow{\quad \texttt{s0} \quad}$$

```
await immediate s1 || await immediate s0; emit s1
```

$$\xrightarrow{\quad \texttt{s0, s1} \quad}$$

```
await immediate s1
```

Dans Rewrite, le fait que `await immediate s1` ne puisse pas réagir tant que le signal `s1` n'a pas été émis n'est pas utilisé. À chaque tour d'activation la première branche est testée même si elle ne peut pas se réduire. Dans ce cas, l'implantation fait de *l'attente active*. La solution proposée par Laurent Hazard est d'enregistrer la première branche dans une file d'attente de `s1` et de la réactiver uniquement lorsque le signal est émis. Ainsi une instruction est activée au plus deux fois par instant. Ceci correspond donc à une *attente passive* : un calcul est mis en attente sur un signal et sera activé seulement lorsque le signal sera émis.

Étudions la variante non-instantanée de la cascade inverse [1] :

```
await s1 || await s0; emit s1 || emit s0
```

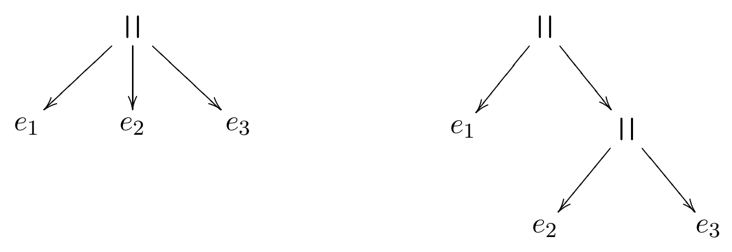
Ici, `s1` n'est pas émis au premier instant mais au second. En fonction des implantations, les listes d'attente sont conservées d'un instant à l'autre ou pas. Nous distinguons donc l'attente *intra-instant* de l'exemple précédent, de l'attente *inter-instants* de cet exemple.

Si l'ordonnancement est parfait (on n'active jamais plus d'une fois une expression), la gestion des files d'attente ajoute bien sûr un surcoût. Mais en général, il est toujours pertinent d'implanter l'attente intra-instant dans un interprète. Le choix n'est pas aussi simple pour l'attente inter-instants car elle interfère avec la suspension et la préemption. Dans l'exemple suivant, l'expression `await s1` ne peut pas être débloquée par l'émission de `s1` même si elle est enregistrée dans la file d'attente de `s1` car `s0` est absent au second instant.

```
do await s1 when s0 || emit s0; pause; emit s1
```

Dans certain cas, les mécanismes mis en place pour avoir de l'attente passive inter-instants peuvent être plus coûteux que d'activer une fois par instant chaque instruction. Une des difficultés est qu'un choix d'implantation peut être bon ou mauvais en fonction de la structure du programme et même de ses entrées.

Un autre point mis en avant dans [1] est l'utilisation du parallèle n-aire. Par exemple, si on veut exécuter trois expressions en parallèle, il est préférable d'utiliser un parallèle à trois branches que deux parallèles binaires.



L'argument utilisé est que cela évite d'avoir à parcourir la structure du terme pendant l'exécution. Malheureusement, il est assez naturel en ReactiveML d'avoir des parallèles binaires. Par exemple le processus récursif `add` qui exécute un nouveau processus à chaque fois que le signal `new_proc` est émis, crée un peigne de parallèles :

```
let rec process add new_proc =
  await new_proc(p) in
  run p
  ||
  run (add new_proc)
val add : ('a, 'b process) event -> unit process
```

La solution que nous proposons est de casser la structure du terme à l'exécution. Ainsi, il y a toujours un accès direct à l'expression à exécuter. Au lieu de manipuler une expression, l'intepréte travaille avec des ensembles d'expressions à exécuter en parallèle.

Pour faire l'optimisation précédente, il faut que le parallèle soit associatif. Si le parallèle est en plus commutatif, il y a une plus grande liberté dans l'ordonnancement ce qui facilite l'implantation. En SugarCubes, la composition parallèle (`merge`) n'est pas comme en ReactiveML. Elle garantit que la branche gauche est toujours exécutée avant la branche droite au cours d'une réaction. Cette propriété ajoute des contraintes dans l'implantation des ordonnanceurs et donc se traduit souvent par un surcoût à l'exécution.

Enfin, le modèle réactif fait disparaître des problèmes d'ordonnancement des langages asynchrones. Par exemple, le choix de l'ordonnancement a très peu d'influence sur la consommation mémoire ce qui n'est pas vrai dans le cadre asynchrone. Illustrons cela sur un exemple simple. Définissons un processus producteur qui émet dix entiers et un processus consommateur qui les affiche. En ReactiveML, le producteur peut être écrit d'au moins deux façons. La première émet toutes les valeurs dans le même instant, la seconde émet une valeur par instant :

```
let process producer1 s =
  for i = 1 to 10 do
    emit s i
  done

let process producer2 s =
  for i = 1 to 10 do
    emit s i;
    pause
  done

let consumer s =
  loop
    await s(x) in List.iter print_int x
  end
```

Ces deux producteurs ont des consommations de mémoire différentes. Le premier doit mémoriser les dix entiers pendant un instant, alors que le second doit garder un seul entier par instant. Le programmeur peut choisir l'implantation la plus adaptée à son problème. Dans un cadre asynchrone, c'est l'ordonnanceur qui équilibre l'exécution entre le producteur et le consommateur.

En fonction de l'ordonnancement, un programme asynchrone peut donc avoir à stocker entre un et dix entiers.

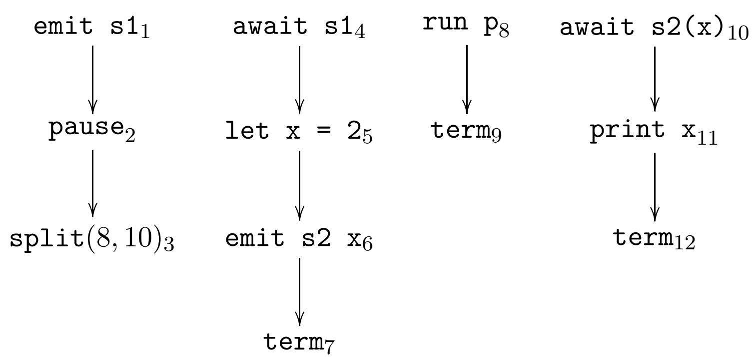
6.2 Les groupes : le format d'exécution de Glouton

6.2.1 La définition des groupes

La sémantique de GLOUTON n'est pas donnée directement sur le source REACTIVEML, mais sur un format que nous appelons les *Groupes*. Ce format est fait pour faciliter l'exécution efficace des programmes. Les deux idées fondatrices de ce format sont l'utilisation de continuations et le regroupement dans des *groupes* des branches parallèles qui doivent se synchroniser avant de pouvoir exécuter leur continuation.

Grâce à la représentation avec continuations, nous savons toujours ce qui doit être exécuté après une instruction. Ainsi, un programme peut être vu comme un ensemble de suites d'instructions et un ensemble de points de contrôle indiquant les branches à exécuter. Illustrons cela sur un exemple[2]. Nous trouvons à gauche le programme source et à droite sa représentation dans une forme simplifiée des groupes :

```
emit s1;
pause;
(run p
 ||
 await s2(x) in
 print x)
||
await s1;
let x = 2 in
emit s2 x
```



Les points d'entrée de ce programme sont les instructions 1 et 4 (le numéro des instructions est donné en indice). L'instruction `split`$(8, 10)$ représente le parallèle qui est imbriqué dans la branche gauche. Son exécution donne deux points de contrôle sur les instructions 8 et 10. Les instructions `term` indiquent la fin des branches parallèles. Nous constatons que cette représentation casse la structure de l'arbre de syntaxe abstraite du programme. À la place, elle propose une séquence d'instructions par branche parallèle.

Nous voulons maintenant ajouter l'instruction `emit s3` en séquence après le bloc précédent. Cette instruction ne peut être exécutée qu'une fois le bloc terminé. Afin de garder cette dépen-

[2]Cet exemple ne calcule rien d'intéressant, mais il permet de montrer la structure du programme.

dance, nous mettons les suites instructions précédentes dans un groupe :

$$\left\{\begin{array}{cccc} \texttt{emit s1}_1 & \texttt{await s1}_4 & \texttt{run p}_8 & \texttt{await s2(x)}_{10} \\ \downarrow & \downarrow & \downarrow & \downarrow \\ \texttt{pause}_2 & \texttt{let x = 2}_5 & \texttt{term}_9 & \texttt{print x}_{11} \\ \downarrow & \downarrow & & \downarrow \\ \texttt{split}(7,8)_3 & \texttt{emit s2 x}_6 & & \texttt{term}_{12} \\ & \downarrow & & \\ & \texttt{term}_7 & & \end{array} \quad \rhd^3\ 13 \right\} \quad \begin{array}{c} \texttt{emit s3}_{13} \\ \downarrow \\ \texttt{term}_{14}\end{array}$$

La partie "$\rhd^3$ 13" du groupe indique qu'il faut exécuter trois instructions `term` pour terminer l'exécution du groupe et que sa continuation est l'instruction 13. Le chiffre 3 vient du fait qu'il y a trois branches parallèles dans le bloc. La synchronisation de la terminaison n'est pas faite à la fin de chaque parallèle mais au moment de la séquence. Un groupe définit donc un bloc dont la continuation est exécutée lorsque toutes les branches parallèles du bloc ont terminé leur exécution.

Précisons la définition des groupes (G) :

$$G ::= \{\mathcal{I} \rhd^{cpt} k\} \qquad \mathcal{I} ::= \emptyset \mid t_i, \mathcal{I} \qquad k ::= (id, i)$$

On trouve trois informations dans un groupe : la liste des instructions numérotées $\mathcal{I}$, *cpt* le nombre de branches parallèles du groupe et k l'instruction à exécuter une fois le groupe terminé. k est composé de id un nom de groupe et i le numéro de l'instruction dans ce groupe[3]. Chaque groupe est donc associé à un nom dans un environnement $\mathcal{G}$.

$$\mathcal{G} ::= [G/id, ..., G/id]$$

Les instructions t dans les groupes sont les suivantes :

$$\begin{array}{rcl} t & ::= & \texttt{term} \mid e \mid \texttt{run}\ e \mid \texttt{await immediate}\ x \\ & & \mid \texttt{present}\ x\ \texttt{then}\ k\ \texttt{else}\ k \mid \texttt{split}\ k\ k \\ & & \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ (\mathcal{G}, k) \\ & & \mid \texttt{await}\ x\texttt{(}x'\texttt{)}\ \texttt{in}\ (\mathcal{G}, k) \\ & & \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ (\mathcal{G}, k) \\ \\ e & ::= & x \mid c \mid (e, e) \mid \lambda x.e \mid e\,e \mid \texttt{rec}\,x\,\texttt{=}\,e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{process}(\lambda x.\mathcal{G}, k) \\ & & \mid \texttt{pre}\ e \mid \texttt{pre ?}e \mid \texttt{emit}\ e\ e \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \end{array}$$

L'instruction `term` correspond à la fin d'une branche parallèle. Une expression e est une expression instantanée. Dans les expressions instantanées, la définition de processus est un ensemble de groupes et un point d'entrée pour commencer l'exécution. Le paramètre x représente la continuation du processus.

Les instructions `await/immediate` et `run` sont celles de REACTIVEML. Pour l'instruction `present`, dans les branches `then` et `else`, on donne le numéro des instructions correspondant

[3]Dans les exemples précédents, nous avons omis les noms de groupes car chaque instruction avait un numéro unique.

à ces branches. L'instruction `split` est paramétrée par ses deux continuations. Enfin, pour les instructions qui sont des lieurs, le corps est un ensemble de groupes et un point d'entrée.

Nous pouvons remarquer que les instructions `let/in` et `signal/in` sont dupliquées car elles ont une forme instantanée et une forme réactive.

Ce format se rapproche des formats intermédiaires utilisés pour la compilation de ESTEREL vers du logiciel (pas des circuits). Il se rapproche en particulier de ceux définis dans SAXO-RT [32], GRC [89] ou CEC [38]. La grande différence entre notre approche et la compilation d'ESTEREL est qu'en REACTIVEML, l'ordonnancement est fait à l'exécution alors qu'en ESTEREL, l'ordonnancement est fait à la compilation.

La définition du format des groupes que nous avons donnée correspond au noyau de REACTIVEML suivant :

$$\begin{array}{rcl} e & ::= & x \mid c \mid (e,e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x = e \mid \texttt{process}\ e \mid e\texttt{;}e \\ & & \mid e\texttt{||}e \mid \texttt{present}\ x\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{emit}\ e\ e \\ & & \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \mid \texttt{await immediate}\ x \\ & & \mid \texttt{await}\ x(x)\ \texttt{in}\ e \mid \texttt{pause} \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{run}\ e \mid \texttt{pre}\ e \mid \texttt{pre ?}e \\ \\ c & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \texttt{0} \mid \ldots \mid \texttt{+} \mid \texttt{-} \mid \ldots \end{array}$$

Ce noyau est un peu différent de celui du chapitre 3. Afin de simplifier la présentation nous ne traitons pas la suspension et la préemption pour le moment mais dans la seconde partie du chapitre. Par conséquent, nous ajoutons les constructions `await` au noyau pour expliquer l'attente d'un signal. De plus, GLOUTON ayant été fait pour JUNIOR, la terminaison de processus ne rend pas de valeurs. Donc, nous avons remplacé la construction `let/and/in` par $e_1 \texttt{||} e_2$ et nous interdisons dans la construction $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$ que e_1 soit une expression réactive. Nous verrons chapitre 7 les conséquences de ce choix.

Remarque :

> Avoir la partie gauche du `let/in` instantanée correspond à avoir la règle suivante dans la figure 3.2 page 48 :
>
> $$\frac{0 \vdash e_1 \qquad k \vdash e_2}{k \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2}$$

Enfin, dans ce noyau, les instructions qui testent les signaux s'appliquent uniquement à des noms de variables et pas à des expressions. Cette modification simplifie la présentation de la sémantique et ne réduit pas l'expressivité du langage. En effet, on peut toujours calculer le signal avant de le tester. Par exemple, l'expression $\texttt{await immediate}\ e$ devient :

$$\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in await immediate}\ x$$

6.2.2 Traduction de ReactiveML vers les Groupes

Nous ne faisons pas la traduction directement de REACTIVEML vers les groupes, mais nous utilisons un format intermédiaire. Dans ce format, nous distinguons deux formes de séquences. La première est la continuation qui permet d'exécuter une instruction après l'autre. La seconde forme attend la terminaison de plusieurs branches parallèles avant de pouvoir exécuter la suite du programme. Elle a également un rôle de synchronisation.

Traditionnellement, c'est la composition parallèle qui joue un double rôle. Elle permet à la fois de séparer le point de contrôle d'un programme en deux puis de le fusionner. Ici, le parallèle va garder son rôle de séparation, mais pas celui de fusion. Nous laissons la séquence synchroniser la terminaison des branches parallèles.

Cette approche permet de factoriser les points de synchronisation. Par exemple dans l'expression $(e_1 \,||\, (e_2 \,||\, e_3))\texttt{;}e_4$, si le parallèle synchronise la terminaison, il faut deux points de synchronisation alors que si c'est la séquence qui synchronise, il n'en faut qu'un seul.

Le langage intermédiaire que nous utilisons pour obtenir les groupes est le suivant :

$$
\begin{array}{lcl}
p & ::= & \texttt{term} \mid e.p \mid \texttt{pause}.p \mid \texttt{present}\ x\ \texttt{then}\ P\ \texttt{else}\ P.p \\
& & \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ P.p \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ P.p \\
& & \mid \texttt{await immediate}\ x.p \mid \texttt{await}\ x\texttt{(}x\texttt{)}\ \texttt{in}\ P.p \mid \texttt{run}\ e.p \mid P \,||\, P \\
\\
P & ::= & p \mid P\ \texttt{;} P \\
\\
e & ::= & x \mid c \mid (e, e) \mid \lambda x.e \mid e\, e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{rec}\, x\texttt{=}e \mid \texttt{pre}\ e \mid \texttt{pre ?}e \\
& & \mid \texttt{process}\ P \mid \texttt{emit}\ e\ e \mid \texttt{signal}\ e\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e
\end{array}
$$

Le "." correspond à la continuation et le ";" permet la synchronisation.

La fonction de traduction de ce format vers celui des groupes a la forme suivante :

$$Tr(P, id, i) = (\mathcal{G}, \mathcal{I}, k, cpt, i')$$

- P est le programme à traduire ;
- id est le nom du groupe dans lequel P est traduit ;
- i est le numéro de la prochaine instruction de P ;
- $\mathcal{G}$ est l'environnement contenant les groupes créés par la traduction ;
- $\mathcal{I}$ est la liste des instructions à ajouter dans id ;
- k est le point d'entrée du programme ;
- cpt est le nombre de branches parallèles créées par P ;
- i' est le compteur pour numéroter la prochaine instruction du groupe.

Cette fonction de traduction est définie récursivement sur la structure du terme. Étudions chacun des cas.

$$Tr(\texttt{term}, a, i) = ([\,], \texttt{term}_i, (a, i), 1, i + 1)$$

L'instruction `term` se traduit en $\texttt{term}_i$ où i est le numéro d'instruction. La traduction ne crée pas de sous-groupes et a une seule branche parallèle. Le numéro de la prochaine instruction à traduire est $i + 1$.

$$
\begin{array}{l}
Tr(e.p, a, i) = \\
\quad \text{let } (\mathcal{G}, \mathcal{I}, k, cpt, i') = Tr(p, a, i + 1) \text{ in} \\
\quad \text{let } e' = Tr_{ML}(e) \text{ in} \\
\quad (\mathcal{G}, (e'_i, \mathcal{I}), (a, i), cpt, i')
\end{array}
$$

Pour traduire $e.p$, nous traduisons p à partir du numéro d'instruction $i+1$, puis nous traduisons e en e'. Nous donnons à e' le numéro i pour que sa continuation soit la première instruction de p. Le résultat de la traduction est le résultat de la traduction de p auquel nous avons ajouté e'_i comme instruction et où (a, i) est le nouveau point d'entrée.

La fonction Tr_{ML} laisse les expressions ML inchangées. Le seul cas intéressant est celui de la traduction de processus :

$Tr_{ML}(\texttt{process } P) =$
  let $a = fresh()$ in
  let $(\mathcal{G}, \mathcal{I}, k, cpt, i) = Tr(P, a, 1)$ in
  $\texttt{process}(\lambda x.\mathcal{G}[\{\mathcal{I} \rhd^{cpt} x\}/a], k)$

La fonction $fresh$ crée un nouveau nom de groupe a et P est traduit dans ce groupe. À partir du résultat de la traduction de P, le groupe a est créé. Il est paramétré par sa continuation x.

$Tr(\texttt{pause}.p, a, i) =$
  let $(\mathcal{G}, \mathcal{I}, k, cpt, i') = Tr(p, a, i+1)$ in
  $(\mathcal{G}, ((\texttt{await immediate } eoi)_i, \mathcal{I}), (a, i), cpt, i')$

`pause` se traduit par l'attente d'un signal spécial : eoi. Ce signal est émis lorsque le programme ne peut plus réagir pour préparer la phase de traitement de fin d'instant.

$Tr(\texttt{run } e.p, a, i) =$
  let $(\mathcal{G}, \mathcal{I}, k, cpt, i') = Tr(p, a, i+1)$ in
  let $e' = Tr_{ML}(e)$ in
  $(\mathcal{G}, (\texttt{run } {e'}_i, \mathcal{I}), (a, i), cpt, i')$

$Tr(\texttt{await immediate } x.p, a, i) =$
  let $(\mathcal{G}, \mathcal{I}, k, cpt, i') = Tr(p, a, i+1)$ in
  $(\mathcal{G}, ((\texttt{await immediate } x)_i, \mathcal{I}), (a, i), cpt, i')$

Les traductions de `await/immediate` et `run` sont comme pour les cas précédents.

$Tr(\texttt{present } x \texttt{ then } P_1 \texttt{ else } P_2.p, a, i) =$
  let $(\mathcal{G}, \mathcal{I}, k, cpt, i') = Tr(p, a, i+1)$ in
  let $b = fresh()$ in
  let $(\mathcal{G}_1, \mathcal{I}_1, k_1, cpt_1, i_1) = Tr(P_1, b, 1)$ in
  let $G_1 = \{\mathcal{I}_1 \rhd^{cpt_1} k\}$ in
  let $c = fresh()$ in
  let $(\mathcal{G}_2, \mathcal{I}_2, k_2, cpt_2, i_2) = Tr(P_2, c, 1)$ in
  let $G_2 = \{\mathcal{I}_2 \rhd^{cpt_2} k\}$ in
  $([G_1/b, G_2/c] \cup \mathcal{G} \cup \mathcal{G}_1 \cup \mathcal{G}_2, ((\texttt{present } x \texttt{ then } k_1 \texttt{ else } k_2)_i, \mathcal{I}), (a, i), cpt, i')$

Pour traduire `present`, nous traduisons la continuation p dans le groupe a. Cette traduction nous donne un point d'entrée k. Nous traduisons ensuite les deux branches dans deux groupes différents. Ces groupes prennent le même k comme continuation. À partir de là, nous construisons l'instruction `present`.

$Tr(P_1 \,||\, P_2, a, i) =$
  let $(\mathcal{G}_1, \mathcal{I}_1, k_1, cpt_1, i_1) = Tr(P_1, a, i+1)$ in
  let $(\mathcal{G}_2, \mathcal{I}_2, k_2, cpt_2, i_2) = Tr(P_2, a, i_1)$ in
  $((\mathcal{G}_1, \mathcal{G}_2), ((\texttt{split } k_1\ k_2)_i, \mathcal{I}_1, \mathcal{I}_2), (a, i), cpt_1 + cpt_2, i_2)$

Pour la composition parallèle, les deux branches sont traduites et nous ajoutons une instruction `split` pour pouvoir les commencer. Le nombre d'instructions `term` à exécuter $(cpt_1 + cpt_2)$ est égal à la somme de celles des deux branches.

$Tr(P_1 \texttt{;} P_2, a, i) =$
  let $(\mathcal{G}_2, \mathcal{I}_2, k_2, cpt_2, i_2) = Tr(P_2, a, i)$ in
  let $b = fresh()$ in
  let $(\mathcal{G}_1, \mathcal{I}_1, k_1, cpt_1, i_1) = Tr(P_1, b, 1)$ in
  let $G_1 = \{\mathcal{I}_1 \rhd^{cpt_1} k_2\}$ in
  $([G_1/b] \cup \mathcal{G}_1 \cup \mathcal{G}_2, \mathcal{I}_2, k_1, cpt_2, i_2)$

Pour traduire la séquence, nous commençons par traduire la branche droite, puis la branche gauche dans un nouveau groupe. Ce nouveau groupe prend k_2 (le point d'entrée de la branche droite) comme continuation.

$$
\begin{array}{l}
Tr(\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ P.p, a, i) = \\
\quad \text{let } (\mathcal{G}, \mathcal{I}, k, cpt, i') = Tr(p, a, i+1) \text{ in} \\
\quad \text{let } e' = Tr_{ML}(e) \text{ in} \\
\quad \text{let } b = fresh() \text{ in} \\
\quad \text{let } (\mathcal{G}_b, \mathcal{I}_b, k_b, cpt_b, i_b) = Tr(P, b, 1) \text{ in} \\
\quad \text{let } G = \{\mathcal{I}_b \rhd^{cpt_b} k\} \text{ in} \\
\quad (\mathcal{G}, ((\texttt{let}\ x\ \texttt{=}\ e'\ \texttt{in}\ ([G/b] \cup \mathcal{G}_b, k_b))_i, \mathcal{I}), (a, i), cpt, i')
\end{array}
$$

Pour le `let/in`, nous traduisons le corps dans un groupe pour marquer la portée de x. Le groupe est associé au `let` pour pouvoir faire la substitution de x par sa valeur au moment de l'évaluation.

$$
\begin{array}{l}
Tr(\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ P.p, a, i) = \\
\quad \text{let } (\mathcal{G}, \mathcal{I}, k, cpt, i') = Tr(p, a, i+1) \text{ in} \\
\quad \text{let } e_1' = Tr_{ML}(e_1) \text{ in} \\
\quad \text{let } e_2' = Tr_{ML}(e_2) \text{ in} \\
\quad \text{let } b = fresh() \text{ in} \\
\quad \text{let } (\mathcal{G}_b, \mathcal{I}_b, k_b, cpt_b, i_b) = Tr(P, b, 1) \text{ in} \\
\quad \text{let } G = \{\mathcal{I}_b \rhd^{cpt_b} k\} \text{ in} \\
\quad (\mathcal{G}, ((\texttt{signal}\ x\ \texttt{default}\ e_1'\ \texttt{gather}\ e_2'\ \texttt{in}\ ([G/b] \cup \mathcal{G}_b, k_b))_i, \mathcal{I}), (a, i), cpt, i')
\end{array}
$$

La traduction de `signal` utilise la même technique de traduction que le `let/in`.

6.3 Évaluation

6.3.1 Exécution d'un programme

La stratégie d'ordonnancement que nous présentons est une stratégie *gloutonne*. Nous l'avons réalisée en essayant de comprendre les travaux de Laurent Hazard dont il existe une implantation mais pas de description. Elle est basée sur l'utilisation de files d'attente. Ainsi, les expressions en attente d'un événement sont exécutées uniquement lorsqu'il est émis.

Dans cette sémantique, un programme est représenté par son code $\mathcal{G}$, l'ensemble des instructions $\mathcal{C}$ à exécuter dans l'instant courant et l'ensemble des instructions $\mathcal{W}$ en attente. Le principe de l'évaluation d'un programme est le suivant :

1. Les instructions présentes dans $\mathcal{C}$ sont prises une à une pour être traitées. Si l'instruction traitée teste un signal qui est absent, elle est enregistrée dans l'ensemble $\mathcal{W}$. Par contre, si elle peut être entièrement traitée, elle est exécutée et sa continuation est ajoutée dans $\mathcal{C}$ pour être exécutée dans le même instant. Enfin, l'émission d'un signal fait passer les instructions en attente dans $\mathcal{W}$ dans l'ensemble $\mathcal{C}$.

2. Une fois que toutes les instructions de $\mathcal{C}$ ont été traitées ($\mathcal{C} = \emptyset$), la fin d'instant peut être déclarée. Le signal *eoi* (End Of Instant) est alors émis et les instructions présentes dans $\mathcal{W}$ sont traitées pour construire le nouvel ensemble $\mathcal{C}$ à exécuter à l'instant suivant.

Nous définissons maintenant formellement ces deux étapes de la réaction d'un programme avec des règles de la forme suivante :

$$S,\,\mathcal{G},\,\mathcal{W} \vdash \mathcal{C} \rightsquigarrow S',\,\mathcal{G}',\,\mathcal{W}' \qquad \text{et} \qquad S,\,\mathcal{G},\,\mathcal{W} \rightsquigarrow_{eoi} S,\,\mathcal{G}',\,\mathcal{W}' \vdash \mathcal{C}$$

$\mathcal{C}$ est un ensemble de couples : (nom de groupe, numéro d'instruction). $\mathcal{W}$ est une fonction qui associe à chaque signal la liste des numéros d'instruction qui sont en attente.

$$\mathcal{C} ::= \emptyset \mid (id, i), \mathcal{C} \qquad\qquad \mathcal{W} ::= [\ldots, \mathcal{C}/n, \ldots]$$

Pour les environnements de signaux S, nous reprenons la même définition que dans le chapitre 3.3.1 (page 49). Donc à chaque signal n, on associe dans l'environnement : une valeur par défaut, une fonction de combinaison, le statut précédent du signal et le multi-ensemble des valeurs émises pendant l'instant.

La première étape de la réaction est définie par les règles :

$$S,\,\mathcal{G},\,\mathcal{W} \vdash \emptyset \rightsquigarrow S,\,\mathcal{G},\,\mathcal{W}$$

$$\frac{\begin{array}{c}\mathcal{G}(a,i) = t \qquad S,\,\mathcal{G},\,\mathcal{W} \vdash t_i^a \longrightarrow S',\,\mathcal{G}',\,\mathcal{W}' \vdash \mathcal{C}' \\ \mathcal{W}'', \mathcal{C}'' = wakeUp(S\#S', \mathcal{W}') \qquad S',\,\mathcal{G}',\,\mathcal{W}'' \vdash \mathcal{C} \cup \mathcal{C}' \cup \mathcal{C}'' \rightsquigarrow S'',\,\mathcal{G}'',\,\mathcal{W}'''\end{array}}{S,\,\mathcal{G},\,\mathcal{W} \vdash (a,i), \mathcal{C} \rightsquigarrow S'',\,\mathcal{G}'',\,\mathcal{W}'''}$$

La première règle indique que si l'ensemble des instructions à exécuter à l'instant courant est vide alors l'instant est terminé.

Pour la seconde règle, le code de l'instruction (a,i) est récupéré dans $\mathcal{G}$ avec la fonction $\mathcal{G}(a,i)$. Cette fonction est définie par :

$$\text{Si } \mathcal{G}(a) = \{..., t_i, ... \rhd^{cpt} k\} \text{ alors } \mathcal{G}(a,i) = t$$

Puis, cette expression est exécutée avec la réaction $S,\,\mathcal{G},\,\mathcal{W} \vdash t_i^a \longrightarrow S',\,\mathcal{G}',\,\mathcal{W}' \vdash \mathcal{C}'$ définie section 6.3.2.

Ensuite avec la fonction *wakeUp*, toutes les instructions qui étaient en attente d'un signal qui a été émis pendant la réaction de t sont sorties de $\mathcal{W}'$. L'opération $S\#S'$ définit l'ensemble N contenant tous les signaux qui étaient absents dans S et qui sont présents dans S'. Ainsi pour tout $n \in Dom(S')$, alors

$$n \in S\#S' \quad \text{ssi} \quad n \notin S \text{ et } n \in S'$$

La fonction *wakeUp* qui débloque les instructions en attente d'un signal qui a été émis est définie récursivement par :

$wakeUp(\emptyset, \mathcal{W}) = \mathcal{W}, \emptyset$

$wakeUp(\{n\} \cdot N, \mathcal{W}) =$
  let $\mathcal{W}', \mathcal{C} = wakeup(N, \mathcal{W})$ in
  let $\mathcal{C}' = \mathcal{W}'(n)$ in
  let $\mathcal{W}'' = \mathcal{W}' \backslash n$ in
  $\mathcal{W}'', (\mathcal{C} \cup \mathcal{C}')$

Après avoir débloqué les instructions associées dans $\mathcal{W}'$ à un signal qui a été émis, l'exécution de l'instant continue avec la réaction $\leadsto$.

La seconde étape de l'exécution d'un instant applique la réaction $\longrightarrow$ à tous les éléments de $\mathcal{W}$:

$$S,\, \mathcal{G},\, \emptyset \leadsto_{eoi} S,\, \mathcal{G}',\, \emptyset \vdash \emptyset$$

$$\frac{\mathcal{G}(a,i) = t \qquad S,\, \mathcal{G},\, \emptyset \vdash t_i^a \longrightarrow S,\, \mathcal{G}',\, \mathcal{W}' \vdash \mathcal{C}' \qquad S,\, \mathcal{G}',\, \mathcal{W} \leadsto_{eoi} S,\, \mathcal{G}'',\, \mathcal{W}'' \vdash \mathcal{C}''}{S,\, \mathcal{G},\, (\mathcal{W} + [(a,i)/n]) \leadsto_{eoi} S,\, \mathcal{G}'',\, \mathcal{W}', \mathcal{W}'' \vdash \mathcal{C}', \mathcal{C}''}$$

L'opération + ajoute la continuation k dans la file d'attente du signal n. Elle est définie par :
$(\mathcal{W} + [k/n])(n') = \begin{cases} \mathcal{W}(n') & \text{si } n' \neq n \\ k, \mathcal{W}(n) & \text{si } n' = n \end{cases}$

Il nous reste à définir la réaction d'une instruction t :

$$S,\, \mathcal{G},\, \mathcal{W} \vdash t_i^a \longrightarrow S',\, \mathcal{G}',\, \mathcal{W}' \vdash \mathcal{C}$$

6.3.2 Exécution d'une instruction

La règle $S,\, \mathcal{G},\, \mathcal{W} \vdash t_i^a \longrightarrow S',\, \mathcal{G}',\, \mathcal{W}' \vdash \mathcal{C}$ définit la réaction de l'instruction t définie dans le groupe a à la position i. Cette réaction se fait dans un environnement de signaux S. $\mathcal{W}$ est l'ensemble des instructions en attente. La réaction de t peut modifier l'ensemble des instructions en attente et générer un ensemble $\mathcal{C}$ d'instructions à exécuter dans l'instant.

Nous commençons par définir l'exécution d'une expression instantanée e.

$$\frac{e/S \Downarrow v/S'}{S,\, \mathcal{G},\, \mathcal{W} \vdash e_i^a \longrightarrow S',\, \mathcal{G},\, \mathcal{W} \vdash (a, i+1)}$$

e est évalué en la valeur v et l'exécution dans l'instant de l'instruction $(a, i+1)$ est demandée. e étant l'instruction (a, i), l'instruction $(a, i+1)$ est sa continuation.

Le prédicat $e/S \Downarrow v/S'$ définit l'évaluation d'une expression e en une valeur v dans un environnement de signaux S. L'environnement S' contient les signaux créés et émis par la réaction. Les règles définissant ce prédicat sont définies figure 6.1. Ce sont celles de ML auxquelles nous ajoutons les opérations sur les signaux.

Nous regardons maintenant l'attente d'un signal. On illustre ainsi l'utilisation des files d'attente $\mathcal{W}$.

$$\frac{n \in S}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{await immediate } n)_i^a \longrightarrow S,\, \mathcal{G},\, \mathcal{W} \vdash (a, i+1)}$$

$$\frac{n \notin S \qquad \mathcal{W}' = \mathcal{W} + [(a,i)/n]}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{await immediate } n)_i^a \longrightarrow S,\, \mathcal{G},\, \mathcal{W}' \vdash \emptyset}$$

Dans la première règle, le signal est présent. On exécute donc instantanément la continuation. Dans la seconde, le signal testé est absent, on enregistre alors l'expression associée au signal n dans $\mathcal{W}$.

$$\frac{n \in S}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{present } n \texttt{ then } k_1 \texttt{ else } k_2)_i^a \longrightarrow S,\, \mathcal{G},\, \mathcal{W} \vdash k_1}$$

$$v/S \Downarrow v/S \qquad \frac{e[x \leftarrow \texttt{rec}\, x \texttt{=} e]/S \Downarrow v/S'}{\texttt{rec}\, x \texttt{=} e/S \Downarrow v/S'} \qquad \frac{e_1/S \Downarrow v/S_1 \quad e_2[x \leftarrow v]/S_1 \Downarrow v'/S'}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2/S \Downarrow v'/S'}$$

$$\frac{e_1/S \Downarrow \lambda x.e/S_1 \quad e_2/S_1 \Downarrow v/S_2 \quad e[x \leftarrow v]/S_2 \Downarrow v'/S'}{e_1\, e_2/S \Downarrow v'/S'} \qquad \frac{e_1/S \Downarrow v_1/S_1 \quad e_2/S_1 \Downarrow v_2/S'}{(e_1, e_2)/S \Downarrow (v_1, v_2)/S'}$$

$$\frac{e/S \Downarrow n/S' \quad (b, v) = S'^p(n)}{\texttt{pre}\ e/S \Downarrow b/S'} \qquad \frac{e/S \Downarrow n/S' \quad (b, v) = S'^p(n)}{\texttt{pre ?}e/S \Downarrow v/S'} \qquad \frac{e_1/S \Downarrow n/S_1 \quad e_2/S_1 \Downarrow v/S'}{\texttt{emit}\ e_1\ e_2/S \Downarrow \texttt{()}/S' + [v/n]}$$

$$\frac{e_1/S \Downarrow v_1/S_1 \quad e_2/S_1 \Downarrow v_2/S_2 \quad n \notin Dom(S_2) \quad e[x \leftarrow n]/S_2[(v_1, v_2, (\mathit{false}, v_1), \emptyset)/n] \Downarrow v/S'}{\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e/S \Downarrow v/S'}$$

FIG. 6.1 – Réaction des expressions instantanées.

$$\frac{n \notin S \qquad eoi \notin S \qquad \mathcal{W}' = \mathcal{W} + [(a, i)/n]}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{present}\ n\ \texttt{then}\ k_1\ \texttt{else}\ k_2)_i^a \longrightarrow S,\, \mathcal{G},\, \mathcal{W}' \vdash \emptyset}$$

$$\frac{n \notin S \qquad eoi \in S}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{present}\ n\ \texttt{then}\ k_1\ \texttt{else}\ k_2)_i^a \longrightarrow S,\, \mathcal{G},\, \mathcal{W} \vdash k_2}$$

Pour la construction `present`, lorsque le signal est présent, on exécute instantanément la branche `then`. On ajoute donc l'instruction k_1 à l'ensemble des instructions à exécuter. Quand le signal est absent, il faut distinguer deux cas. Si le signal de fin d'instant n'a pas encore été émis, il peut encore y avoir des émissions de signaux. Dans ce cas, il faut donc attendre pour savoir si n va être émis pendant l'instant ou pas. Si *eoi* est présent, cela signifie que l'instant est terminé, et que l'on prépare la réaction pour l'instant suivant. On peut donc demander l'exécution de la branche `else`.

Afin de coder la composition parallèle, nous avons l'instruction `split` :

$$\frac{}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{split}\ k_1\ k_2)_i^a \longrightarrow S,\, \mathcal{G},\, \mathcal{W} \vdash (k_1, k_2)}$$

Cette instruction demande simplement l'exécution des instructions k_1 et k_2 dans l'instant.

Pour la terminaison des branches parallèles et le déclenchement de la séquence, nous voyons l'instruction `term` :

$$\frac{\mathcal{G}(a) = \{\mathcal{I} \rhd^{cpt} k\} \qquad cpt > 1 \qquad \mathcal{G}' = \mathcal{G}[\{\mathcal{I} \rhd^{cpt-1} k\}/a]}{S,\, \mathcal{G},\, \mathcal{W} \vdash \texttt{term}_i^a \longrightarrow S,\, \mathcal{G}',\, \mathcal{W} \vdash \emptyset}$$

$$\frac{\mathcal{G}(a) = \{\mathcal{I} \rhd^{1} k\}}{S,\, \mathcal{G},\, \mathcal{W} \vdash \texttt{term}_i^a \longrightarrow S,\, \mathcal{G}\backslash a,\, \mathcal{W} \vdash k}$$

`term` décrémente le compteur de branches parallèles de son groupe. Lorsque toutes les branches parallèles d'un groupe ont terminé leur exécution, la continuation k est alors déclenchée. Dans ce cas, le groupe a étant terminé, il est retiré de l'environnement $\mathcal{G}$.

L'instruction `run` e ajoute de nouveaux groupes à l'environnement $\mathcal{G}$ et les exécute.

$$\frac{e/S \Downarrow \texttt{process}(\lambda x.\mathcal{G}', k)/S' \qquad \mathcal{G}'' = \mathcal{G}.(\mathcal{G}'[x \leftarrow (a, i+1)])}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{run}\ e)_i^a \longrightarrow S',\, \mathcal{G}'',\, \mathcal{W} \vdash k}$$

L'expression e s'évalue en une définition de processus. Dans la définition $\texttt{process}(\lambda x.\mathcal{G}', k)$, $\mathcal{G}'$ est paramétré par la continuation x. Donc pour ajouter $\mathcal{G}'$ à $\mathcal{G}$, on substitue x par la continuation du `run`. L'opération $\mathcal{G}.\mathcal{G}'$ étend la définition de $\mathcal{G}$ avec celle de $\mathcal{G}'$. Elle est possible uniquement s'il n'y a pas de collisions entre les noms de groupes ($Dom(\mathcal{G}) \cap Dom(\mathcal{G}') = \emptyset$). Afin de garantir cette propriété, les règles sont données modulo le renommage des groupes.

Pour les déclarations de signaux et de variables locales, on a une approche similaire.

$$\frac{\begin{array}{c} e_1/S \Downarrow v_1/S_1 \quad e_2/S_1 \Downarrow v_2/S_2 \quad n \notin Dom(S_2) \quad S' = S_2[(v_1, v_2, (\mathit{false}, v_1), \emptyset)/n] \\ \mathcal{G}'' = \mathcal{G}.(\mathcal{G}'[x \leftarrow n]) \end{array}}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ (\mathcal{G}', k))_i^a \longrightarrow S',\, \mathcal{G}'',\, \mathcal{W} \vdash k}$$

Pour la déclaration de signal, on évalue la valeur par défaut (v_1) et la fonction de combinaison (v_2). Puis à partir de ces informations, on enrichit l'environnement de signaux avec un nouveau signal n pour construire l'environnement S'. Enfin on substitue x par n dans la définition de $\mathcal{G}'$.

$$\frac{e/S \Downarrow v/S' \qquad \mathcal{G}'' = \mathcal{G}.(\mathcal{G}'[x \leftarrow v])}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ (\mathcal{G}', k))_i^a \longrightarrow S',\, \mathcal{G}'',\, \mathcal{W} \vdash k}$$

Pour le `let/in`, on évalue simplement e en v et on substitue x par v dans les groupes qui définissent le corps du `let`.

Nous terminons avec le `await` valué.

$$\frac{(n \notin S) \vee (eoi \notin S) \qquad \mathcal{W}' = \mathcal{W} + [(a, i)/n]}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{await}\ n(x)\ \texttt{in}\ (\mathcal{G}', k))_i^a \longrightarrow S,\, \mathcal{G},\, \mathcal{W}' \vdash \emptyset}$$

$$\frac{n \in S \qquad eoi \in S \qquad S(n) = (d, g, p, m) \qquad v = \mathit{fold}\ g\ m\ d \qquad \mathcal{G}'' = \mathcal{G}.(\mathcal{G}'[x \leftarrow v])}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{await}\ n(x)\ \texttt{in}\ (\mathcal{G}', k))_i^a \longrightarrow S,\, \mathcal{G},\, \mathcal{W} \vdash k}$$

Tant que le signal n est absent ou que l'on n'est pas à la fin d'instant, il faut attendre. Une fois ces deux conditions réunies, on peut calculer la valeur v associée au signal (comme dans la sémantique comportementale page 55) et substituer x par v dans la définition de $\mathcal{G}'$.

6.3.3 Sémantique sans échappement de portée

En Junior, la liaison des noms est dynamique alors qu'elle est statique en ReactiveML. Donc en Junior, il n'y a pas de phénomène d'échappement de portée. On peut ainsi implanter la boucle `loop` comme un opérateur de réinitialisation plutôt que comme une fonction récursive. Cette approche limite les allocations et désallocations pendant l'exécution et conduit donc à une implantation plus efficace.

Dans le programme suivant, le signal `channel` transporte des noms de signaux. Il est émis avec pour valeur associée le signal `s` déclaré à l'intérieur du `loop`. La valeur émise sur `channel` est récupérée et émise à l'extérieur de la boucle.

```
signal channel in
signal s in
loop
  signal s in
  emit channel s;
  pause
end
||
await one channel (x) in
emit x
||
await s;
print_string "Present"
```

Avec la sémantique de ReactiveML, le message `Present` n'est jamais affiché. Mais avec la sémantique de Junior, le signal `s` émis sur `channel` est lié dynamiquement au signal `s` le plus externe. Dans ce cas, l'instruction `await s` peut être réduite et le message `Present` est affiché. Comme il n'y a pas d'échappement de portée, il n'y a jamais plus d'une instance active du signal `s` définie à l'intérieur de la boucle.

Dans un interprète sans échappement de portée, il n'est pas nécessaire d'allouer un signal frais à chaque fois que l'on rencontre une même instruction `signal`, nous pouvons réutiliser la même cellule mémoire dans l'environnement S. Pour implanter la réinitialisation, il faut que les groupes gardent la valeur initiale du compteur de branches parallèles actives. Nous ajoutons donc en indice du symbole $\triangleright$ cette valeur.

$$G ::= \{\mathcal{I} \triangleright^{cpt}_{cpt_{init}} k\}$$

Ainsi maintenant lorsqu'un groupe termine son exécution, au lieu d'être retiré de l'environnement des groupes, il est réinitialisé :

$$\frac{\mathcal{G}(a) = \{\mathcal{I} \triangleright^{1}_{cpt} k\} \qquad \mathcal{G}' = \mathcal{G}[a \leftarrow \{\mathcal{I} \triangleright^{cpt}_{cpt} k\}]}{S, \mathcal{G}, \mathcal{W} \vdash \mathtt{term}^a_i \longrightarrow S, \mathcal{G}', \mathcal{W} \vdash k}$$

Enfin, nous pouvons définir le `loop` comme un groupe qui prend son point d'entrée comme continuation.

$$\begin{array}{l} Tr(\mathtt{loop}\ P, a, i) = \\ \quad \text{let } b = fresh() \text{ in} \\ \quad \text{let } (\mathcal{G}, \mathcal{I}, k, cpt, i') = Tr(P, b, 1) \text{ in} \\ \quad \text{let } G = \{\mathcal{I} \triangleright^{cpt}_{cpt} k\} \text{ in} \\ \quad ([G/b], \emptyset, k, 1, i) \end{array}$$

Pour donner la sémantique de Junior, il faut apporter d'autres modifications afin de coder la liaison dynamique. Mais cette approche peut aussi être intéressante en ReactiveML. Si on arrive à identifier des parties de programmes sans échappement de portée, on peut appliquer la même technique d'implantation.

6.4 Ajout des constructions de contrôle

La sémantique de Glouton que nous venons de présenter ne prend pas en compte les constructions de suspension et préemption. L'ajout de ces constructions apporte une grande complexité à la réalisation des interprètes. De nombreux choix peuvent être faits pour les implanter. La pertinence de ces choix dépend du programme exécuté. Nous proposons ici une première solution pour les deux constructions. Nous en verrons une autre chapitre 7.3.

6.4.1 La suspension

Nous traitons ici la construction `do/when`. Nous allons voir comment remplacer cette construction par des `await`.

La construction `do/when` peut être vue comme le test de présence d'un signal avant l'exécution de chaque instruction de son corps. Par exemple, les deux programmes suivants sont équivalents :

```
do                                await immediate s;
  emit a;                         emit a;
  pause;                          await immediate s;
  emit b                          pause;
when s                            await immediate s;
                                  emit b
```

Nous pouvons affiner l'analyse en remarquant qu'il suffit de tester la présence du signal qui contrôle l'exécution qu'une seule fois au début de chaque instant. Il n'est donc pas nécessaire de conserver les `await` après les expressions instantanées.

```
do                                await immediate s;
  emit a;                         emit a;
  pause;                          pause;
  emit b                          await immediate s;
when s                            emit b
```

Pour les instructions qui peuvent être exécutées sur plusieurs instants comme `await`, Il faut faire attention de bien tester le signal de contrôle à chaque instant et pas seulement au premier instant. Par exemple, les deux programmes suivants ne sont pas équivalents :

```
do                                await immediate s;
  await immediate a;              await immediate a;
  emit b                          await immediate s;
when s                            emit b
```

Si `s` est émis au premier et au troisième instants et `a` est émis au deuxième instant, alors le programme de droite émet `b` alors que celui de gauche ne l'émet pas.

Pour résoudre ce problème, nous utilisons les *configurations événementielles*. Les configurations permettent d'exprimer des expressions booléennes sur la présence des signaux. En particulier, la présence simultanée de deux signaux peut être testée avec l'opérateur $\wedge$. Les configurations événementielles existent en Junior et nous verrons chapitre 11.3 comment elles ont été intégrées dans ReactiveML.

En utilisant les configurations, l'exemple précédent devient :

```
do                                await immediate (s /\ a);
  await immediate a;              emit b
  emit b
when s
```

Ici, il faut bien que `s` et `a` soient présents dans le même instant pour pouvoir passer le `await` et émettre `b`.

Donc la méthode que nous utilisons pour traiter le `do/when` est de le traduire avec des `await` et des configurations événementielles. La fonction de transformation de programme a la forme suivante :

$$e \overset{c}{\rightsquigarrow} e'$$

e est l'expression à traduire, c est la configuration qui contrôle l'exécution de e, et e' est le programme transformé. Cette transformation est faite directement sur le source ReactiveML. Ainsi, nous pouvons utiliser cette transformation pour toutes les implantations du langage.

Les configurations utilisées dans la transformation sont seulement des conjonctions. Elles sont définies par :

$$c \ ::= \ n \ | \ x \ | \ \mathit{true} \ | \ c \wedge c$$

n est un nom de signal, x est une variable, *true* est la configuration qui est toujours vraie et $c \wedge c$ la conjonction de deux configurations. La valeur *true* est utilisée pour transformer les programmes qui ne sont pas dans un `do/when`.

La fonction de transformation est définie figure 6.2. Commentons ces règles :

- Pour la transformation des définitions de processus (`process` e), nous introduisons un nom x qui sert à paramétrer la transformation du corps du processus car nous ne connaissons pas le contexte dans lequel le processus sera exécuté.
- `run` donne le contexte de contrôle courant au processus qu'il doit exécuter.
- Pour les lieurs comme `signal/in` et `let/in`, il faut vérifier que le nom qui est introduit ne capture pas une variable qui est utilisée dans la configuration. En cas de capture, nous pouvons toujours appliquer une α-conversion.
- Un `await` est ajouté derrière la `pause` car l'instruction qui la suit sera toujours exécutée à l'instant d'après. Il faut donc vérifier si la configuration est présente avant de l'exécuter.
- Dans la construction `present`, le test à l'absence ajoute un délai. Il faut donc vérifier si la configuration est satisfaite avant d'exécuter la branche `else`. Au contraire, l'exécution de la branche `then` étant instantanée, si le test de présence a pu être exécuté, c'est que la configuration est présente. Il n'est donc pas nécessaire de la tester de nouveau dans la branche `then`.
- Comme nous l'avons vu, `await/immediate` doit tester la conjonction de la configuration de contrôle et du signal attendu. L'exécution de sa continuation étant instantanée, nous n'avons pas à retester la configuration après le `await`.
- Pour la construction `await/in`, nous commençons par attendre la conjonction de la configuration de contrôle et du signal. Un fois le signal émis et la configuration satisfaite, on exécute le corps (*body*). Le corps récupère instantanément la valeur du signal car avec le `await` précédent nous sommes sûr que le signal est présent. La partie droite du `await/in` étant exécutée à l'instant d'après, il faut tester la configuration avant de pouvoir l'exécuter.
- Enfin, pour le `do/when`, le nouveau contexte de contrôle c' est la conjonction du contexte courant et de x. On transforme le corps du `do/when` dans ce contexte. L'expression rendue par la transformation est le corps du `do/when` gardé par l'attente de c'.
- Toutes les autres règles propagent seulement la transformation.

Afin de définir complètement le traitement de la suspension, il nous faut maintenant définir le `await/immediate` qui attend plusieurs signaux. Les règles sont quasiment les mêmes que dans la section 6.3. Il suffit de remplacer les tests $n \in S$ et $n \notin S$ par des tests de configurations.

$$x \overset{c}{\leadsto} x \qquad const \overset{c}{\leadsto} const \qquad \frac{e \overset{c}{\leadsto} e'}{\lambda x.e \overset{c}{\leadsto} \lambda x.e'} \qquad \frac{x \notin fv(c) \qquad e \overset{c}{\leadsto} e'}{\texttt{rec}\ x\texttt{=}e \overset{c}{\leadsto} \texttt{rec}\ x\texttt{=}e'}$$

$$\frac{e_1 \overset{c}{\leadsto} e_1' \quad e_2 \overset{c}{\leadsto} e_2'}{e_1\ e_2 \overset{c}{\leadsto} e_1'\ e_2'} \qquad \frac{e_1 \overset{c}{\leadsto} e_1' \qquad e_2 \overset{c}{\leadsto} e_2'}{\texttt{emit}\ e_1\ e_2 \overset{c}{\leadsto} \texttt{emit}\ e_1'\ e_2'} \qquad \frac{e_1 \overset{c}{\leadsto} e_1' \quad e_2 \overset{c}{\leadsto} e_2'}{e_1\texttt{;}e_2 \overset{c}{\leadsto} e_1'\texttt{;}e_2'}$$

$$\frac{e_1 \overset{c}{\leadsto} e_1' \quad e_2 \overset{c}{\leadsto} e_2'}{e_1\texttt{||}e_2 \overset{c}{\leadsto} e_1'\texttt{||}e_2'} \qquad \frac{x \notin fv(e) \qquad e \overset{x}{\leadsto} e'}{\texttt{process}\ e \overset{c}{\leadsto} \lambda x.\texttt{process}\ e'} \qquad \frac{e \overset{c}{\leadsto} e'}{\texttt{run}\ e \overset{c}{\leadsto} \texttt{run}\ (e'\ c)}$$

$$\frac{x \notin fv(c) \qquad e_1 \overset{c}{\leadsto} e_1' \qquad e_2 \overset{c}{\leadsto} e_2' \qquad e \overset{c}{\leadsto} e'}{\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e \overset{c}{\leadsto} \texttt{signal}\ x\ \texttt{default}\ e_1'\ \texttt{gather}\ e_2'\ \texttt{in}\ e'}$$

$$\frac{x \notin fv(c) \qquad e_1 \overset{c}{\leadsto} e_1' \qquad e_2 \overset{c}{\leadsto} e_2'}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \overset{c}{\leadsto} \texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2'} \qquad \frac{}{\texttt{pause} \overset{c}{\leadsto} \texttt{pause;await immediate}\ c}$$

$$\frac{e_1 \overset{c}{\leadsto} e_1' \qquad e_2 \overset{c}{\leadsto} e_2'}{\texttt{present}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \overset{c}{\leadsto} \texttt{present}\ x\ \texttt{then}\ e_1'\ \texttt{else}\ (\texttt{await immediate}\ c\texttt{;}e_2')}$$

$$\texttt{await immediate}\ x \overset{c}{\leadsto} \texttt{await immediate}\ (c \wedge x)$$

$$\frac{y \notin fv(c) \quad e_1 \overset{c}{\leadsto} e_1' \quad body = \texttt{await}\ x(y)\ \texttt{in}\ \texttt{await immediate}\ c\texttt{;}e_1'}{\texttt{await}\ x(y)\ \texttt{in}\ e_1 \overset{c}{\leadsto} \texttt{await immediate}\ (c \wedge x)\texttt{;}body}$$

$$\frac{c' = c \wedge x \qquad e \overset{c'}{\leadsto} e'}{\texttt{do}\ e\ \texttt{when}\ x \overset{c}{\leadsto} \texttt{await immediate}\ c'\texttt{;}e'}$$

FIG. 6.2 – Suppression des `do/when`

Ainsi, nous devons définir $c \in S$ et $c \notin S$.

$$\begin{array}{lll} c \in S & \text{ssi} & \forall n \in c.\ n \in S \\ c \notin S & \text{ssi} & \exists n \in c.\ n \notin S \end{array}$$

Donc une configuration est dite *présente* si tous les signaux qui la composent sont présents. Elle est *absente* si au moins un des signaux est absent.

6.4.2 La préemption

Contrairement au `do/when`, la construction `do/until` n'est pas traitée par transformation dans des instructions du noyau. Nous étendons et modifions le langage des groupes afin de traiter la préemption. Nous devons donc étendre également le langage intermédiaire avec continuation :

$$p \quad ::= \qquad \ldots \mid \texttt{do}\ P\ \texttt{until}\ x(y)\ \texttt{->}\ P\ \texttt{done}.p$$

L'idée pour le traitement du `do/until` est de traduire le corps dans un groupe et d'avoir une instruction `kill` qui attende la présence du signal de préemption pour supprimer le corps. Nous ajoutons donc aux instructions le `kill` qui attend la présence de x pour effectuer la préemption et déclencher l'exécution du traitement d'échappement où y prend la valeur du signal :

$$t \quad ::= \qquad \ldots \mid \texttt{kill}\ x(y)\ \texttt{->}\ (\mathcal{G}, k)$$

Pour comprendre la nécessité de modifier la définition des groupes, il faut remarquer que la traduction que nous avons définie section 6.2.2 fait perdre la structure du programme. Tous les groupes sont définis au même niveau dans $\mathcal{G}$, on a donc perdu la notion de sous-expression. Illustrons cela sur un exemple :

```
do
  (emit x || emit y);
  emit z
until s done
```

La traduction du corps du `do/until` dans un groupe a avec la continuation k crée un groupe b dans lequel la partie gauche de la séquence est traduite. Le point d'entrée de ce programme est l'instruction $(b, 1)$ et l'environnement de groupes $\mathcal{G}$ est le suivant :

$$\mathcal{G} = \left[\begin{array}{l} \{\texttt{split}\ (b,2)\ (b,4)_1,\ (\texttt{emit x})_2,\ \texttt{term}_3,\ (\texttt{emit y})_4,\ \texttt{term}_5 \rhd^2 (a,1)\}/b; \\ \{(\texttt{emit z})_1,\ \texttt{term}_2 \rhd^1 k\}/a \end{array} \right]$$

On constate qu'il n'apparaît pas dans le groupe a que b est un sous groupe de a. Ceci pose problème pour le `do/until` car en cas de préemption il faut supprimer a et b.

La solution que nous proposons est d'ajouter dans chaque groupe l'ensemble de ses fils. Ceci permet de reconstituer l'arbre des groupes qui correspond la structure du programme. Sur l'exemple précédent cela nous donne l'environnement :

$$\mathcal{G} = \left[\begin{array}{l} \{\emptyset \mid \texttt{split}\ (b,2)\ (b,4)_1,\ (\texttt{emit x})_2,\ \texttt{term}_3,\ (\texttt{emit y})_4,\ \texttt{term}_5 \rhd^2 (a,1)\}/b; \\ \{b \mid (\texttt{emit z})_1,\ \texttt{term}_2 \rhd^1 k\}/a \end{array} \right]$$

La nouvelle définition des groupes est la suivante :

$$G ::= \{\mathcal{F} \mid \mathcal{I} \rhd^{cpt} k\} \qquad\qquad \mathcal{F} ::= \emptyset \mid id, \mathcal{F}$$

Comme nous avons modifié la définition des groupes, il faut modifier la fonction de traduction. Nous ajoutons l'ensemble $\mathcal{F}$ des noms de groupes directement créés par la traduction à sa valeur de retour.

$$Tr(P, id, i) = (\mathcal{F}, \mathcal{G}, \mathcal{I}, k, cpt, i')$$

Les modifications à apporter à la fonction de traduction sont assez simples. Illustrons ces changements sur un des cas les plus intéressants, la séquence :

$$
\begin{array}{l}
Tr(P_1\,;P_2, a, i) = \\
\quad \text{let } (\mathcal{F}_2, \mathcal{G}_2, \mathcal{I}_2, k_2, cpt_2, i_2) = Tr(P_2, a, i) \text{ in} \\
\quad \text{let } b = fresh() \text{ in} \\
\quad \text{let } (\mathcal{F}_1, \mathcal{G}_1, \mathcal{I}_1, k_1, cpt_1, i_1) = Tr(P_1, b, 1) \text{ in} \\
\quad \text{let } G_1 = \{\mathcal{F}_1 \mid \mathcal{I}_1 \rhd^{cpt_1} k_2\} \text{ in} \\
\quad (\mathcal{F}_2, [G_1/b] \cup \mathcal{G}_1 \cup \mathcal{G}_2, \mathcal{I}_2, k_1, cpt_2, i_2)
\end{array}
$$

La traduction de P_1 se faisant dans b, on ajoute $\mathcal{F}_1$ dans la définition de G_1. La traduction de P_2 se faisant quant à elle directement dans a, $\mathcal{F}_2$ est l'ensemble des noms de groupes rendu par la traduction.

Maintenant que nous avons introduit l'instruction `kill` et que nous avons modifié la définition des groupes, nous pouvons passer à la définition de la traduction du `do/until`.

$$
\begin{array}{l}
Tr(\texttt{do } P_1 \texttt{ until } x(y) \texttt{ -> } P_2 \texttt{ done}.p, a, i) = \\
\quad \text{let } (\mathcal{G}, \mathcal{I}, k, cpt, i') = Tr(p, a, i+1) \text{ in} \\
\quad \text{let } c = fresh() \text{ in} \\
\quad \text{let } (\mathcal{F}_2, \mathcal{G}_2, \mathcal{I}_2, k_2, cpt_2, i_2) = Tr(P_2, c, 1) \text{ in} \\
\quad \text{let } G_2 = \{\mathcal{F}_2 \mid \mathcal{I}_2 \rhd^{cpt_2} k\} \text{ in} \\
\quad \text{let } b = fresh() \text{ in} \\
\quad \text{let } (\mathcal{F}_1, \mathcal{G}_1, \mathcal{I}_1, k_1, cpt_1, i_1) = Tr(P_1, b, 2) \text{ in} \\
\quad \text{let } G_1 = \{\mathcal{F}_1 \mid (\texttt{kill } x(y) \texttt{ -> } ([G_2/c] \cup \mathcal{G}_2, k_2))_1, \, \mathcal{I}_1 \rhd^{cpt_1} k\} \text{ in} \\
\quad ((b, c, \mathcal{F}), [G_1/b] \cup \mathcal{G}_1 \cup \mathcal{G}, ((\texttt{split } (b,1) \ k_1)_i, \mathcal{I}), (a, i), cpt, i')
\end{array}
$$

La traduction commence par la traduction de la continuation p. Puis P_2 et P_1 sont traduits respectivement dans les nouveaux groupes c et b avec la même continuation k. L'instruction `kill` est ajoutée au groupe b. Enfin on ajoute une instruction `split` au groupe a qui va servir de point d'entrée. Cette instruction `split` permet à la fois d'exécuter P_1 et l'instruction `kill` qui contrôle l'exécution du programme. Cette instruction attend la fin d'instant et que le signal soit émis pour effectuer la préemption. Si le corps du `do/until` termine son exécution avant qu'il y ait préemption, l'instruction `kill` est supprimée.

Il nous reste enfin à décrire les règles de réaction de l'instruction `kill`.

$$\frac{eoi \notin S \qquad \mathcal{W}' = \mathcal{W} + [(a,i)/n]}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{kill } n(x) \texttt{ -> } (\mathcal{G}', k))_i^a \longrightarrow S,\, \mathcal{G},\, \mathcal{W}' \vdash \emptyset}$$

La première règle attend la fin d'instant avant de tester s'il y a préemption ou pas.

$$\frac{a \notin Dom(\mathcal{G})}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{kill } n(x) \texttt{ -> } (\mathcal{G}', k))_i^a \longrightarrow S,\, \mathcal{G},\, \mathcal{W} \vdash \emptyset}$$

La condition $a \in Dom(\mathcal{G})$ teste si le groupe a existe toujours ou pas dans l'environnement $\mathcal{G}$. S'il n'est plus dans l'environnement, cela signifie que le `do/until` a terminé l'exécution de son corps. Dans ce cas il ne peut plus y avoir de préemption, l'instruction `kill` est donc supprimée.

$$\frac{eoi \in S \qquad a \in Dom(\mathcal{G}) \qquad n \notin S \qquad \mathcal{W}' = \mathcal{W} + [(a,i)/n]}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{kill}\ n(x)\ \texttt{->}\ (\mathcal{G}',k))_i^a \longrightarrow S,\, \mathcal{G},\, \mathcal{W}' \vdash \emptyset}$$

À la fin d'instant, si le corps du `do/until` n'est pas terminé, on teste la présence du signal. S'il est absent, l'instruction `kill` est mise en attente.

Il ne nous reste plus qu'à traiter le cas où il y a préemption. L'idée de ce dernier cas est de supprimer le groupe du corps du `do/until` et ses sous-groupes ainsi que toutes les instructions appartenant à ces groupes présentes dans $\mathcal{W}$ et $\mathcal{C}$. Nous définissons la fonction $remove(a, \mathcal{G}, \mathcal{W}, \mathcal{C})$ qui supprime tout ce qui concerne a et ses sous-groupes de $\mathcal{G}$, $\mathcal{W}$ et $\mathcal{C}$.

$remove(a, \mathcal{G}, \mathcal{W}, \mathcal{C}) =$
  Set $\mathcal{G}' := \mathcal{G}$, $\mathcal{W}' := \mathcal{W}$, $\mathcal{C}' := \mathcal{C}$ in
  forAll b sous-groupe de a do
    $\mathcal{G}', \mathcal{W}', \mathcal{C}' := remove(b, \mathcal{G}', \mathcal{W}', \mathcal{C}')$
  done ;
  $(\mathcal{G}' \backslash a, \mathcal{W}' \backslash a, \mathcal{C}' \backslash a)$

Nous constatons qu'il faut avoir accès à l'ensemble $\mathcal{C}$ des instructions à exécuter à l'instant courant pour réaliser la préemption. Or, dans les règles de réactions, cette information n'est pas présente. Il faut donc ajouter $\mathcal{C}$ dans la définition des réactions :

$$S,\, \mathcal{G},\, \mathcal{W},\, \mathcal{C} \vdash t_i^a \longrightarrow S',\, \mathcal{G}',\, \mathcal{W}' \vdash \mathcal{C}'$$

Cette modification dans la forme des règles de réaction modifie légèrement l'ensemble des règles précédentes. En effet, elles doivent conserver toutes les instructions qui sont dans $\mathcal{C}$ en plus de celles qu'elles génèrent. Les règles

$$S,\, \mathcal{G},\, \mathcal{W} \vdash t_i^a \longrightarrow S',\, \mathcal{G}',\, \mathcal{W}' \vdash \mathcal{C}'$$

deviennent

$$S,\, \mathcal{G},\, \mathcal{W},\, \mathcal{C} \vdash t_i^a \longrightarrow S',\, \mathcal{G}',\, \mathcal{W}' \vdash \mathcal{C}' \cup \mathcal{C}$$

Finalement, la règle du `kill` qui traite la préemption est la suivante :

$$\frac{\begin{array}{c} eoi \in S \qquad a \in Dom(\mathcal{G}) \qquad n \in S \qquad \mathcal{G}'', \mathcal{W}', \mathcal{C}' = remove(a, \mathcal{G}, \mathcal{W}, \mathcal{C}) \\ S(n) = (d, g, p, m) \qquad v = fold\ g\ m\ d \qquad \mathcal{G}''' = \mathcal{G}''.(\mathcal{G}'[x \leftarrow v]) \end{array}}{S,\, \mathcal{G},\, \mathcal{W},\, \mathcal{C} \vdash (\texttt{kill}\ n(x)\ \texttt{->}\ (\mathcal{G}',k))_i^a \longrightarrow S,\, \mathcal{G}''',\, \mathcal{W}' \vdash k, \mathcal{C}'}$$

Dans la première ligne des prémisses, on vérifie que la préemption peut avoir lieu et on calcule le nouvel état du programme dans lequel on a supprimé le corps du `do/until`. La seconde ligne récupère la valeur du signal et prépare l'exécution du traitement d'échappement.

6.5 Conclusion

Nous avons défini une première sémantique pour l'implantation efficace du modèle réactif. Dans cette sémantique, les programmes sont décomposés en un ensemble $\mathcal{C}$ d'instructions à exécuter, des files d'attente $\mathcal{W}$ et des groupes $\mathcal{G}$ qui gardent la structure du programme. L'ordonnancement des programmes est défini par l'ordre d'activation des instructions présentes dans $\mathcal{C}$. La composition parallèle étant associative et commutative en ReactiveML, il n'y a donc aucune contrainte sur le choix des instructions dans $\mathcal{C}$.

Nous pouvons constater que dans l'activation de fin d'instant toutes les instructions en attentes sont activées. Or nous savons que les instructions `await/immediate` ne peuvent pas évoluer alors que les instructions `present` sont réduites. Dans l'implantation de la sémantique, nous faisons la distinction entre ces deux types d'attente. Ainsi, nous séparons l'environnement de files d'attente $\mathcal{W}$ en deux environnements $\mathcal{W}_a$ et $\mathcal{W}_p$ qui conservent respectivement les instructions `await/immediate` et `present`. Comme cela, à la fin de l'instant, nous savons que nous pouvons conserver tel quel l'environnement $\mathcal{W}_a$ alors qu'il faut faire réagir l'environnement $\mathcal{W}_p$. Enfin pour l'instruction `await/in`, tant que le signal testé est absent, l'instruction est enregistrée dans $\mathcal{W}_a$ et lorsque le signal est émis, l'instruction attend la fin d'instant dans l'environnement $\mathcal{W}_p$. Dans cette sémantique, nous avons donc de l'attente passive intra-instants et inter-instants.

Cette sémantique étant une sémantique à petits pas, elle s'étend simplement avec la construction `await/one` et les références. Les règles pour le `await/one` sont les suivantes :

$$\frac{n \in S \qquad \mathcal{G}'' = \mathcal{G}.(\mathcal{G}'[x \leftarrow one(S^v(n))])}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{await one } n\texttt{(}x\texttt{) in } (\mathcal{G}', k))^a_i \longrightarrow S,\, \mathcal{G},\, \mathcal{W} \vdash k}$$

$$\frac{n \notin S \qquad \mathcal{W}' = \mathcal{W} + [(a,i)/n]}{S,\, \mathcal{G},\, \mathcal{W} \vdash (\texttt{await one } n\texttt{(}x\texttt{) in } (\mathcal{G}', k))^a_i \longrightarrow S,\, \mathcal{G},\, \mathcal{W}' \vdash \emptyset}$$

Pour les références, il faut ajouter un environnement M qui permet de conserver l'état de la mémoire et il faut étendre la sémantique de ML avec les références.

Résumé

Les trois points principaux pour réaliser un interprète efficace du modèle réactif sont : (1) l'attente passive des signaux, (2) la réduction du temps de parcours de la structure du programme et (3) avoir la composition parallèle associative et commutative. L'attente passive est certainement le point le plus important. Elle permet d'exécuter les instructions uniquement lorsque le signal dont elles dépendent est émis. Ainsi cela évite de tester systématiquement des instructions qui ne peuvent pas évoluer.

La sémantique Glouton intègre les trois points précédents. Elle est basée sur l'idée qu'il faut toujours "aller de l'avant" en gardant les tests de signaux qui sont absents dans des files d'attente. La sémantique n'est pas donnée directement sur le langage source ReactiveML, mais sur un format que nous avons appelé les groupes. Dans cette sémantique, les programmes sont décomposés en un ensemble $\mathcal{C}$ d'instructions à exécuter, des files d'attente $\mathcal{W}$ et des groupes $\mathcal{G}$ qui gardent la structure du programme. L'exécution d'un instant consiste à exécuter toutes les instructions de $\mathcal{C}$.

Chapitre 7

Une sémantique efficace pour le modèle réactif II : L_k

La sémantique que nous présentons ici est beaucoup plus proche de l'implantation actuelle de ReactiveML que la sémantique du chapitre précédent. Elle se base sur l'utilisation de files d'attente et de continuations mais nous n'utilisons plus les groupes. Cette fois ci, nous donnons la sémantique complète du langage avec le `let/and/in`.

La structure de ce chapitre est la même que celle du chapitre précédent. Nous commençons par présenter la sémantique d'un sous-ensemble du langage dans lequel il n'y a pas d'opération de suspension ni de préemption. Puis, nous ajoutons ces deux constructions.

7.1 Le langage avec continuation L_k

Le langage L_k sur lequel nous définissons la sémantique a la syntaxe abstraite suivante :

$$
\begin{array}{rcl}
k & ::= & \texttt{end} \mid \kappa \mid e.k \mid \texttt{present}\ e\ \texttt{then}\ k\ \texttt{else}\ k \\
 & & \mid \texttt{await immediate}\ e.k \mid \texttt{await}\ e(x)\ \texttt{in}\ k \mid \texttt{run}\ e.k \\
 & & \mid \texttt{split}\ (\lambda x.(k,k)) \mid \texttt{join}\ x\ i.k \mid \texttt{def}\ x\ \texttt{and}\ y\ \texttt{in}\ k \\
 & & \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ k \\
 & & \\
e & ::= & x \mid c \mid (e,e) \mid \lambda x.e \mid e\,e \mid \texttt{rec}\,x = e \mid \texttt{pre}\ e \mid \texttt{pre ?}e \mid \texttt{emit}\ e\ e \\
 & & \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \mid \texttt{process}\ \Lambda\kappa.k
\end{array}
$$

Dans ce langage, κ est une variable qui sera substituée par une continuation (k) et i peut prendre la valeur 1 ou 2. L'expression `end` marque la fin du programme. Il ne faut pas la confondre avec l'instruction `term` du chapitre précédent qui indique la fin d'une branche parallèle. Comme une expression se termine nécessairement par un `end`, la continuation de l'expression `present` est propagée dans ses deux branches. En effet, si on avait l'expression

$$(\texttt{present}\ e\ \texttt{then}\ k_1\ \texttt{else}\ k_2).k_3$$

la continuation k_3 ne pourrait jamais être exécutée car k_1 et k_2 se terminent avec un `end` qui marque la fin du programme. Les expressions `split`, `join` et `def` servent à coder le `let/and/in`. Le `split` commence l'exécution du parallèle, le `join` synchronise la terminaison et le `def` récupère les valeurs avant d'exécuter la continuation. La variable x, introduite dans le `split` et

$C_k[e] = C[e].k$ si $0 \vdash e$ $\qquad C_k[\texttt{run } e] = \texttt{run } C[e].k$ $\qquad C_k[e_1\,\texttt{;}\,e_2] = C_{(C_k[e_2])}[e_1]$

$C_k[\texttt{present } e \texttt{ then } e_1 \texttt{ else } e_2] = \texttt{present } C[e] \texttt{ then } C_k[e_1] \texttt{ else } C_k[e_2]$

$C_k[\texttt{await immediate } e] = \texttt{await immediate } C[e].k$

$C_k[\texttt{await } e_1(x) \texttt{ in } e_2] = \texttt{await } C[e_1](x) \texttt{ in } C_k[e_2]$ avec $x \notin fv(k)$

$C_k[\texttt{signal } x \texttt{ default } e_1 \texttt{ gather } e_2 \texttt{ in } e] =$
$\quad \texttt{signal } x \texttt{ default } C[e_1] \texttt{ gather } C[e_2] \texttt{ in } C_k[e]$ avec $x \notin fv(k)$

$C_k[\texttt{let } x \texttt{ = } e_1 \texttt{ and } y \texttt{ = } e_2 \texttt{ in } e] = \texttt{split } (\lambda z.(C_{(\texttt{join } z\ 1.k')}[e_1], C_{(\texttt{join } z\ 2.k')}[e_2]))$
avec $k' = \texttt{def } x \texttt{ and } y \texttt{ in } C_k[e]$ et $z \notin fv(e_1) \cup fv(e_2) \cup fv(k)$

FIG. 7.1 – Traduction de REACTIVEML vers L_k.

utilisée dans le `join`, est une variable partagée par les deux branches parallèles. Elle sert à la synchronisation. Enfin, le paramètre i du `join` différencie la branche gauche de la branche droite du parallèle. Pour les expressions instantanées (e), il faut noter que la définition de processus est paramétrée par sa continuation.

Le langage L_k est différent du format intermédiaire que nous avons utilisé pour la transformation vers les *groupes* dans la section 6.2.2. Comme nous devons récupérer les valeurs des branches parallèles à la fin de leur exécution, nous ne pouvons pas attendre la séquence pour faire la synchronisation de la terminaison. À la place du couple `term`/séquence, nous avons l'instruction `join` avec sa continuation.

Traduction

La fonction $C_k[e]$ traduit l'expression REACTIVEML e dans L_k avec une transformation CPS [4] (Continuation-Passing Style). Cette fonction, définie figure 7.1, est paramétrée par une continuation k. Pour les expressions instantanées (telles que $0 \vdash e$), la fonction de traduction $C[e]$ est utilisée. Cette fonction propage l'information de traduction. Les traductions de définitions de processus sont paramétrées par des variables κ.

$$C[x] = x \qquad C[c] = c \qquad C[(e_1, e_2)] = (C[e_1], C[e_2]) \qquad C[e_1\, e_2] = C[e_1]\, C[e_2]$$

$$C[\lambda x.e] = \lambda x.C[e] \qquad C[\texttt{rec}\, x \texttt{=} e] = \texttt{rec}\, x \texttt{=} C[e] \qquad C[\texttt{pre } e] = \texttt{pre } C[e]$$

$$C[\texttt{pre ?}e] = \texttt{pre ?}C[e] \qquad C[\texttt{emit } e_1\ e_2] = \texttt{emit } C[e_1]\ C[e_2]$$

$$C[\texttt{signal } x \texttt{ default } e_1 \texttt{ gather } e_2 \texttt{ in } e] = \texttt{signal } x \texttt{ default } C[e_1] \texttt{ gather } C[e_2] \texttt{ in } C[e]$$

$$C[\texttt{process } e] = \texttt{process } \Lambda\kappa.C_\kappa[e]$$

Il n'y a pas de séquence en L_k puisqu'elle est codée avec les continuations. La branche droite de e_1;e_2 est traduite avec la continuation k. Le résultat de cette traduction est utilisé comme continuation de la branche gauche.

Pour le `present`, nous traduisons les deux branches avec la continuation k. Pour les lieurs `await/in` et `signal/in`, il faut vérifier que le nom qui est introduit ne capture pas une variable dans la continuation.

Enfin, voyons la traduction du `let/and/in`. L'instruction `split` déclenche l'exécution des deux branches e_1 et e_2. Ces branches ont pour continuation une instruction `join` qui attend la terminaison de l'autre branche. La continuation des `join` est une instruction `def` qui récupère instantanément les valeurs calculées par les deux branches.

7.2 Sémantique

Nous présentons une sémantique petits pas de L_k. Cette sémantique utilise une approche similaire à Glouton. Elle se présente en deux étapes. Pendant l'instant, il y a un ensemble $\mathcal{C}$ d'expressions à exécuter. Lorsqu'une expression de $\mathcal{C}$ est bloquée sur le test d'un signal, elle est enregistrée dans une file d'attente de $\mathcal{W}$. À la fin d'instant, lorsque $\mathcal{C}$ est vide, les expressions `present` et `await` qui sont dans $\mathcal{W}$ sont traitées pour préparer la réaction de l'instant suivant.

La règle de réaction d'un instant est de la forme :

$$S,\, J,\, \mathcal{W} \vdash \mathcal{C} \Longrightarrow S',\, J',\, \mathcal{W}' \vdash \mathcal{C}'$$

S est l'environnement de signaux défini comme dans les sémantiques précédentes. J est un environnement qui est utilisé pour la synchronisation de la terminaison des branches parallèles. Il associe à chaque adresse j, un compteur cpt qui indique le nombre de branches parallèles encore actives et une paire de valeurs v_{opt} qui contient la valeur calculée par chaque branche. v_{opt} est égal à $\perp$ tant que le calcul de la branche parallèle n'est pas terminé.

$$J ::= [\ldots, (cpt, (v_{opt}, v_{opt}))/j, \ldots] \qquad\qquad v_{opt} ::= v \mid \perp$$

Nous définissons les fonctions d'accès J^{cpt}, J^{v}, J^{v_1} et J^{v_2} dans un environnement J tel que si $J(j) = (n, (v_1, v_2))$ alors :

$$J^{cpt}(j) = n \qquad J^{v}(j) = (v_1, v_2) \qquad J^{v_1}(j) = v_1 \qquad J^{v_2}(j) = v_2$$

L'ensemble des expressions à exécuter et l'ensemble des files d'attentes sont définis par :

$$\mathcal{C} ::= \emptyset \mid \texttt{<} k \texttt{,}\, v \texttt{>}, \mathcal{C} \qquad\qquad \mathcal{W} ::= [\ldots, \mathcal{C}/n, \ldots]$$

Les informations enregistrées dans $\mathcal{C}$ et $\mathcal{W}$ sont des paires : (expression k, valeur v). k est l'expression à exécuter et v le résultat de l'expression calculée avant k. Nous devons toujours garder la valeur v afin de coder le `let/and/in`.

La réaction d'un instant est donc définie par :

$$\frac{S,\, J,\, \mathcal{W} \vdash \mathcal{C} \Longrightarrow S',\, J',\, \mathcal{W}' \quad O = next(S') \quad O \vdash \mathcal{W}' \Longrightarrow_{eoi} \mathcal{W}'' \mid \mathcal{C}''}{S,\, J,\, \mathcal{W} \vdash \mathcal{C} \Longrightarrow S',\, J',\, \mathcal{W}'' \vdash \mathcal{C}''}$$

Détaillons chaque étape de la réaction d'un instant. La réaction *pendant* l'instant $S,\ J,\ \mathcal{W} \vdash \mathcal{C} \Longrightarrow S',\ J',\ \mathcal{W}'$ est définie par la réaction de toutes les instructions qui sont dans $\mathcal{C}$. On a ainsi :

$$\frac{}{S,\ J,\ \mathcal{W} \vdash \emptyset \Longrightarrow S,\ J,\ \mathcal{W}}$$

$$\frac{\begin{array}{c} S,\ J,\ \mathcal{W} \vdash \texttt{<}\, e\texttt{,}\ v \,\texttt{>} \longrightarrow S_1,\ J_1,\ \mathcal{W}_1 \vdash \mathcal{C}_1 \\ \mathcal{W}_2, \mathcal{C}_2 = wakeUp(S\#S_1, \mathcal{W}_1) \qquad S_1,\ J_1,\ \mathcal{W}_2 \vdash \mathcal{C}, \mathcal{C}_1, \mathcal{C}_2 \Longrightarrow S',\ J',\ \mathcal{W}' \end{array}}{S,\ J,\ \mathcal{W} \vdash \texttt{<}\, e\texttt{,}\ v \,\texttt{>}, \mathcal{C} \Longrightarrow S',\ J',\ \mathcal{W}'}$$

Comme pour Glouton, la fonction *wakeUp* débloque dans $\mathcal{W}_1$ les expressions associées à un signal émis par la réaction de e. La fonction *wakeUp* est définie au chapitre 6.3.1 (page 98).

La réaction de chaque expression est définie figures 7.2 et 7.3. Commentons ces règles :

- La réaction de $e.k$ évalue l'expression ML e en une valeur v', et demande l'exécution de la continuation k dans l'instant. k est enregistré avec la valeur v' dans $\mathcal{C}$.
- L'instruction `present` exécute instantanément sa branche `then` si le signal testé est présent. Si le signal est absent, l'instruction `present` est enregistrée dans la file d'attente de n.
- Le comportement de `await/immediate` est similaire au `present`.
- La construction `await/in` se place toujours en attente dans $\mathcal{W}$. Elle ne peut pas se réduire avant la fin d'instant.
- `run` $e.k$ évalue e en une définition de processus paramétrée par la variable κ. Cette variable κ est substituée par la continuation du `run` (k) dans le corps du processus. Le nouveau processus est alors placé dans $\mathcal{C}$.
- La règle pour `signal` alloue un nouveau signal n dans S et substitue x par n dans la continuation. Cette nouvelle continuation doit être exécutée dans l'instant courant.
- `split` crée un nouveau point de synchronisation j dans l'environnement J. Ce point est initialisé à $(2, \perp, \perp)$. Le chiffre 2 indique que les deux branches du parallèle sont actives et les $\perp$ que les valeurs de terminaison ne sont pas encore calculées.
  La variable de synchronisation x est substituée par j dans les deux continuations de `split`. Ainsi les deux branches parallèles partagent le même point de synchronisation dans J. Finalement, les deux continuations sont à exécuter dans l'instant courant.
- L'expression `join` indique la terminaison d'une branche d'un parallèle. On applique donc la fonction *join* sur l'environnement J. Cette fonction décrémente le compteur de branches actives et mémorise la valeur calculée par la branche. L'environnement J' est donc égal à $join(J, j, i, v)$ si et seulement si :

$$\forall j' \ : \ J'(j') = \begin{cases} J(j') & \text{si } j' \neq j \\ (J^{cpt}(j) - 1, (v, J^{v_2}(j))) & \text{si } j' = j \text{ et } i = 1 \\ (J^{cpt}(j) - 1, (J^{v_1}(j), v)) & \text{si } j' = j \text{ et } i = 2 \end{cases}$$

  Les deux cas du `join` distinguent le cas où il reste une branche active du cas où les deux branches ont terminé leur exécution. Dans le premier cas, il n'y a plus rien à faire après le `join`. Dans le second, la continuation est exécutée en transmettant les valeurs calculées par les deux branches du parallèle.
- Enfin, l'instruction `def` qui se trouve toujours derrière un `join` est donc associée avec une paire de valeurs. Ces valeurs substituent x et y dans la continuation k.

$$\frac{e/S \Downarrow v'/S'}{S,\, J,\, \mathcal{W} \vdash \mathtt{<} e.k \,\mathtt{,}\, v \mathtt{>} \longrightarrow S',\, J,\, \mathcal{W} \vdash \mathtt{<} k \,\mathtt{,}\, v' \mathtt{>}}$$

$$\frac{e/S \Downarrow n/S' \qquad n \in S'}{S,\, J,\, \mathcal{W} \vdash \mathtt{<}\mathtt{present}\ e\ \mathtt{then}\ k_1\ \mathtt{else}\ k_2 \,\mathtt{,}\, v \mathtt{>} \longrightarrow S',\, J,\, \mathcal{W} \vdash \mathtt{<} k_1 \,\mathtt{,}\, \mathtt{()} \mathtt{>}}$$

$$\frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad \mathit{self} = \mathtt{present}\ n\ \mathtt{then}\ k_1\ \mathtt{else}\ k_2}{S,\, J,\, \mathcal{W} \vdash \mathtt{<}\mathtt{present}\ e\ \mathtt{then}\ k_1\ \mathtt{else}\ k_2 \,\mathtt{,}\, v \mathtt{>} \longrightarrow S',\, J,\, \mathcal{W} + [\mathtt{<} \mathit{self} \,\mathtt{,}\, v \mathtt{>}/n] \vdash \emptyset}$$

$$\frac{e/S \Downarrow n/S' \qquad n \in S'}{S,\, J,\, \mathcal{W} \vdash \mathtt{<}\mathtt{await\ immediate}\ e.k \,\mathtt{,}\, v \mathtt{>} \longrightarrow S',\, J,\, \mathcal{W} \vdash \mathtt{<} k \,\mathtt{,}\, \mathtt{()} \mathtt{>}}$$

$$\frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad \mathit{self} = \mathtt{await\ immediate}\ n.k}{S,\, J,\, \mathcal{W} \vdash \mathtt{<}\mathtt{await\ immediate}\ e.k \,\mathtt{,}\, v \mathtt{>} \longrightarrow S',\, J,\, \mathcal{W} + [\mathtt{<} \mathit{self} \,\mathtt{,}\, v \mathtt{>}/n] \vdash \emptyset}$$

$$\frac{e/S \Downarrow n/S' \qquad \mathit{self} = \mathtt{await}\ n(x)\ \mathtt{in}\ k}{S,\, J,\, \mathcal{W} \vdash \mathtt{<}\mathtt{await}\ e(x)\ \mathtt{in}\ k \,\mathtt{,}\, v \mathtt{>} \longrightarrow S',\, J,\, \mathcal{W} + [\mathtt{<} \mathit{self} \,\mathtt{,}\, v \mathtt{>}/n] \vdash \emptyset}$$

$$\frac{e/S \Downarrow \mathtt{process}\ \Lambda\kappa.k_{\mathit{body}}/S' \qquad k' = k_{\mathit{body}}[\kappa \leftarrow k]}{S,\, J,\, \mathcal{W} \vdash \mathtt{<}\mathtt{run}\ e.k \,\mathtt{,}\, v \mathtt{>} \longrightarrow S',\, J,\, \mathcal{W} \vdash \mathtt{<} k' \,\mathtt{,}\, \mathtt{()} \mathtt{>}}$$

$$\frac{\begin{array}{c} e_1/S \Downarrow v_1/S_1 \quad e_2/S_1 \Downarrow v_2/S_2 \\ n \notin \mathit{Dom}(S_2) \quad S' = S_2[(v_1, v_2, (\mathit{false}, v_1), \emptyset)/n] \quad k' = k[x \leftarrow n] \end{array}}{S,\, J,\, \mathcal{W} \vdash \mathtt{<}\mathtt{signal}\ x\ \mathtt{default}\ e_1\ \mathtt{gather}\ e_2\ \mathtt{in}\ k \,\mathtt{,}\, v \mathtt{>} \longrightarrow S',\, J,\, \mathcal{W} \vdash \mathtt{<} k' \,\mathtt{,}\, \mathtt{()} \mathtt{>}}$$

FIG. 7.2 – Réaction de L_k.

$$\frac{j \notin Dom(J) \qquad J' = J[(2,(\bot,\bot))/j] \qquad k_1' = k_1[x \leftarrow j] \qquad k_2' = k_2[x \leftarrow j]}{S,\ J,\ \mathcal{W} \vdash \texttt{<split}\ \lambda x.(k_1,k_2)\texttt{,}\ v\texttt{>} \longrightarrow S,\ J',\ \mathcal{W} \vdash \texttt{<}k_1'\texttt{, ()>,<}k_2'\texttt{, ()>}}$$

$$\frac{J' = join(J,j,i,v) \qquad J'^{cpt}(j) > 0}{S,\ J,\ \mathcal{W} \vdash \texttt{<join}\ j\ i.k\texttt{,}\ v\texttt{>} \longrightarrow S,\ J',\ \mathcal{W} \vdash \emptyset}$$

$$\frac{J' = join(J,j,i,v) \qquad J'^{cpt}(j) = 0 \qquad J'^{v}(j) = (v_1,v_2)}{S,\ J,\ \mathcal{W} \vdash \texttt{<join}\ j\ i.k\texttt{,}\ v\texttt{>} \longrightarrow S,\ J',\ \mathcal{W} \vdash \texttt{<}k\texttt{,}\ (v_1,v_2)\texttt{>}}$$

$$\frac{k' = k[x \leftarrow v_1, y \leftarrow v_2]}{S,\ J,\ \mathcal{W} \vdash \texttt{<def}\ x\ \texttt{and}\ y\ \texttt{in}\ k\texttt{,}\ (v_1,v_2)\texttt{>} \longrightarrow S,\ J,\ \mathcal{W} \vdash \texttt{<}k'\texttt{, ()>}}$$

FIG. 7.3 – Réaction de L_k.

Nous définissons maintenant la réaction de fin d'instant. Cette étape commence par le calcul du statut et des valeurs associées aux signaux avec la fonction $next(S)$. Cette fonction est définie chapitre 3.3.2 (page 51). Puis, les expressions présentes dans $\mathcal{W}$ sont parcourues pour construire les ensembles $\mathcal{C}$ et $\mathcal{W}$ du début de l'instant suivant.

$$\frac{}{O \vdash \emptyset \Longrightarrow_{eoi} \mathcal{W} \mid \mathcal{C}} \qquad \frac{O \vdash \texttt{<}e\texttt{,}\ v\texttt{>} \longrightarrow_{eoi} \mathcal{W}' \mid \mathcal{C}' \qquad O \vdash \mathcal{W} \Longrightarrow_{eoi} \mathcal{W}'' \mid \mathcal{C}''}{O \vdash \texttt{<}e\texttt{,}\ v\texttt{>}, \mathcal{W} \Longrightarrow_{eoi} \mathcal{W}', \mathcal{W}'' \mid \mathcal{C}', \mathcal{C}''}$$

Seules les expressions qui testent la présence d'un signal sont présentes à la fin d'instant car toutes les autres peuvent être résolues pendant l'instant. Les règles de réactions pour chacune des expressions sont définies figure 7.4.

La réaction de `present` donne la branche k_2 à exécuter à l'instant suivant car le signal ne peut pas être présent sinon l'expression aurait été réduite pendant l'instant. De même pour `await/immediate`, l'expression reste nécessairement en attente.

Le `await/in` teste la présence du signal. Si le signal est absent, il reste en attente. Mais s'il est présent, il substitue x par la valeur associée au signal dans k. Cette continuation sera exécutée à l'instant suivant.

7.3 Ajout des constructions de contrôle

Comme pour GLOUTON, la sémantique de L_k ne garde pas la structure du terme pendant l'exécution. Nous sommes donc confrontés au même problème d'implantation du `do/when` et du `do/until` (voir chapitre 6.4). Il faut pouvoir retrouver le contexte de suspension et de préemption de chaque expression.

$$O \vdash \texttt{<present } n \texttt{ then } k_1 \texttt{ else } k_2\texttt{,}\, v\texttt{>} \longrightarrow_{eoi} \emptyset \mid \texttt{<}k_2\texttt{, ()>}$$

$$O \vdash \texttt{<await immediate } n.k\texttt{,}\, v\texttt{>} \longrightarrow_{eoi} \texttt{<await immediate } n.k\texttt{,}\, v\texttt{>} \mid \emptyset$$

$$\frac{O(n) = (\mathit{false}, v')}{O \vdash \texttt{<await } n(x) \texttt{ in } k\texttt{,}\, v\texttt{>} \longrightarrow_{eoi} \texttt{<await } n(x) \texttt{ in } k\texttt{,}\, v\texttt{>} \mid \emptyset}$$

$$\frac{O(n) = (\mathit{true}, v') \qquad k' = k[x \leftarrow v']}{O \vdash \texttt{<await } n(x) \texttt{ in } k\texttt{,}\, v\texttt{>} \longrightarrow_{eoi} \emptyset \mid \texttt{<}k'\texttt{, ()>}}$$

FIG. 7.4 – Réaction de fin d'instant de L_k.

La solution que nous proposons ici est d'introduire un environnement T qui conserve l'imbrication des constructions de suspension et de préemption du programme et d'annoter les expressions du langage par les étiquettes de T. Nous appelons *arbre de contrôle* l'environnement T car l'imbrication des `do/when` et des `do/until` définit un arbre n-aire.

Nous ajoutons au langage L_k les nouvelles expressions `start_when` et `start_until` qui représentent l'entrée dans un bloc contrôlé. Leur exécution pendant l'instant ajoute un nœud dans l'arbre de contrôle. Les règles de réaction pendant l'instant des autres expressions ne sont pas modifiées. Le changement se trouve dans la réaction de fin d'instant. Au lieu de construire les ensembles $\mathcal{C}$ et $\mathcal{W}$ pour le prochain instant, les expressions s'enregistrent dans l'environnement T pour retrouver la structure du programme. Nous pouvons alors effectuer la préemption et trouver les expressions à exécuter à l'instant suivant. La fin d'un bloc contrôlé est marquée par l'exécution d'une expression `end_when` ou `end_until`.

Commençons par définir l'environnement T avant de voir comment modifier la sémantique L_k pour ajouter le `do/when` et le `do/until`.

Définition 6 (arbre de contrôle)
Un arbre de contrôle T *est une fonction qui à un nom* $ctrl$ *associe un quintuplet* $(Kind, n, \mathcal{C}, ctrl, \mathcal{F})$ *que nous appelons* nœud *:*

$$\begin{array}{lcl} T & ::= & [\ldots,\ (Kind_i, n_i, \mathcal{C}_i, ctrl_i, \mathcal{F}_i)/ctrl_i,\ \ldots] \\ Kind & ::= & \texttt{When} \mid \texttt{Until}(\lambda x.k) \\ \mathcal{F} & ::= & \emptyset \mid ctrl, \mathcal{F} \end{array}$$

$Kind$ indique la nature du nœud : `When` *s'il est issu d'un* `do/when` *ou* $\texttt{Until}(\lambda x.k)$ *pour un* `do/until`*. Dans le cas de la préemption, $\lambda x.k$ est la fonction de traitement d'échappement. Le second élément du quintuplet n est le nom du signal sur lequel porte le contrôle. $\mathcal{C}$ est un ensemble d'expressions L_k qui dépendent de ce nœud de contrôle. Enfin $ctrl$ et $\mathcal{F}$ servent à coder la structure d'arbre. $ctrl$ est le nœud père et $\mathcal{F}$ est l'ensemble des fils.*

Les fonctions T^{Kind}, T^n, $T^{\mathcal{C}}$, T^{ctrl} et $T^{\mathcal{F}}$ sont les fonctions d'accès à chacune des composante d'un nœud.

La fonction $new\_node(T, ctrl, Kind, n)$ ajoute un nouveau nœud dans l'environnement T. Ce nœud est un fils de $ctrl$. La fonction retourne $ctrl'$ l'étiquette du nœud créé et le nouvel environnement T'. Dans T', le nœud $ctrl'$ est ajouté et le champ $T^{\mathcal{F}}(ctrl)$ est modifié.

$$ctrl', T' = new\_node(T, ctrl, Kind, n)$$

$$\text{si } ctrl' \notin Dom(T) \text{ et } T'(ctrl'') = \begin{cases} T(ctrl'') \text{ si } ctrl'' \neq ctrl \text{ et } ctrl'' \neq ctrl' \\ (Kind_{ctrl}, n_{ctrl}, \mathcal{C}_{ctrl}, ctrl_{ctrl}, (\mathcal{F}_{ctrl}, ctrl')) \text{ si } ctrl'' = ctrl \\ (Kind, n, \emptyset, ctrl, \emptyset) \text{ si } ctrl'' = ctrl' \end{cases}$$

Nous introduisons les notations suivantes :
- Si T est en arbre et $ctrl$ est un nœud, on notera $T - ctrl$ le résultat de la suppression du sous-arbre de racine $ctrl$ de l'arbre T.
- L'opération $T + [\texttt{<}k\texttt{,}\ v\texttt{>}/ctrl]$ ajoute la paire $\texttt{<}k\texttt{,}\ v\texttt{>}$ à l'ensemble $T^{\mathcal{C}}(ctrl)$. À l'inverse, l'opération $T - [T^{\mathcal{C}}(ctrl)/ctrl]$ met l'ensemble $\mathcal{C}$ du nœud $ctrl$ à vide.

Étudions maintenant les modifications apportées à L_k et les nouvelles expressions ajoutées.

$$\begin{array}{lll} k & ::= & \ldots \\ & & \mid \texttt{present}_{ctrl}\ e\ \texttt{then}\ k\ \texttt{else}\ k \mid \texttt{await immediate}_{ctrl}\ e.k \\ & & \mid \texttt{await}_{ctrl}\ e(x)\ \texttt{in}\ k \mid \texttt{run}_{ctrl}\ e.k \\ & & \mid \texttt{start\_until}_{ctrl}\ e\ (\lambda ctrl.k)\ (\lambda x.k) \mid \texttt{start\_when}_{ctrl}\ e\ (\lambda ctrl.k) \\ & & \mid \texttt{end\_until}\ ctrl.k \mid \texttt{end\_when}\ ctrl.k \mid \texttt{when}_{ctrl}\ e\ ctrl \\ \\ e & ::= & \ldots \mid \texttt{process}\ \Lambda\kappa, ctrl.k \end{array}$$

Les expressions réagissant à la fin d'instant doivent être annotées par leur nœud de contrôle. L'expression `run` doit aussi connaître son nœud de contrôle pour transmettre cette information au processus qu'elle exécute. Enfin les expressions `start_until`, `end_until` et `start_when`, `end_when` sont ajoutées pour marquer le début et la fin des `do/until` et des `do/when`. L'expression `when` est ajoutée pour garder l'exécution du corps des `do/when`.

Les deux expressions `start_` sont paramétrées par e le signal de contrôle et $(\lambda ctrl.k)$ le corps de la construction de contrôle. `start_until` est également paramétré par $(\lambda x.k)$ la fonction de traitement d'échappement. L'étiquette du nœud père se trouve en indice des deux expressions.

Afin de pouvoir générer ces nouvelles expressions, il faut modifier la fonction de compilation. Elle doit être paramétrée par le nœud de contrôle en plus de la continuation. Sa définition est donnée figure 7.5.

La traduction du `do/when` commence par l'expression `start_when` et la traduction du corps se fait dans un nouveau nœud de contrôle $ctrl'$ avec la continuation `end_when`. La compilation du `do/until` suit le même schéma, elle ajoute en plus la compilation du traitement d'échappement.

La sémantique de la réaction pendant l'instant est définie figure 7.6. Nous donnons les règles seulement pour le `run` et les nouvelles expressions car les autres restent inchangées.
- Le `run` fournit au processus qu'il exécute sa continuation et son contexte de contrôle.
- L'exécution de `start_until` crée un nœud `Until` dans l'environnement T et demande l'exécution de son corps dans lequel $ctrl'$ a été remplacé par $ctrl''$.

$$C_{k,ctrl}[\texttt{run}\ e] = \texttt{run}_{ctrl}\ C[e].k$$

$$C_{k,ctrl}[\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2] = \texttt{present}_{ctrl}\ C[e]\ \texttt{then}\ C_{k,ctrl}[e_1]\ \texttt{else}\ C_{k,ctrl}[e_2]$$

$$C_{k,ctrl}[\texttt{await immediate}\ e] = \texttt{await immediate}_{ctrl}\ C[e].k$$

$$C_{k,ctrl}[\texttt{await}\ e_1(x)\ \texttt{in}\ e_2] = \texttt{await}_{ctrl}\ C[e_1](x)\ \texttt{in}\ C_{k,ctrl}[e_2] \quad \text{avec} \quad x \notin fv(k)$$

$$C_{k,ctrl}[\texttt{do}\ e_1\ \texttt{when}\ e] = \texttt{start\_when}_{ctrl}\ C[e]\ (\lambda ctrl'.C_{\texttt{end\_when}\ ctrl'.k,ctrl'}[e_1])$$
$$\text{avec} \quad ctrl' \notin fv(k) \cup fv(e_1)$$

$$C_{k,ctrl}[\texttt{do}\ e_1\ \texttt{until}\ e(x)\ \texttt{->}\ e_2\ \texttt{done}] =$$
$$\texttt{start\_until}_{ctrl}\ C[e]\ (\lambda ctrl'.C_{\texttt{end\_until}\ ctrl'.k,ctrl'}[e_1])\ (\lambda x.C_{k,ctrl}[e_2])$$
$$\text{avec} \quad ctrl' \notin fv(k) \cup fv(e_1)$$

$$C[\texttt{process}\ e] = \texttt{process}\ \Lambda\kappa, ctrl.C_{\kappa,ctrl}[e]$$

FIG. 7.5 – Traduction de REACTIVEML vers L_k avec structures de contrôle.

- L'expression `end_until` supprime le nœud *ctrl* de l'environnement T et demande l'exécution sa continuation.
- L'entrée dans un bloc `do/when` crée un nœud `When`. Le corps n'est pas exécuté instantanément. Il est enregistré dans l'arbre de contrôle. Il est remplacé par une expression `when` dans l'ensemble des expressions à exécuter instantanément.
  L'expression `when` attend que le signal soit émis pour débloquer le corps du `do/when` qui attend dans l'arbre de contrôle.

Il nous reste maintenant à définir la réaction de fin d'instant. Cette étape est complètement différente du cas sans suspension ni préemption. Elle commence par le calcul de l'environnement $O = next(S)$. Puis toutes les instructions de $\mathcal{W}$ sont parcourues pour reconstruire l'arbre du programme avec une réaction $T, O \vdash \mathcal{W} \Longrightarrow_{eoi} T'$. Enfin, les préemptions sont effectuées et le nouvel ensemble des expressions à exécuter est calculé avec la fonction $next(O, T, ctrl)$.

La réaction $T, O \vdash \mathcal{W} \Longrightarrow_{eoi} T'$ est définie par les deux règles suivantes :

$$\frac{}{T, O \vdash \emptyset \Longrightarrow_{eoi} T} \qquad \frac{T, O \vdash \texttt{<}e, v\texttt{>} \longrightarrow_{eoi} T' \qquad T', O \vdash \mathcal{W} \Longrightarrow_{eoi} T''}{T, O \vdash \texttt{<}e, v\texttt{>}, \mathcal{W} \Longrightarrow_{eoi} T''}$$

Les règles pour chaque expressions sont données figure 7.7. Elles calculent les mêmes résidus que ceux donnés par les règles de la figure 7.4. Mais les expressions sont ici placées dans T au lieu d'être mises dans $\mathcal{W}$ ou $\mathcal{C}$.

Il nous reste enfin à définir la fonction $next(O, T, ctrl)$ qui retourne un nouvel arbre T' et l'ensemble $\mathcal{C}$ des expressions à exécuter par le sous-arbre étiqueté par *ctrl*.

$$\frac{e/S \Downarrow \texttt{process}\ \Lambda\kappa, ctrl'.k_{body}/S' \qquad k' = k_{body}[\kappa \leftarrow k, ctrl' \leftarrow ctrl]}{S,\ J,\ T,\ \mathcal{W} \vdash \texttt{<}\texttt{run}_{ctrl}\ e.k\texttt{,}\ v\texttt{>} \longrightarrow S',\ J,\ T,\ \mathcal{W} \vdash \texttt{<}k'\texttt{,}\ \texttt{()}\texttt{>}}$$

$$\frac{e/S \Downarrow n/S' \qquad ctrl'', T' = new\_node(T, ctrl, \texttt{Until}(\lambda x.k'), n) \qquad k'' = k[ctrl' \leftarrow ctrl'']}{S,\ J,\ T,\ \mathcal{W} \vdash \texttt{<}\texttt{start\_until}_{ctrl}\ e\ (\lambda ctrl'.k)\ (\lambda x.k')\texttt{,}\ v\texttt{>} \longrightarrow S',\ J,\ T',\ \mathcal{W} \vdash \texttt{<}k''\texttt{,}\ \texttt{()}\texttt{>}}$$

$$\frac{}{S,\ J,\ T,\ \mathcal{W} \vdash \texttt{<}\texttt{end\_until}\ ctrl.k\texttt{,}\ v\texttt{>} \longrightarrow S,\ J,\ T - ctrl,\ \mathcal{W} \vdash \texttt{<}k\texttt{,}\ \texttt{()}\texttt{>}}$$

$$\frac{\begin{array}{c} e/S \Downarrow n/S' \qquad ctrl'', T' = new\_node(T, ctrl, \texttt{When}, n) \qquad k' = k[ctrl' \leftarrow ctrl''] \\ T'' = T' + [\texttt{<}k'\texttt{,}\ \texttt{()}\texttt{>}/ctrl''] \end{array}}{S,\ J,\ T,\ \mathcal{W} \vdash \texttt{<}\texttt{start\_when}_{ctrl}\ e\ (\lambda ctrl'.k)\texttt{,}\ v\texttt{>} \longrightarrow S',\ J,\ T'',\ \mathcal{W} \vdash \texttt{<}\texttt{when}_{ctrl}\ n\ ctrl''\texttt{,}\ \texttt{()}\texttt{>}}$$

$$\frac{n \notin S \qquad \mathcal{W}' = \mathcal{W} + [\texttt{<}\texttt{when}_{ctrl}\ n\ ctrl'\texttt{,}\ v\texttt{>}/n]}{S,\ J,\ T,\ \mathcal{W} \vdash \texttt{<}\texttt{when}_{ctrl}\ n\ ctrl'\texttt{,}\ v\texttt{>} \longrightarrow S,\ J,\ T,\ \mathcal{W}' \vdash \emptyset}$$

$$\frac{n \in S \qquad \mathcal{C} = T^{\mathcal{C}}(ctrl') \qquad T' = T - [T^{\mathcal{C}}(ctrl')/ctrl']}{S,\ J,\ T,\ \mathcal{W} \vdash \texttt{<}\texttt{when}_{ctrl}\ n\ ctrl'\texttt{,}\ v\texttt{>} \longrightarrow S,\ J,\ T',\ \mathcal{W} \vdash \mathcal{C}}$$

$$\frac{}{S,\ J,\ T,\ \mathcal{W} \vdash \texttt{<}\texttt{end\_when}\ ctrl.k\texttt{,}\ v\texttt{>} \longrightarrow S,\ J,\ T - ctrl,\ \mathcal{W} \vdash \texttt{<}k\texttt{,}\ \texttt{()}\texttt{>}}$$

FIG. 7.6 – Réaction de L_k avec structures de contrôle.

$$T, O \vdash \texttt{<present}_{ctrl}\ n\ \texttt{then}\ k_1\ \texttt{else}\ k_2\texttt{,}\ v\texttt{>} \longrightarrow_{eoi} T + [\texttt{<}k_2\texttt{, () >}/ctrl]$$

$$T, O \vdash \texttt{<await immediate}_{ctrl}\ n.k\texttt{,}\ v\texttt{>} \longrightarrow_{eoi} T + [\texttt{<await immediate}_{ctrl}\ n.k\texttt{,}\ v\texttt{>}/ctrl]$$

$$\frac{O(n) = (\mathit{false}, v')}{T, O \vdash \texttt{<await}_{ctrl}\ n(x)\ \texttt{in}\ k\texttt{,}\ v\texttt{>} \longrightarrow_{eoi} T + [\texttt{<await}_{ctrl}\ n(x)\ \texttt{in}\ k\texttt{,}\ v\texttt{>}/ctrl]}$$

$$\frac{O(n) = (\mathit{true}, v') \qquad k' = k[x \leftarrow v']}{T, O \vdash \texttt{<await}_{ctrl}\ n(x)\ \texttt{in}\ k\texttt{,}\ v\texttt{>} \longrightarrow_{eoi} T + [\texttt{<}k'\texttt{, () >}/ctrl]}$$

$$T, O \vdash \texttt{<when}_{ctrl}\ n\ ctrl'\texttt{,}\ v\texttt{>} \longrightarrow_{eoi} T + [\texttt{<when}_{ctrl}\ n\ ctrl'\texttt{,}\ v\texttt{>}/ctrl]$$

FIG. 7.7 – Réaction de fin d'instant de L_k avec structures de contrôle.

$next(O, T, ctrl) =$
  match $T^{Kind}(ctrl)$ with
  | `When` $\rightarrow$
    if $T^n \notin O$ then
      $T, \emptyset$
    else
      Set $T' := T$, $\mathcal{C}' := \emptyset$ in
      forAll $ctrl'$ sous-arbre de $ctrl$ do
        $T', \mathcal{C} := next(O, T', ctrl')$;
        $\mathcal{C}' := \mathcal{C}', \mathcal{C}$;
      done ;
      $T' + [\mathcal{C}'/ctrl]$, $\texttt{<when}_{T^{ctrl}(ctrl)}\ T^n(ctrl)\ ctrl\texttt{, () >}$
  | `Until`$(\lambda x.k) \rightarrow$
    if $T^n \in O$ then
      $T - ctrl$, $\texttt{<}k[x \leftarrow O^v(n)]\texttt{, () >}$
    else
      Set $T' := T$, $\mathcal{C}' := \emptyset$ in
      forAll $ctrl'$ sous-arbre de $ctrl$ do
        $T', \mathcal{C} := next(O, T', ctrl')$;
        $\mathcal{C}' := \mathcal{C}', \mathcal{C}$;
      done ;
      $T' - [T'^{\mathcal{C}}(ctrl)/ctrl]$, $(T'^{\mathcal{C}}(ctrl), \mathcal{C}')$

La fonction commence par déterminer le traitement à appliquer en fonction du type du nœud. Si c'est un nœud `When` et le signal qui contrôle le nœud est absent, cela signifie que ce sous-arbre n'a pas été activé pendant la réaction. Dans ce cas, il n'y a rien à faire. Si le signal est présent, la fonction *next* est appelée sur chacun des sous-arbres du nœud *ctrl*. Puis, toutes les instructions à exécuter par le nœud *ctrl* et ses sous-arbres sont enregistrés dans l'arbre de contrôle. L'expression `when` qui peut débloquer ce nœud est rendue.

Dans le cas d'un nœud `Until`, si le signal est présent, il faut effectuer la préemption. Le sous-arbre *ctrl* est supprimé de l'arbre de contrôle ($T - ctrl$) et l'expression de traitement d'échappement est retournée comme expression à exécuter. Si le signal est absent, la fonction *next* est appelée sur tous les sous-arbres.

Attente passive inter-instant

Dans cette version de la sémantique de L_k, les files d'attente de $\mathcal{W}$ ne sont pas conservées d'un instant à l'autre. Donc cette sémantique n'implante pas l'attente passive inter-instant. Par contre la version de la sémantique sans suspension ni préemption gère l'attente passive inter-instant. Dans l'implantation, pour les expressions définies en dehors de toute construction de contrôle, nous implantons l'attente passive inter-instant mais pas pour les autres.

Une solution pour avoir de l'attente inter-instant avec les instructions de contrôle est de distribuer l'environnement $\mathcal{W}$ dans chaque nœud de T. Ainsi, les files d'attente prennent en compte le signal attendu et le contexte de contrôle dans lequel elles se trouvent. Un des inconvénients majeurs de cette approche vient de l'émission de signaux : elle est plus coûteuse. En effet, pour débloquer les instructions en attente, il faut parcourir la structure de T alors que dans notre version de la sémantique, il y a une seule file d'attente associée à un signal.

D'autres questions se posent avec cette approche. Par exemple, faut-il débloquer les instructions qui sont en attente sous des nœuds `When` suspendus ? Si oui, lorsque le nœud est réactivé dans le même instant alors l'exécution des expressions est instantanée. Mais lorsque le nœud n'est pas activé dans l'instant, cela fait perdre l'information d'attente. Maintenant, si les expressions en attente sous un `When` sont laissées dans les files d'attente, lorsque le corps est réactivé il faut vérifier qu'il n'y a pas de files d'attente à débloquer.

7.4 Conclusion

Dans ce chapitre, nous avons présenté le langage à base de continuation L_k. C'est ce langage qui est utilisé pour l'exécution efficace de ReactiveML. Il permet d'en implanter toutes les constructions.

La difficulté principale de la sémantique de L_k, comme pour Glouton, vient des constructions de contrôle. Elles demandent de concilier deux aspects contradictoires. D'une part, pour être exécutée efficacement, la réaction du programme doit destructurer le terme afin d'avoir un accès direct aux expressions à exécuter. D'autre part, pour pouvoir effectuer les opérations de suspension et préemption, il faut garder la structure du programme. Ainsi, nous sommes obligés d'exprimer du partage entre les ensembles d'expressions à exécuter ou en attente ($\mathcal{C}$ et $\mathcal{W}$) et l'arbre de contrôle (T).

Enfin, nous pouvons remarquer que pour avoir une exécution plus efficace, nous pouvons implanter directement des expressions qui ne sont pas dans le noyau du langage. Par exemple, l'expression `let` x `=` e_1 `in` e_2 peut être traduite directement avec une expression `def` sans avoir besoin de `split` et `join`.

Résumé

Dans ce chapitre, nous avons présenté la sémantique du langage L_k qui est à la base de l'implantation la plus efficace de ReactiveML. L_k est un langage à base de continuations dans lequel nous savons traduire toutes les constructions du noyau de ReactiveML.

Comme pour Glouton, la sémantique de L_k est basée sur l'utilisation de files d'attentes qui permettent l'implantation de l'attente passive intra-instant.

Pour l'implantation de la suspension et de la préemption, nous introduisons la structure d'arbre de contrôle. L'arbre de contrôle permet d'associer à chaque expression son contexte de suspension et de préemption.

Chapitre 8

Implantation

Ce chapitre décrit comment nous passons de la sémantique L_k du chapitre précédent jusqu'à l'implantation en OCAML. Nous commençons par présenter les problèmes dans la compilation de REACTIVEML vers L_k. Puis nous expliquons comment implanter les expressions de L_k par un jeu de combinateurs dans un langage purement fonctionnel. Enfin nous décrivons l'implantation efficace de ces combinateurs en OCAML.

La seconde partie du chapitre est consacrée à la présentation d'une bibliothèque pour la programmation réactive en OCAML et ses deux implantations.

8.1 Compilation vers L_k

Le compilateur `rmlc`[1] que nous avons développé type les programmes, sépare les expressions réactives des expressions instantanées et traduit les parties réactives en L_k. La sortie du compilateur est un ficher OCAML où les instructions de L_k sont implantées par des fonctions OCAML.

Le typeur suit le système de type défini chapitre 5. Pour la séparation des expressions instantanées/réactives, nous nous basons sur les règles de bonne formation du chapitre 3.2. Les expressions instantanées sont laissées inchangées et les expressions réactives sont traduites en L_k. Lors de la séparation, pour les expressions qui peuvent être vues comme instantanées ou réactives nous privilégions la forme instantanée pour qu'elles soient compilées telles quelles vers OCAML.

Pour la traduction vers L_k, nous ne pouvons pas utiliser directement la fonction C_k définie chapitre 7.1. Cette traduction pose en effet des problèmes de croissance de code. Par exemple, traduisons la séquence suivante :

```
present s1 then f1() else f2();
present s2 then g1() else g2()
```

La traduction ne partageant pas les continuations, si nous compilons le programme avec la continuation `k`, nous obtenons le programme L_k suivant :

```
present s1 then
  f1(). present s2 then g1().k else g2().k
else
  f2(). present s2 then g1().k else g2().k
```

[1]Disponible sur `http://www-spi.lip6.fr/~mandel/rml`.

On constate qu'il y a quatre copies de la continuation `k`. Le code produit peut croître exponentiellement par rapport au code source. Pour répondre à ce problème, nous introduisons la construction `bind/in` qui permet d'exprimer le partage des continuations. Ainsi, le programme précédent devient :

```
bind k1 =
  bind k2 = k in
  present s2 then g1().k2 else g2().k2
in
present s1 then f1().k1 else f2().k1
```

La taille du code généré reste ainsi linéaire par rapport au programme source.

Précisons la sémantique de cette construction `bind/in`. Nous étendons le langage L_k défini page 111 :

$$k \quad ::= \quad \dots \mid \texttt{bind } \kappa = k \texttt{ in } k$$

La règle de réaction de cette construction substitue la variable κ par sa définition :

$$S,\ J,\ \mathcal{W} \vdash \texttt{bind } \kappa = k_1 \texttt{ in } k_2 \longrightarrow S,\ J,\ \mathcal{W} \vdash k_2[\kappa \leftarrow k_1]$$

Nous modifions maintenant la fonction de traduction C_k. Comme nous l'avons vu dans l'exemple, il faut modifier la traduction de `present` :

$$C_k[\texttt{present } e \texttt{ then } e_1 \texttt{ else } e_2] = \texttt{bind } \kappa = k \texttt{ in present } C[e] \texttt{ then } C_\kappa[e_1] \texttt{ else } C_\kappa[e_2]$$

De la même façon, la composition parallèle `split` partage dans ses deux branches la continuation `def` x `and` y `in` k :

$$\begin{array}{l} C_k[\texttt{let } x \texttt{ = } e_1 \texttt{ and } y \texttt{ = } e_2 \texttt{ in } e] = \\ \qquad \texttt{bind } \kappa = \texttt{def } x \texttt{ and } y \texttt{ in } C_k[e] \texttt{ in split } (\lambda z.(C_{(\texttt{join } z \ 1.\kappa)}[e_1], C_{(\texttt{join } z \ 2.\kappa)}[e_2])) \\ \text{avec} \quad z \notin fv(e_1) \cup fv(e_2) \end{array}$$

La construction `bind/in` peut également être utilisée pour éviter les problèmes de capture de variable dans la traduction des lieurs. Par exemple, la construction `await/in` peut se traduire par

$$C_k[\texttt{await } e_1(x) \texttt{ in } e_2] = \texttt{bind } \kappa = k \texttt{ in await } C[e_1](x) \texttt{ in } C_\kappa[e_2]$$

Si le domaine de nom de κ n'appartient pas au même domaine de nom que x, il ne peut pas y avoir de capture de variable. De plus, on peut espérer que cette transformation aide le compilateur du langage cible à ne pas étendre inutilement la portée de la variable x à la continuation k.

8.2 Implantation des combinateurs

Pour l'implantation, nous utilisons la technique introduite dans [31]. L'interprète est basé sur un ensemble de combinateurs qui calculent les fonctions de transition à exécuter. Nous les présentons dans un style purement fonctionnel [2]. Nous verrons dans la section suivante comment en donner une traduction efficace en OCaml avec des traits impératifs.

[2] Nous écrivons ici du pseudo-code dans lequel nous nous autorisons des facilités d'écriture comme $s \in S$ pour tester si le signal s est présent dans l'environnement de signaux S.

Nous traduisons les règles des figures 7.2, 7.3 et 7.4 par des fonctions de type *step*. Le type de ces fonctions de transition associe un environnement et une valeur à un nouvel environnement :

$$\begin{array}{lcl} step & = & env \times value \rightarrow env \\ env & = & signal\_env \times join\_env \times waiting \times current \end{array}$$

Un environnement est le produit de *signal_env* le type des environnements de signaux S, *join_env* le type des environnements J, *waiting* celui des fonctions qui associent une file d'attente à un signal et *current* le type des listes de fonctions de transition à exécuter. Nous avons donc :

$$\begin{array}{lcl} signal\_env & = & event\_ref \rightarrow (\alpha, \beta)\, event \\ join\_env & = & join\_ref \rightarrow int \times (value \times value) \\ current & = & (step \times value)\, list \\ waiting & = & event\_ref \rightarrow (step \times value)\, list \end{array}$$

L'implantation des combinateurs reprend les règles du chapitre 7.1. Le combinateur `present` est paramétré par le signal testé s et les deux fonctions de transition $\texttt{f}_1$ et $\texttt{f}_2$.

let `present` s $\texttt{f}_1$ $\texttt{f}_2$ =
  let rec `f` $((S,\ J,\ \mathcal{W},\ \mathcal{C}),\ v)$ =
    if $s \in S$
    then $\texttt{f}_1$ $((S,\ J,\ \mathcal{W},\ \mathcal{C}),\ \texttt{()})$
    else
      if $eoi \in S$ then $(S,\ J,\ \mathcal{W},\ ((\texttt{f}_2,\ \texttt{()}),\ \mathcal{C}))$
      else $(S,\ J,\ \mathcal{W} + [(\texttt{f},\ v)/s],\ \mathcal{C}))$
  in `f`
`present` : *event_ref* $\rightarrow$ *step* $\rightarrow$ *step* $\rightarrow$ *step*

`f` est la fonction de transition de `present`. Si le signal est présent, la branche `then` ($\texttt{f}_1$) est exécutée. Lorsque le signal est absent, nous distinguons deux cas. À la fin de l'instant (*eoi*), la fonction $\texttt{f}_2$ est enregistrée dans $\mathcal{C}$ pour être exécutée à l'instant suivant. Sinon, dans le dernier cas, la fonction de transition de `present` est enregistrée en attente de s dans $\mathcal{W}$.

Il est important de remarquer que l'on enregistre directement les fonctions de transition des expressions à exécuter dans $\mathcal{C}$ et $\mathcal{W}$. Dans les autres implantations du modèle réactif, l'information d'activation ou d'attente remonte dans la structure du terme et lorsque l'on veut réactiver une instruction, il faut reparcourir le terme. Dans certaines implantations, comme Reflex [28], il y a des heuristiques pour essayer de minimiser les parcours, mais la qualité de ces heuristiques dépend du programme à exécuter.

Dans l'implantation, la liaison des noms est faite par le langage hôte. Par exemple, le combinateur `signal` est paramétré par une fonction qui appliquée à un nom retourne une fonction de transition.

let `signal` e_1 e_2 $(\lambda s.\texttt{f})$ $((S,\ J,\ \mathcal{W},\ \mathcal{C}),\ v)$ =
  let $n,\ S' = n \notin Dom(S).\ (n,\ \ S[(e_1, e_2, (false, e_1), \emptyset)/n])$ in
  let `f'` = $(\lambda s.\texttt{f})\ n$ in
  `f'` $((S,\ J,\ \mathcal{W},\ \mathcal{C}),\ \texttt{()})$
`signal` : (*event_ref* $\rightarrow$ *step*) $\rightarrow$ *step*

De cette façon, la substitution est déléguée au langage hôte et également l'allocation et la désallocation de la mémoire. Cette approche est à la fois plus efficace et source de moins d'erreurs.

Implantons maintenant les constructions `split`, `join` et `def` pour étudier la composition parallèle.

let $\texttt{split}$ $(\lambda x.(\texttt{f}_1,\ \texttt{f}_2))$ $((S,\ J,\ \mathcal{W},\ \mathcal{C}),\ v) =$
  let $j,\ J' = j \notin J.\ (j, J[(2,\ (\bot,\ \bot))/n])$ in
  let $\texttt{f}_1\text{'},\ \texttt{f}_2\text{'} = (\lambda x.(\texttt{f}_1,\ \texttt{f}_2))\ j$ in
  $\texttt{f}_1\text{'}\ ((S,\ J',\ \mathcal{W},\ ((\texttt{f}_2\text{'},\ \texttt{()}),\ \mathcal{C})),\ \texttt{()})$
$\texttt{split} : (\mathit{join\_ref} \rightarrow \mathit{step} \times \mathit{step}) \rightarrow \mathit{step}$

Ce combinateur alloue un nouveau point de synchronisation j et exécute la branche gauche dans un environnement où la branche droite doit être exécutée dans l'instant courant.

let $\texttt{join}\ j\ i\ \texttt{f}\ ((S,\ J,\ \mathcal{W},\ \mathcal{C}),\ v) =$
  let $J' = \mathit{join}(J,\ j,\ i,\ v)$ in
  if $J'^{cpt}(j) > 0$ then $(S,\ J',\ \mathcal{W},\ \mathcal{C})$
  else $\texttt{f}((S,\ J',\ \mathcal{W},\ \mathcal{C}),\ J'^{v}(j))$
$\texttt{join} : \mathit{join\_ref} \rightarrow \mathit{int} \rightarrow \mathit{step} \rightarrow \mathit{step}$

Le combinateur `join` décrémente le compteur de branches actives avec la fonction *join* définie dans le chapitre précédent (page 114). La continuation `f` est exécutée uniquement lorsque le compteur arrive à zéro.

let $\texttt{def}\ (\lambda(x{,}y).\texttt{f})\ ((S,\ J,\ \mathcal{W},\ \mathcal{C}),\ v) =$
  let $\texttt{f'} = (\lambda(x{,}y).\texttt{f})\ v$ in
  $\texttt{f'}\ ((S,\ J,\ \mathcal{W},\ \mathcal{C}),\ \texttt{()})$
$\texttt{def} : (\mathit{value} \times \mathit{value} \rightarrow \mathit{step}) \rightarrow \mathit{step}$

Le combinateur `def` utilise la valeur v pour récupérer les valeurs de x et y.

Nous implantons de même les autres constructions de L_k. Il nous reste maintenant à définir l'ordonnanceur qui exécute l'instant tant que l'ensemble $\mathcal{C}$ n'est pas vide, puis prépare le changement d'instant.

La réaction complète d'un programme est définie par la fonction `exec` :

let rec $\texttt{exec}\ (S,\ J,\ \mathcal{W},\ \mathcal{C}) =$
  match $\mathcal{W},\ \mathcal{C}$ with
  | $\mathcal{W},\ ((\texttt{f}, v),\ \mathcal{C}') \rightarrow$
    let $(S',\ J',\ \mathcal{W}',\ \emptyset) = \texttt{schedule f}\ ((S,\ J,\ \mathcal{W},\ \mathcal{C}'),\ v)$ in
    let $\mathcal{L} = \cup_{n \in Dom(\mathcal{W})} \mathcal{W}(n)$ in
    let $(\mathcal{W}'',\ \mathcal{C}'') = \texttt{schedule\_eoi}\ \mathcal{L}\ (S' + [\texttt{()}/\mathit{eoi}])\ J'\ (\mathcal{W}',\ \mathcal{C}')$ in
    let $S'' = \forall n \in S'.S'[n \leftarrow (S'^{d}(n),\ S'^{g}(n),\ \mathit{next}(S')(n),\ \emptyset)]$ in
    $\texttt{exec}(S'',\ J',\ \mathcal{W}'',\ \mathcal{C}'')$
  | $\emptyset, \emptyset \rightarrow \texttt{()}$
$\texttt{exec} : \mathit{env} \rightarrow \mathit{unit}$

La première étape de la réaction d'un instant est l'exécution de la fonction `schedule` jusqu'à ce que l'ensemble des instructions à exécuter à l'instant courant soit vide. Puis toutes les fonctions en attente dans $\mathcal{W}$ sont récupérées et réactivées avec la fonction `schedule_eoi`. Enfin, l'environnement des signaux de l'instant suivant S'' est calculé. Lorsqu'il n'y a plus de fonctions de transition dans $\mathcal{W}$ et $\mathcal{C}$, l'exécution du programme est terminée.

La fonction `schedule` calcule la réaction pendant l'instant en appelant toutes les fonctions de transition présentes dans $\mathcal{C}$:

let rec $\texttt{schedule f}$ $((S,\ J,\ \mathcal{W},\ \mathcal{C}),\ v) =$
  let $(S',\ J',\ \mathcal{W}',\ \mathcal{C}') = \texttt{f}\ ((S,\ J,\ \mathcal{W},\ \mathcal{C}),\ v)$ in
  match $\mathcal{C}'$ with
  | $(\texttt{f}, v),\ \mathcal{C}'' \rightarrow \texttt{schedule f}\ ((S',\ J',\ \mathcal{W}',\ \mathcal{C}''),\ v)$
  | $\emptyset \rightarrow (S',\ J',\ \mathcal{W}',\ \emptyset)$
$\texttt{schedule} : \mathit{step} \rightarrow \mathit{env} \times \mathit{value} \rightarrow \mathit{env}$

`schedule_eoi` réactive les instructions présentes dans $\mathcal{L}$ pour construire le nouvel environnement à exécuter à l'instant suivant :

let $\texttt{schedule\_eoi}\ \mathcal{L}\ S\ J\ (\mathcal{W},\ \mathcal{C}) =$
  match $\mathcal{L}$ with
  | $(\texttt{f}, v),\ \mathcal{L}' \rightarrow$
    let $(S,\ J,\ \mathcal{W}',\ \mathcal{C}') = \texttt{f}\ ((S,\ J,\ \mathcal{W},\ \mathcal{C}),\ v)$ in
    $\texttt{schedule\_eoi}\ \mathcal{L}'\ S\ J\ (\mathcal{W}',\ \mathcal{C}')$
  | $\emptyset \rightarrow (\mathcal{W},\ \mathcal{C})$
$\texttt{schedule\_eoi}$:
  $\mathit{current} \rightarrow \mathit{signal\_env} \rightarrow \mathit{join\_env} \rightarrow (\mathit{waiting} \times \mathit{current}) \rightarrow (\mathit{waiting} \times \mathit{current})$

8.3 Implantation en Ocaml

Une version impérative plus efficace des combinateurs précédents a été réalisée en OCAML. Elle est basée sur la remarque que la fonction de transition de chaque combinateur prend en paramètre et retourne l'environnement. L'environnement peut être directement implanté dans le tas en utilisant des références et en les modifiant physiquement. Les fonctions de transition ont donc le type suivant :

```
type 'a step = 'a -> unit
```

L'environnement est modifié par effet de bord. Le paramètre `'a` est le type de la valeur calculée par l'instruction précédente.

Commençons par l'environnement J de points de synchronisation. En OCAML, nous pouvons écrire les combinateurs `split` et `join` de la façon suivante :

```
let split f v =
  let j = (ref None, ref None) in
  let f1, f2 = f j in
  current := f2 :: !current;
  f1 ()
val split :
  ('a option ref * 'b option ref -> unit step * unit step) -> 'c step

let join (vref1, vref2) vrefi f v =
  vrefi := Some v;
  match !vref1, !vref2 with
  | Some v1, Some v2 -> f (v1, v2)
  | _ -> ()
val join :
  'a option ref * 'b option ref -> 'c option ref -> ('a * 'b) step -> 'c step
```

Le combinateur `split` alloue la mémoire directement dans le tas en créant les deux références initialisées à `None`. Le `join` est paramétré par ces références (`vref1` et `vref2`) et le numéro de la branche à modifier est remplacé par `vrefi`, le pointeur vers une des deux références précédentes. Ainsi il a un accès direct à l'environnement J. Nous avons remplacé le compteur de branches actives par du filtrage sur la valeur calculée par chacune des branches.

Dans l'implantation de `split`, nous observons également que l'ensemble $\mathcal{C}$ est implanté par la liste `current`. C'est une liste modifiable de fonctions de type `unit step`. Nous n'avons pas à enregistrer le couple fonction de transition/valeur dans cette liste, car toutes les fonctions qui s'enregistrent dans cette liste peuvent être vues comme des fonctions de transition de type `unit step`. En effet, seuls les combinateurs `join` et `def` utilisent la valeur donnée en argument de la fonction de transition et donc peuvent avoir des contraintes sur le type de cette dernière. Le combinateur `def` est toujours derrière un `join`. Il est donc exécuté directement sans passer par la liste `current`. Pour le combinateur `join`, soit il est placé derrière une expression ML qui l'exécute instantanément, soit dans tous les autres cas, il attend la valeur `()`.

Étudions maintenant l'implantation des expressions instantanées. Les expressions instantanées sont séparées des expressions réactives avec l'analyse que nous avons présentées au chapitre 3.2. Cela évite d'appliquer la transformation CPS à l'ensemble du code ce qui rendrait les parties OCAML moins efficaces.

Pour l'exécution d'une expression instantanée, le calcul d'une valeur OCAML est effectué et la continuation est exécutée instantanément pour ne pas avoir de problème avec le `join`. Il faut faire attention au moment où l'expression est évaluée. Pour contrôler l'évaluation des expressions ML, nous utilisons le mécanisme de glaçon. Cela consiste à ajouter une abstraction autour de l'expression. Par exemple, l'expression :

```
print_string "hello"; print_newline()
```

traduite avec la continuation `k` devient :

```
compute (fun () -> print_string "hello"; print_newline()) k
```

Ainsi, le message `hello` est affiché lorsque l'on évalue la fonction de transition du combinateur `compute` et pas au moment de sa définition. Le code du combinateur est le suivant :

```
let compute e f v =
  let v' = e() in
  f v'
```

Remarque :

Il est intéressant de définir les parties ML les plus grandes possibles car cela diminue le nombre de glaçons. Le code de l'exemple précédent est équivalent mais meilleur que celui ci :

```
compute (fun () -> print_string "hello")
  (compute (fun () -> print_newline()) k)
```

Étudions maintenant la représentation de l'environnement des signaux. Les idées maîtresses sont de ne jamais manipuler explicitement l'ensemble de tous les signaux et d'avoir un accès direct à toutes les informations tel que le statut d'un signal ou sa valeur associée.

La plupart des implantations de JUNIOR utilisent des tables de hachage pour représenter l'environnement des signaux. Cette représentation ajoute une indirection pour tous les accès.

Dans notre approche, les expressions accèdent directement au signal. L'expression L_k : `signal x default e1 gather e2 in k`, se traduit en OCAML par :

```
fun v -> let x = new_event e1 e2 in k ()
```

Cette fonction de transition alloue la structure de donnée qui implante le signal avec la fonction `new_event`. Puis elle substitue `x` par la référence vers le signal dans la continuation.

Ne pas maintenir l'ensemble des signaux du programme dans une structure de donnée permet au Glaneur de Cellule (GC) du langage hôte de récupérer la mémoire allouée pour les signaux qui ne sont plus utilisables. Si l'environnement est conservé par l'interprète dans une table des signaux, il faut nettoyer régulièrement cette table pour que les signaux inutiles ne soient plus accessibles et donc que le GC puisse récupérer la mémoire. Certaines implantations du modèle réactif comme SIMPLE et REFLEX ont un GC pour la table des signaux. Mais utiliser directement le GC du langage hôte simplifie l'implantation, évite des bogues et surtout évite d'avoir deux GC qui s'exécutent en même temps. De plus, le GC de OCAML [36] se révèle efficace.

L'implantation du type `event` utilisée par défaut dans le compilateur actuel est la suivante :

```
type ('a, 'b) event_struct =
    { mutable status: int;
      mutable pre_status: int;
      mutable value: 'b;
      mutable pre_value: 'b;
      default_value: 'b;
      combine: ('a -> 'b -> 'b); }
and ('a, 'b) event =
   ('a, 'b) event_struct * unit step list ref * unit step list ref
```

Le type `event_struct` représente les informations contenues dans l'environnement de signaux S et le type `event` ajoute à cet enregistrement deux files d'attentes.

Les files d'attente de $\mathcal{W}$ sont directement liées aux signaux. Ceci permet d'avoir un accès direct à la bonne file d'attente quand un test de signal ne peut pas être résolu. Nous avons séparé les files d'attente en deux : $\mathcal{W}_a$ pour l'attente des `await` qui peuvent être gardés d'un instant à l'autre, et $\mathcal{W}_p$ pour les `present` qui doivent être réactivés à la fin de l'instant.

Dans l'enregistrement, le statut d'un signal est un entier qui représente le numéro de l'instant où le signal a été émis. Ainsi, la fonction qui teste la présence d'un signal s'écrit :

```
let is_present s = s.status = !instant
val is_present : ('a, 'b) event -> bool
```

où `instant` est la variable globale qui compte les instants. Cette représentation du statut est utilisée dans la plupart des implantations de JUNIOR. Elle permet de remettre le statut de tous les signaux à la valeur absent seulement en incrémentant le compteur d'instants. En contrepartie, la question de la limite des entiers machine se pose nécessairement. En OCAML, le type `int` définit des entiers modulaires. Il ne peut donc pas y avoir de dépassement de capacité mais un signal qui n'a pas été émis peut être considéré comme présent.

De plus tester le `pre` d'un signal est un peu plus difficile :

```
let pre_is_present s =
  if is_present s
  then s.pre_status = (!instant-1)
  else s.status = !instant-1
val pre_is_present : ('a, 'b) event -> bool
```

Il faut commencer par tester la présence pour savoir si le champ `pre_status` a été mis à jour ou pas. S'il a été mis à jour, il est comparé avec le numéro de l'instant précédent. Sinon nous testons le champ `status`.

Nous avons également réalisé un autre codage des signaux où le statut est un booléen et donc règle le problème des "fausses" émissions de l'implantation précédente. Dans cette version, tous les signaux qui sont émis, sont enregistrés dans une liste `to_update`. Cette liste doit être parcourue à la fin de l'instant pour mettre à jour le statut et les champs `pre` de chaque signal. Afin de réaliser une liste de signaux de types différents, nous avons utilisé le sous-typage des objets de OCAML.

Voyons maintenant l'implantation de l'émission de signal. Pour ne pas avoir à englober toutes les émissions de signaux dans un combinateur `compute`, nous définissons le combinateur `emit`. Il est paramétré par le signal et la valeur à émettre qui sont dans des glaçons et sa continuation. Son code est le suivant :

```
let emit e1 e2 k =
  let (n,wa,wp) = e1() in
  let v = e2() in
  set_emit n v;
  wakeUp wa;
  wakeUp wp;
  k ()
val emit : (unit -> ('a, 'b) event) -> (unit -> 'a) -> unit step -> 'c step
```

Ce combinateur calcule le signal et la valeur à émettre. Il met à jour la structure du signal avec la fonction `set_emit`. Il passe toutes les fonctions de transition de `wa` et `wp` dans `current` avec la fonction `wakeUp`. Puis il active sa continuation.

Dans l'implantation, contrairement à la sémantique, les expressions en attente dans `wa` et `wp` sont réveillées au moment de l'émission et les valeurs émises sont combinées aussi à ce moment là et pas à la fin d'instant. Nous faisons ce changement car au moment de l'émission, toutes les informations nécessaires pour faire ces traitements sont disponibles. Ainsi cela évite de devoir traiter le signal plus tard dans l'instant. Le code de `set_emit` est donc :

```
let set_emit n v =
  if n.status <> !instant
  then
    (n.pre_status <- n.status;
     n.pre_value <- n.value;
     n.status <- !instant;
     n.value <- n.combine v n.default_value)
  else
    n.value <- n.combine v n.value
val set_emit : ('a, 'b) event_struct -> 'a -> unit
```

La fonction `wakeUp` modifie les listes d'attente et la liste `current` par effet de bord.

```
let wakeUp w =
  current := List.rev_append !w !current;
  w := []
val wakeUp : unit step list ref -> unit
```

Le code du combinateur `await_immediate` illustre l'utilisation des files d'attente. Si le signal est présent, la continuation est exécutée instantanément. S'il est absent, la fonction de transition `f` est enregistrée dans la file d'attente persistante (`wa`).

```
let await_immediate s k v =
  let (n, wa, _) = s() in
  let rec f () =
    if is_present n then
      k ()
    else
      wa := f :: !wa
  in f ()
val await_immediate : (unit -> ('a, 'b) event) -> unit step -> 'c step
```

Remarque :

> Le signal `s` est évalué avant la définition de `f` pour qu'il soit calculé seulement lors de la première évaluation de la fonction de transition.

Le dernier combinateur que nous présentons est `present`. La difficulté de `present` par rapport à `await_immediate` est qu'il faut réactiver la fonction de transition à la fin de l'instant si le signal n'a pas été émis. Comme il n'y a pas de table de signaux, à la fin d'instant nous ne pouvons pas retrouver les files d'attente qui sont associées aux signaux. La solution que nous proposons est d'enregistrer les files d'attente qui testent l'absence dans une liste `to_wakeup`. Ainsi, à la fin de l'instant, il y a seulement à parcourir cette liste pour réveiller les instructions en attente.

```
let present s k1 k2 v =
  let (n, _, wp) = s() in
  let rec f () =
    if is_present n
    then k1 ()
    else
      if !eoi
      then current := k2 :: !current
      else (wp := f :: !wp;
            to_wakeup := wp :: !to_wakeup)
  in f ()
val present :
  (unit -> ('a, 'b) event) -> unit step -> unit step -> 'c step
```

La liste `wp` peut être partagée par le signal et la liste `to_wakeup`. Pendant un instant, elle peut être enregistrée dans `to_wakeup` puis, si le signal testé est émis, la liste est transférée dans `current` et le combinateur `present` est réactivé. Dans ce cas, il ne faut pas réactiver le `present` à la fin de l'instant. C'est effectivement ce qui ce passe car la file d'attente enregistrée dans `to_wakeup` est une référence vers une liste de fonctions de transition. Lors de l'émission du signal, la référence est modifiée pour pointer vers la liste vide. Ainsi, à la fin d'instant lorsque `to_wakeup` est parcourue, il n'y a plus les fonctions de transitions qui ont été traitées. Cette méthode évite d'avoir à déposter des informations qui sont dans `to_wakeup`.

Implantation du `loop` avec réinitialisation

Le but de cette section est d'implanter la construction `loop` sans avoir à réallouer les points de synchronisation à chaque itération. Le type de programme qui nous intéresse est le suivant :

```
loop
  let x1 = await immediate s1
  and x2 = await immediate s2 in print_string "hello"
end
```

Nous allons proposer un traitement ad hoc du `loop` et modifier la définition du combinateur `split` que nous avons donnée page 129.

La traduction en OCAML du programme précédent est :

```
loop
  (fun k ->
    let k_def (x1,x2) = compute (print_string "hello") k () in
    split
      (fun j -> (await_immediate (fun () -> s1) (join j 1 k_def),
                 await_immediate (fun () -> s2) (join j 2 k_def))))
```

Le combinateur `loop` est paramétré par une fonction qui associe une fonction de transition à une continuation. Cette fonction représente le corps du `loop`. Une première implantation du combinateur `loop` est de suivre la définition donnée chapitre 3.

```
let rec loop f v =
  let f' = f (loop f) in
  f' ()
val loop : (unit step -> unit step) -> 'a step
```

Ici, `loop` est une fonction récursive qui exécute son corps avec pour continuation la définition du `loop`[3]. On peut constater qu'à chaque itération la fonction de transition `f'` est recalculée alors qu'elle ne dépend pas de `v`.

La solution que nous proposons est de commencer par calculer la fonction de transition du corps du `loop` et donc appliquer `f` à sa continuation. Nous donnons comme continuation la fonction qui exécute la fonction de transition du corps du `loop`. Ainsi la définition du corps ne repasse pas par le combinateur `loop`.

```
let loop f =
  let rec f_1 = lazy (f f_2)
  and f_2 v = Lazy.force f_1 ()
  in f_2
```

La fonction `f_1` est la fonction de transition du corps du `loop`. Elle est obtenue par l'application de `f` à sa continuation `f_2`. Quant à `f_2`, elle évalue `f_1` pour commencer une nouvelle exécution du corps du `loop`. Avec cette nouvelle définition de `loop`, la fonction de transition du corps est calculée une seule fois.

Étudions maintenant le combinateur `split`. Avec la définition que nous avons donnée page 129, un nouveau point de synchronisation est alloué à chaque application de la fonction de transition. Pour pouvoir réutiliser le même point de synchronisation à chaque appel de la fonction de transition, il suffit d'allouer la mémoire avant la définition de cette fonction :

[3]Le type donné ici n'est pas celui inféré par le compilateur. Le typeur de OCAML ne gère pas la récursion polymorphe. Pour obtenir le type souhaité, nous devons utiliser la fonction `Obj.magic`.

```
let split f =
  let (vref1, vref2) as j = (ref None, ref None) in
  let f1, f2 = f j in
  fun v ->
    vref1 := None;
    vref2 := None;
    current := f2 :: !current;
    f1 ()
val split :
  ('a option ref * 'b option ref -> unit step * unit step) -> 'c step
```

L'allocation de la mémoire est faite lors du calcul de la fonction de transition quand `split` est appliquée partiellement à `f`. Quand la fonction de transition est appliquée, la mémoire est seulement réinitialisée.

Remarque :

> Si nous avons une garantie qu'un signal ne peut pas échapper de sa portée, nous pouvons faire une implantation similaire pour le combinateur `signal` en allouant le signal avant la fonction de transition.

Reprenons l'exemple du début de la section :

```
loop
  (fun k ->
    let k_def (x1,x2) = compute (print_string "hello") k () in
    split
      (fun j -> (await_immediate (fun () -> s1) (join j 1 k_def),
                 await_immediate (fun () -> s2) (join j 2 k_def))))
```

Le combinateur `loop` calcule sa fonction de transition au premier instant en appliquant une continuation à la définition de son corps. Le corps de la fonction `(fun k -> ...)` est donc évalué. Lors de cette évaluation le combinateur `split` est appliqué à la définition de ses deux branches. Il alloue alors la mémoire nécessaire à la synchronisation.

Le problème de cette approche est qu'elle augmente la durée de vie de la mémoire allouée. Dans cet exemple, `j` n'est jamais désalloué alors qu'il n'est pas utilisé dans la partie `def`. Cela peut être pénalisant si cette partie est longue et nécessite beaucoup de mémoire.

8.4 Ajout de la suspension et de la préemption

L'ajout des constructions de contrôle suit les indications données chapitre 7.3. Leur implantation repose sur l'utilisation d'un arbre de contrôle.

Les arbres de contrôle ont le type `ctrl_tree` :

```
type contol_tree =
    { kind: contol_type;
      mutable alive: bool;
      mutable cond: (unit -> bool);
      mutable next: unit step list;
      mutable children: contol_tree list; }
```

```
and contol_type =
    Top
  | Until of (unit -> unit step)
  | When
```

Le champ `kind` reprend les sortes de nœuds où `Top` représente la racine de l'arbre. Le champ `alive` est utilisé pour implanter la suppression d'un sous-arbre ($T-ctrl$). Lors de l'exécution des expressions `end_until` et `end_when` ce champ est mis à faux. À la fin de l'instant les branches mortes sont élaguées quand l'arbre de contrôle du prochain instant est calculé avec la fonction `next_ctrl_tree`. Le champ `cond` permet de tester si le signal qui contrôle de nœud est présent. Le champ `next` représente l'ensemble des expressions associées au nœud ($T^{\mathcal{C}}$) et `children` est l'ensemble des sous-arbres ($T^{\mathcal{F}}$).

8.5 Implantation sous forme API générique

Il existe une seconde sortie pour le compilateur que nous appelons L_{co}. Elle génère du code OCAML où les combinateurs sont beaucoup plus proche des constructions du langage source. L'ensemble de leur signature constitue une API (Application Programming Interface) similaire à ce que propose JUNIOR pour JAVA. Par exemple, l'expression REACTIVEML :

```
do
  (await s1 || await s2); emit s3
until s4 done
```

se traduit en OCAML par :

```
rml_until
  (fun () -> s4)
  (rml_seq
     (rml_par
        (rml_await (fun () -> s1))
        (rml_await (fun () -> s2)))
     (rml_emit (fun () -> s3)))
```

Dans le code généré, nous retrouvons le combinateur `rml_until` qui est paramétré par une fonction qui doit s'évaluer en signal et son corps. Le corps est composé d'une séquence (`rml_seq`) qui a un parallèle (`rml_par`) dans sa branche gauche.

L'avantage de cette sortie est de rester très générale. Nous retrouvons la structure du programme source où les constructions REACTIVEML sont directement traduites dans des combinateurs. Ainsi, en fournissant une implantation de ces combinateurs, nous pouvons tester simplement des nouvelles techniques d'ordonnancement sans avoir à modifier le compilateur.

L'interface est définie dans le fichier `lco_interpreter.mli`. Un extrait en est donné figure 8.1. Les expressions REACTIVEML sont de type `'a expr`. Elles doivent être définies à l'intérieur d'un processus qui bloque leur exécution.

Le type des combinateurs reprend les types définis dans le chapitre 5. Nous avons seulement ajouté des glaçons autour des expressions instantanées. Les combinateurs `rml_compute`, `rml_emit_val` ou `rml_run` utilisent des glaçons par exemple. Les lieurs comme les combinateurs `rml_signal_combine` et `rml_until_handler` introduisent des noms en définissant des fonctions. Ils utilisent la même technique que dans la compilation de L_k.

```
type ('a, 'b) event
and 'a expr
and 'a process = unit -> 'a expr

val rml_compute: (unit -> 'a) -> 'a expr
val rml_seq: 'a expr -> 'b expr -> 'b expr
val rml_par: 'a expr -> 'b expr -> unit expr
val rml_pause: unit expr
val rml_run: (unit -> 'a process) -> 'a expr
val rml_emit_val: (unit -> ('a, 'b) event) -> (unit -> 'a) -> unit expr
val rml_await: (unit -> ('a, 'b) event) -> unit expr
val rml_present: (unit -> ('a, 'b) event) -> 'c expr -> 'c expr -> 'c expr
val rml_until_handler:
    (unit -> ('a, 'b) event) -> 'c expr -> ('b -> 'c expr) -> 'c expr
val rml_signal_combine:
    (unit -> 'b) -> (unit -> ('a -> 'b -> 'b)) ->
      (('a, 'b) event -> 'c expr) -> 'c expr

val rml_exec: 'a process -> 'a
```

FIG. 8.1 – Extrait de `lco_interpreter.mli`, l'interface des combinateurs REACTIVEML.

Implantation de Rewrite

Nous avons implanté ces combinateurs en suivant la sémantique de REWRITE [56]. Cette sémantique est basée sur des règles de réécriture simples de la forme suivante :

$$e, S \xrightarrow{\alpha} e', S'$$

où α est un statut de terminaison qui peut être égal à SUSP si l'expression e' doit être réactivée dans le même instant, STOP si e' doit être réactivée à l'instant suivant et TERM (v) si e' a terminé son exécution en rendant la valeur v. Nous donnons quelques règles figure 8.2. La sémantique complète, adaptée à REACTIVEML, avec prise en compte de la partie combinatoire et des signaux valués est donnée en annexe B.

Dans l'implantation de la sémantique REWRITE, le type `'a expr` représente les fonctions de transition des expressions REACTIVEML. Il reprend la structure des règles. La fonction de transition d'une expression calcule le statut de terminaison et la nouvelle expression. Le statut de terminaison est défini par un type somme qui reprend les trois cas de terminaison possibles :

```
type 'a expr = unit -> 'a status * 'a expr
and 'a status = SUSP | STOP | TERM of 'a
```

Les combinateurs traduisent directement les définitions données figure 8.2. Par exemple, lorsque le combinateur `rml_compute` est exécuté, il évalue l'expression `e` et retourne le statut `TERM` et l'expression `v`.

```
let rec rml_compute e =
  fun () ->
    let v = e() in (TERM v, rml_compute (fun () -> v))
val rml_compute: (unit -> 'a) -> 'a expr
```

$$\frac{0 \vdash e \qquad e/S \Downarrow v/S'}{e,\, S \xrightarrow{\text{TERM}\,(v)} v,\, S'} \qquad \frac{e/S \Downarrow n/S' \qquad n \in S' \qquad e_1,\, S' \xrightarrow{\alpha} e_1',\, S''}{\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, S \xrightarrow{\alpha} e_1',\, S''}$$

$$\frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad eoi \notin S'}{\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, S \xrightarrow{\text{SUSP}} \texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, S'}$$

$$\frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad eoi \in S'}{\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, S \xrightarrow{\text{STOP}} e_2,\, S'}$$

$$\frac{e_1,\, S \xrightarrow{\alpha} e_1',\, S' \qquad \alpha \neq \text{TERM}\,(v)}{e_1\,;e_2,\, S \xrightarrow{\alpha} e_1'\,;e_2,\, S'}$$

$$\frac{e_1,\, S \xrightarrow{\text{TERM}\,(v)} e_1',\, S' \qquad e_2,\, S' \xrightarrow{\alpha} e_2',\, S''}{e_1\,;e_2,\, S \xrightarrow{\alpha} e_2',\, S''}$$

FIG. 8.2 – Sémantique REWRITE.

Le combinateur `rml_present` commence par évaluer le signal `evt` à tester, puis il construit la fonction de transition qui teste sa présence et l'exécute. Si le signal est présent, `e1` est exécutée. Si le signal est absent à la fin de l'instant, `e2` doit être exécutée à l'instant suivant. Si le signal est absent pendant l'instant, le statut SUSP est retourné avec la fonction de transition du `present` où `evt` a déjà été évalué.

```
let rml_present evt e1 e2 =
  fun () ->
    let n = evt() in
    let rec self () =
     if is_present n
     then e1 ()
     else
       if !eoi
       then (STOP, e2)
       else (SUSP, self)
  in self ()
val rml_present: (unit -> ('a, 'b) event) -> 'c expr -> 'c expr -> 'c expr
```

L'exécution en séquence de `e1` et `e2` commence par l'exécution de `e1`. Puis, si `e1` est terminée, l'exécution de `e2` peut commencer. Sinon, `e2` est gardée en séquence derrière le résidu de `e1`.

```
let rec rml_seq e1 e2 =
  fun () ->
    match e1 () with
```

```
    | TERM _, _ -> e2 ()
    | (SUSP | STOP) as alpha, e1' -> (alpha, rml_seq e1' e2)
val rml_seq: 'a expr -> 'b expr -> 'b expr
```

Cette implantation de la sémantique REWRITE permet de comparer les sémantiques efficaces par rapport à un ordonnancement naïf. Cela permet également d'avoir un point de comparaison avec JUNIOR qui a une implantation similaire.

Implantation de L_k

Nous avons également implanté une version de L_k dans ces combinateurs. La technique utilisée est d'embarquer la fonction de compilation à l'intérieur des expressions pour qu'elle soit évaluée au moment du `run` et qu'elle génère la fonction de transition. C'est une approche de type *staging* [101].

Ici, tous les combinateurs sont paramétrés par leur continuation et le nœud de contrôle dont ils dépendent. Ils ont la forme suivante :

```
let combinator =
  fun k ctrl ->
    fun v -> ...
```

Ainsi, le type `'a expr` est défini par :

```
type 'a expr = 'a step -> contol_tree -> unit step
and 'a step = 'a -> unit
```

Le type `'a step` est celui des fonctions de transition. Il est défini comme dans la section 8.3. Le type `'a` est celui de la valeur calculée par la fonction de transition précédente. Comme dans le type `'a expr`, la variable de type `'a` représente la valeur calculée par l'expression, alors la continuation de l'expression doit attendre une valeur de ce type.

L'implantation repose les fonctions de transition définies section 8.3. Pour les expressions qui ont une traduction simple de REACTIVEML vers L_k, l'implantation est directe. C'est le cas par exemple du combinateur `rml_emit_val` qui peut appeler la fonction de transition de `emit` :

```
let rml_emit_val e1 e2 =
  fun k ctrl ->
    emit e1 e2 k
val rml_emit_val : (unit -> ('a, 'b) event) -> (unit -> 'a) -> unit expr
```

Pour implanter la construction `let/and/in`, il faut un peu plus de travail. En effet nous avons vu chapitre 7.1 qu'elle se traduit en L_k avec les instructions `split`, `join` et `def`.

```
let rml_def_and e1 e2 e3 =
  fun k ctrl ->
    let def v = e3 v k ctrl () in
    let f ((vref1, vref2) as j) =
      let f1 = e1 (join j vref1 def) ctrl in
      let f2 = e2 (join j vref2 def) ctrl in
      (f1, f2)
    in split f
val rml_def_and : 'a expr -> 'b expr -> ('a * 'b -> 'c expr) -> 'c expr
```

Dans la définition de `def`, la sous-expression `e3 v` est de type `'c expr`. Une fois appliquée à `k` et `ctrl`, elle retourne une fonction de transition de type `unit step`. Cette fonction est exécutée par l'application à la valeur `()`. La fonction `f` définit le corps du `split`.

On peut remarquer que la compilation de `e3` ne peut être effectuée que lorsque la valeur `v` est connue. Donc seulement après la synchronisation des deux `join` et pas lorsque `k` et `ctrl` sont appliqués. De même pour les expressions `e1` et `e2`, elles sont compilées uniquement quand la fonction de transition `split` est exécutée. Il serait intéressant de définir ces combinateurs en MetaOCaml [71] pour spécifier que l'on peut appliquer la phase de compilation avant le premier instant de l'exécution du programme.

8.6 Conclusion

Nous avons présenté ici les différentes implantations de ReactiveML[4]. Elle se regroupent en deux familles : L_k, l'implantation de la sémantique du chapitre précédent et L_{co}, la bibliothèque de combinateurs issus directement de ReactiveML. Les différentes implantations peuvent être testées avec le compilateur `rmlc`. La sélection se fait en utilisant l'option `-runtime` suivie du nom de l'implantation. Chaque nom a le préfixe `Lk_` ou `Lco_` pour indiquer sa famille.

Les implantations L_k sont plus efficaces car les programmes ReactiveML sont traduits dans un format qui est plus simple à exécuter. En revanche, l'avantage du format L_{co} est de proposer une interface simple qui est proche des constructions du langage source ReactiveML. Ainsi, nous pouvons prototyper rapidement des nouveaux ordonnanceurs sans modifier le code du compilateur, mais seulement en fournissant une nouvelle implantation des combinateurs. De plus, l'interface L_{co} peut servir de bibliothèque Ocaml pour la programmation réactive comme le fait Junior pour Java.

Résumé

Nous avons présenté dans la première partie de ce chapitre l'implantation de la sémantique L_k. La construction `bind/in` a été introduite pour avoir un code L_k dont la taille est linéaire par rapport à la taille du programme source. Puis une implantation des expressions de L_k sous forme de combinateurs dans un langage purement fonctionnel a été proposée. À partir de ces combinateurs, une implantation efficace en Ocaml de L_k est présentée.

Dans la seconde partie du chapitre, une bibliothèque pour la programmation réactive en Ocaml est présentée. Cette bibliothèque propose des combinateurs qui sont proche des constructions de ReactiveML. Le compilateur ReactiveML peut générer du code pour cette bibliothèque mais il est relativement simple de les utiliser directement en Ocaml.

De plus cette bibliothèque nous a permis de tester des implantations de ReactiveML simplement sans avoir à modifier le compilateur.

[4]Disponible sur `http://www-spi.lip6.fr/~mandel/rml`.

Quatrième partie

Applications et performances

Chapitre 9

Simulation de protocoles de routage dans les réseaux mobiles ad hoc

Une des principales applications que nous avons réalisées est un simulateur de protocoles de routage dans les réseaux mobiles ad hoc. Ce travail a été réalisé en collaboration avec Farid Benbadis de l'équipe Réseaux et Performances du LIP6 et présenté dans [67]. Les résultats obtenus à partir de ce simulateur ont été présentés dans [8, 7, 6].

Dans ce chapitre, nous commençons par donner les principes des protocoles que nous voulons simuler. Ensuite, nous motivons l'utilisation de ReactiveML pour la réalisation du simulateur. Nous terminons avec la présentation de l'implantation du simulateur et l'analyse de ses performances.

9.1 Le routage géographique basé sur l'âge

Les réseaux mobiles ad hoc sont caractérisés par l'absence d'infrastructure physique. Dans ces réseaux, les nœuds se déplaçant et les connexions entre les nœuds changent dynamiquement. De plus, des nœuds peuvent intégrer ou quitter le réseau à tout moment.

La découverte de la position de la destination est un des problèmes principaux des protocoles de routage classiques. Cette étape génère en général une surcharge importante de trafic. Pour palier à ce problème, les protocoles de Routage Géographique basé sur l'Âge (RGA) ont été introduits récemment [47]. Ils ont pour avantage d'être relativement simples et efficaces. Contrairement aux méthodes classiques où la phase de localisation de la destination et la phase d'acheminement sont séparées, avec les protocoles RGA la découverte de la position de la destination est effectuée pendant l'acheminement des paquets. Il n'y a pas de phase dédiée à la localisation de la destination avant le transfert.

L'hypothèse principale des protocoles de routage basés sur les positions et les âges est que chaque nœud est équipé d'un système de positionnement géographique (GPS ou autre). Ainsi, tous les nœuds peuvent obtenir une information sur la *position* de chaque autre nœud. Cette information est gardée dans une table de positions et est associée à un *âge* qui représente le temps écoulé depuis la dernière mise à jour de la position. La table de positions est utilisée lors du routage d'un paquet pour estimer la position de la destination.

Avec ces méthodes de routage, le nœud source ne connaît pas la localisation exacte de la destination mais seulement une estimation. Cette estimation est mise dans le paquet et est affinée pendant le routage. Nous présentons l'algorithme de routage basé sur l'âge : EASE (Exponential Age SEarch) [47]. Un nœud source n essaye de communiquer avec une destination d :

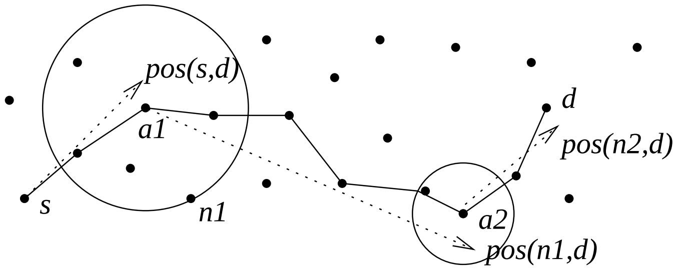


FIG. 9.1 – Routage d'un paquet de s à d : a_1 et a_2 sont des nœuds ancres qui affinent l'estimation de la position de d.

Set $i := 0$, $age := \infty$, $a_0 := s$ in
While $a_i \neq d$ do
    chercher autour de a_i un nœud n_i tel que $age(n_i, d) \leqslant age/2$;
    $age := age(n_i, d)$;
    While le nœud n'est pas le nœud le plus proche de $pos(n_i, d)$ do
        acheminer vers $pos(n_i, d)$
    done ;
    $i := i + 1$;
    $a_i :=$ le nœud le plus proche de $pos(n_i, d)$
done

Les a_i sont les nœuds ancres [1], $pos(n_1, n_2)$ est l'estimation de la position de n_2 donnée par n_1 et $age(n_1, n_2)$ est l'âge de cette information. Une illustration de cet algorithme est présentée figure 9.1.

L'efficacité des protocoles RGA repose sur la distribution des coordonnées des nœuds. Dans notre simulateur, nous comparons deux méthodes de mise à jour des tables de positions. La première, LE (pour Last Encounter), introduite dans [47], utilise les rencontres entre les nœuds. Chaque nœud note dans sa table de positions la localisation et la date de tous les nœuds qu'il rencontre. La seconde méthode, ELIP (Embedded Location Information Protocol) [8], utilise les rencontres comme LE, mais utilise en plus les paquets de données pour disséminer les positions. Avec cette méthode, un nœud source peut inclure ses coordonnées géographiques dans les messages qu'il envoie. Ainsi, tous les nœuds qui participent à l'acheminement des paquets peuvent mettre à jour leur table de positions avec cette information. Nous allons, avec le simulateur, comparer ces deux méthodes de dissémination.

9.2 Discussion du choix de ReactiveML pour implanter le simulateur

La simulation est une étape indispensable avant tout déploiement de nouveaux protocoles de routage. Ceci est particulièrement vrai dans le cadre des réseaux mobiles ad hoc où les expériences à grande échelle sont coûteuses et difficiles à mettre en place. Les simulations nous

[1]Les nœuds ancres cherchent une meilleure estimation de la position de la destination que celle incluse dans le paquet.

permettent d'évaluer les protocoles en mesurant la diffusion des positions, la longueur des routes ou la surcharge du réseau en fonction du choix du protocole.

Les outils de simulation standards comme NS-2 [79] ou OPNET [80] ne sont pas adaptés à la simulation de grands réseaux mobiles ad hoc. NS-2 par exemple a été conçu pour les réseaux filaires et traite donc assez mal les réseaux sans fils. Par exemple, il semble inconcevable de simuler un réseau de 1000 nœuds en NS-2. De plus en NS-2 comme en OPNET, la simulation de la couche 3 (couche réseau) nécessite la simulation de la couche 2 (couche liaison). Ceci induit un sûrcout à l'exécution qui est évitable dans nos simulations. Le dernier point est que ces simulateurs sont difficiles à utiliser et demandent une longue phase d'apprentissage du fait de leur richesse et leur complexité.

Une autre approche envisagée a été l'utilisation de NAB [78]. Ce simulateur est développé par les auteurs de EASE. Il est donc spécialement conçu pour les protocoles de routage géographique basé sur l'âge. NAB est développé en OCAML et est basé sur de la programmation événementielle. Nous verrons dans la section 9.5 que NAB n'est pas adapté pour les simulations avec un grand nombre de communications. La simulation de la couche liaison (même sans perturbations) ralentit les simulations.

Nous avons donc décidé d'implanter un simulateur adapté exactement à notre problème. Une première expérience a été réalisée en C. Pour développer rapidement le simulateur, cette implantation a été faite de façon naïve. Le temps passé pour le calcul du voisinage des nœuds dans ce simulateur rend difficile la simulation de topologies de taille supérieur à 200 nœuds. La seconde version du simulateur faite en REACTIVEML a répondu à nos attentes. Elle permet de simuler des topologies de plus de 1000 nœuds avec un grand nombre de communications.

Les atouts de REACTIVEML pour l'implantation du simulateur sont les suivants. La possibilité de définir facilement des structures de données complexes comme en OCAML aide beaucoup le développement. Le second point est l'adéquation du modèle réactif de Boussinot au problème à traiter. La composition parallèle avec une échelle de temps globale, les communications par diffusion instantanée et la création dynamique aident à écrire simplement des algorithmes efficaces. Il faut noter que le modèle réactif synchrone n'est pas contradictoire avec la nature asynchrone des réseaux. Le synchronisme garantit l'équité dans l'exécution de tous les nœuds comme on le ferait avec un langage impératif.

On peut enfin remarquer qu'il aurait été difficile d'implanter le simulateur dans un langage comme LUSTRE [51], ESTEREL [13] ou SIGNAL [49] pour, au moins, deux raisons : l'utilisation de structures de données complexes partagées entre les parties réactives et combinatoires ainsi que la création dynamique de processus qui n'est pas autorisée dans ces langages.

9.3 Description du simulateur

Le but du simulateur est de comparer deux méthodes de dissémination des coordonnées : LE et ELIP. Nous ne nous intéressons pas aux performances du protocole de routage EASE, mais à la comparaison des méthodes de diffusion des informations de positions.

Couches simulées Un point important à noter est que nous ne simulons pas les couches physique et liaison du modèle OSI (les couches 1 et 2), mais uniquement la couche réseau (couche 3). Nous pouvons faire abstraction des deux premières couches car les protocoles que nous évaluons sont des services utilisés par la couche réseau. Dans notre cas, la perte de paquets ou les interférences n'influencent pas les résultats. Il est seulement important de comparer les deux algorithmes de dissémination dans les mêmes conditions.

Topologie Le simulateur gère des topologies rectangulaires. La topologie est découpée en grille. Les nœuds peuvent se placer sur les intersections de la grille. Plusieurs nœuds peuvent occuper le même point. La distance entre deux points adjacents est de 1 mètre.

Voisinage Le voisinage d'un nœud est l'ensemble des nœuds avec lesquels il peut communiquer directement. Le rayon de couverture r d'un nœud définit la distance maximale à laquelle il peut communiquer. Donc tous les nœuds qui sont à une distance inférieure à r d'un nœud n sont les voisins de n. Dans le simulateur, nous supposons que tous les nœuds ont le même rayon de couverture.

Un paramètre de simulation intéressant est la densité. Elle représente le nombre moyen de voisins d'un nœud. Le rayon se calcule en fonction de la densité avec la formule suivante :

$$r = \sqrt{\frac{\textit{superficie totale de la topologie} * \textit{densité}}{\textit{nombre total de nœuds} * \pi}}$$

Échelle de temps Un instant de simulation correspond au temps nécessaire pour qu'un être humain puisse se déplacer de quelques mètres. En considérant qu'il faut quelques dizaines de millisecondes à un paquet pour aller de la source à la destination, il est raisonnable de considérer que les nœuds sont statiques durant le transfert du paquet de la source jusqu'à la destination. Dans ce cadre, nous faisons l'hypothèse que le routage est instantané.

Algorithme de transfert de paquets Nous avons choisi d'implanter un algorithme de transfert très simple. Un nœud transmet un paquet toujours à celui de ses voisins qui est le plus proche de la destination. Si un paquet atteint une impasse (il faut repasser par le noeud qui vient de transmettre le paquet), nous considérons que le routage a échoué

9.4 Implantation en ReactiveML

Nous décrivons maintenant précisément l'implantation du simulateur. Nous commençons par la description des structures de données utilisées, puis nous verrons l'implantation d'un nœud.

9.4.1 Structures de données

Pour utiliser un protocole de routage géographique basé sur l'âge, un nœud n doit connaître sa position et avoir une table de positions. L'entrée correspondant à un nœud a dans la table de position de n contient les champs suivants : $[ID_a, pos(n,a), date(n,a)]$ où $pos(n,a)$ est une estimation de la position de a et $date(n,a)$ indique l'instant où n a mis à jour cette information pour la dernière fois. Le nœud n possède également la liste de ses voisins.

On utilise donc un type enregistrement pour définir le type des nœuds :

```
type node =
  { id: int;
    mutable pos: position;
    mutable date: int;
    pos_tbl_le: Pos_tbl.t;
    pos_tbl_elip: Pos_tbl.t;
    mutable neighbors: node list; }
```

id est l'identificateur du nœud, pos est sa position courante et neighbors est la liste des nœuds qui sont sous son rayon de couverture. Le champ date est un compteur local qui permet de dater les informations et de calculer leur âge. pos_tbl_le et pos_tbl_elip sont les tables de positions utilisées pour simuler les protocoles de dissémination LE et ELIP.

Les champs de l'enregistrement qui sont annotés mutable peuvent être modifiés, alors que les autres sont fixés lors de la création de l'enregistrement. Les tables de positions pos_tbl_le et pos_tbl_elip ne sont pas mutables, mais elles sont elles-mêmes implantées comme des structures impératives dans le module Pos_tbl. Ces tables de positions associent une position et une date à chaque nœud.

Les paquets pour les protocoles de routage géographique basé sur l'âge doivent contenir les champs suivants : l'identificateur de la source et de la destination, une estimation de la position de la destination, l'âge de cette information et les données à transmettre. Avec le protocole de dissémination ELIP, le paquet peut en plus contenir la position du nœud source.

Dans le simulateur, les paquets n'ont pas besoin de contenir des données mais en revanche, ils contiennent des informations pour calculer des statiques. Ces informations sont aussi utiles pour l'interface graphique.

```
type packet =
  { header: packet_header;
    src_id: int;
    dest_id: int;
    mutable dest_pos: position;
    mutable dest_pos_age: int;
    (* to compute statistics *)
    mutable route: node list;
    mutable anchors: node list; }
```

src_id, dest_id, dest_pos et dest_pos_age sont utilisés pour le routage. route est la liste des nœuds qui ont participé au routage et anchors la liste des nœuds ancres. header indique si c'est un paquet LE ou ELIP.

```
type packet_header =
  | H_LE
  | H_ELIP of position option
```

Le type position option associé au constructeur H_ELIP indique que les paquets ELIP peuvent contenir la position du nœud source ou pas.

9.4.2 Comportement d'un nœud

L'exécution d'une simulation est l'exécution parallèle des nœuds qui composent le réseau. Ainsi, le comportement d'un nœud est à la base du comportement du simulateur.

La réaction d'un nœud est composée de trois étapes. Un nœud (1) se déplace, (2) calcule son voisinage et (3) participe au routage des paquets. Ces trois étapes sont combinées dans un processus node. Ce nœud est paramétré par pos_init (sa position initiale), une fonction move qui lui permet de calculer sa position suivante et une fonction make_msg qui crée une liste de destinations à qui envoyer des paquets. La fonction make_msg représente la couche transport (couche 4).

```
let process node pos_init move make_msg =
  let self = make_node pos_init in
  loop
    self.date <- self.date + 1;

    (* Moving *)
    self.pos <- move self.pos;
    emit draw self;

    (* Neighborhood discovering *)
    ...
    update_pos_tbl self self.neighbors;

    (* Routing *)
    pause;
    let msg = make_msg self in
    ...
    pause;
  end
val node :
  int -> position -> (position -> position) -> (node -> int list) ->
  unit process
```

Ce processus crée un enregistrement de type `node` qui représente son état interne. Puis, il entre dans son comportement permanent qui est exécuté sur trois instants. Au premier instant, le nœud met à jour la date, se déplace et émet sa nouvelle position sur un signal global `draw` pour l'interface graphique[2]. À la fin du premier instant et pendant le second, le voisinage est calculé et les tables de positions sont mises à jour en utilisant les rencontres entre les nœuds. Le troisième instant est consacré au routage des paquets. En plaçant cette étape entre deux `pause`, on garantit que la topologie ne peut pas être modifiée pendant le routage.

Déplacement

Le mouvement des nœuds est paramètré par une fonction `move`. Cette fonction calcule la nouvelle position d'un nœud par rapport à sa position courante. La fonction `move` doit avoir la signature suivante :

```
val move : position -> position
```

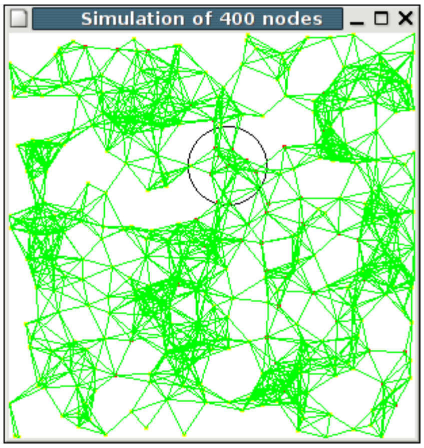
On peut implanter des fonctions de mobilité très simples comme celle qui déplace aléatoirement un nœud dans une des huit positions adjacentes.

```
let random pos = translate pos (Random.int 8)
val random : position -> position
```

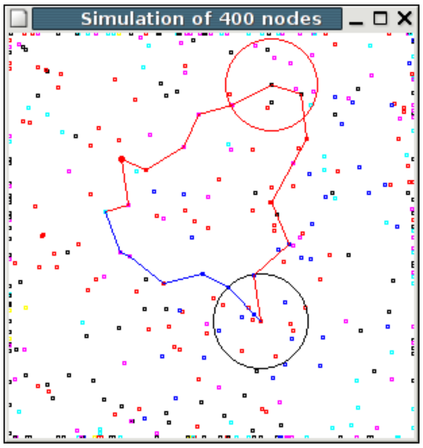
`(Random.int 8)` est l'appel à la fonction `int` du module `Random` de la librairie standard de OCAML. La fonction `translate` retourne une nouvelle position.

On peut implanter également des modèles de mobilité plus réalistes. Celui du Random Waypoint est un des plus utilisés dans les simulations de réseaux mobiles ad hoc. Dans ce

[2]Des captures d'écran sont données figure 9.2.



(a) Voisinage. Chaque segment vert représente une connection entre deux voisins. Le cercle noir représente la zone de couverture d'un nœud.



(b) Un exemple de routes en utilisant les méthodes de dissémination ELIP (bleu) et LE (rouge). Le cercle rouge représente la recherche effectuée par un nœud ancre.

FIG. 9.2 – Captures d'écran de l'interface graphique du simulateur.

modèle, un point est choisi aléatoirement dans l'espace de simulation et le nœud se déplace jusqu'à ce point. Quand le nœud atteint ce point, un nouveau point est choisi. Cette fonction est intéressante, car elle doit garder un état interne.

```
let random_waypoint pos_init =
  let waypoint = ref pos_init in
  fun pos ->
    if pos = !waypoint then waypoint := random_pos();
    (* move in the direction of !waypoint *)
    ...
val random_waypoint : position -> position -> position
```

L'application partielle de cette fonction avec un seul paramètre :

```
  random_waypoint (random_pos())
```

alloue la mémoire pour garder la destination et retourne une fonction de déplacement qui peut être utilisée comme argument par un nœud.

Calcul du voisinage

Dans des réseaux réels, le voisinage est donné directement par les propriétés physiques du réseau. Au contraire, dans un simulateur, il doit être calculé. De plus, un des points clés de l'efficacité du simulateur est cette étape de calcul du voisinage. Nous commençons ici par donner une méthode simple pour ce calcul, puis nous verrons comment l'améliorer.

Pour calculer son voisinage, un nœud doit connaître les positions des autres nœuds. Dans cette première méthode, nous utilisons un signal `hello` pour collecter les coordonnées de tous les nœuds. Chaque nœud émet sa position sur `hello` qui collecte toutes les valeurs émises. Ainsi, le code d'un nœud pour le calcul du voisinage est le suivant (`self` est l'état interne du nœud) :

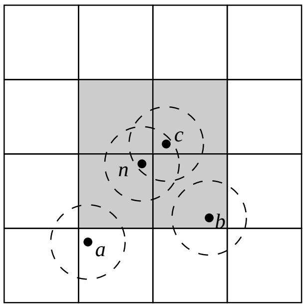


Fig. 9.3 – Topologie découpée en carrés. Le nœud n émet sa position sur les carrés grisés et écoute seulement sur le carré dans lequel il se trouve.

```
emit hello self;
await hello(all) in
self.node_neighbors <- get_neighbors self all;
```

La fonction `get_neighbors` retourne la sous liste de `all` contenant les nœuds à une distance inférieure au rayon de couverture (`coverage_range`) de `self`.

```
let get_neighbors n1 others =
  let filter n2 =
    distance n1.pos n2.pos < coverage_range
  in
  List.filter filter others
val get_neighbors : node -> node list -> node list
```

Cette méthode pour le calcul du voisinage est simple, mais a pour inconvénient d'être coûteuse. En effet, chaque nœud doit calculer sa distance par rapport à tous les autres nœuds participant à la simulation. Pour améliorer cette méthode, l'espace de simulation est découpé en zones. Un signal `hello` est alors associé à chaque zone. Ainsi, un nœud doit calculer sa distance, uniquement, avec les nœuds se trouvant dans les zones qui sont sous son rayon de couverture.

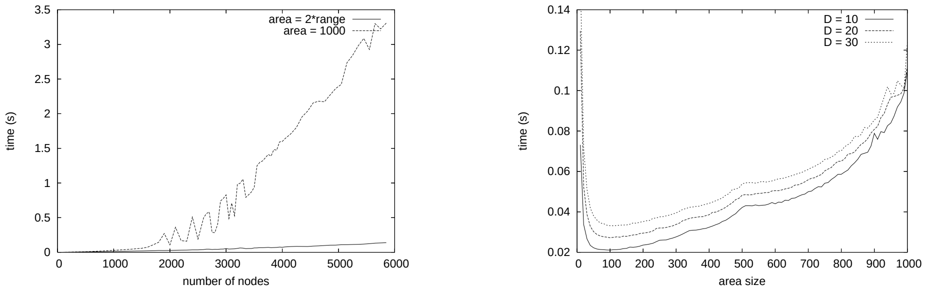
Considérons le nœud n de la figure 9.3. D'une part, n émet sa position sur les signaux associés aux quatre carrés qui sont sous sa portée (les carrés grisés dans la figure). D'autre part, les nœuds a et c qui ont leur rayon de couverture qui touche le carré de n, émettent leurs positions sur le signal associé à ce carré. n écoute le signal du carré sur lequel il se trouve, il reçoit donc les positions de a et c. À partir de ces informations, n calcule sa distance par rapport à a et c. Il en conclue que c est son voisin alors que a ne l'est pas. n ne considère pas le nœud b car b n'émet pas sa position sur le signal associé au carré sur lequel n se trouve.

Tous les signaux `hello` sont stockés dans un tableau à deux dimensions `hello_array`. On définit la fonction `get_areas` qui retourne la zone sur laquelle le nœud se trouve et la liste des zones voisines qui sont sous son rayon de couverture.

```
val get_areas : position -> (int * int) * (int * int) list
```

Maintenant, le comportement d'un nœud est d'émettre sa position dans toutes les zones qui sont sous sa portée et de calculer sa distance par rapport à tous les nœuds qui ont émis leurs positions sur sa zone. Donc, le code d'un nœud devient le suivant :

```
let (i,j) as local_area, neighbor_areas =
```



(a) Comparaison du temps de simulation en fonction du nombre de nœuds et de la méthode de découverte du voisinage.

(b) Comparaison du temps de simulation en fonction de la taille des zones pour la méthode améliorée.

FIG. 9.4 – Temps de simulation pour le calcul du voisinage.

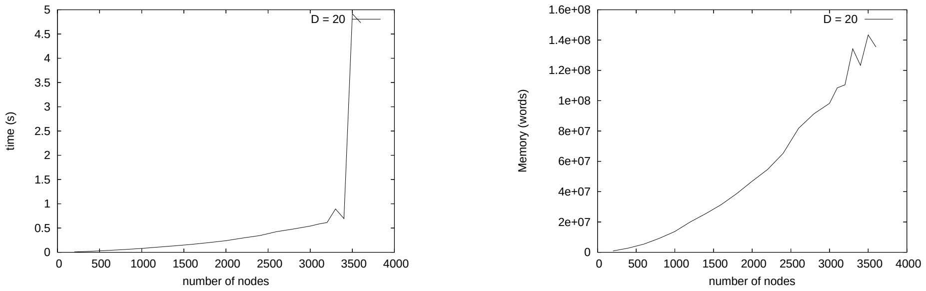
```
    get_areas self.pos.x self.pos.y
  in
  List.iter
    (fun (i,j) -> emit hello_array.(i).(j) self)
    (local_area::neighbor_areas);
  await hello_array.(i).(j) (all) in
  self.neighbors <- get_neighbors self all;
```

La figure 9.4 montre l'effet de cette optimisation sur le temps d'exécution. Dans la figure 9.4(a), nous comparons la première méthode, où tous les nœuds émettent et écoutent sur le même signal, avec la seconde méthode que l'on vient de présenter. Dans la première méthode, chaque nœud doit calculer sa distance par rapport à tous les autres nœuds. Le temps de calcul est ainsi beaucoup plus long que dans la seconde méthode où un nœud calcule sa distance seulement avec les nœuds qui sont dans les zones adjacentes. On observe que pour 1000 nœuds, la seconde méthode est 2 fois plus rapide que la première. Pour 2000 nœuds, elle est 4 fois plus rapide et pour 5000 nœuds elle est plus de 20 fois plus rapide.

Dans la figure 9.4(b), on se concentre sur la seconde méthode. Cette figure représente le temps nécessaire pour simuler 2000 nœuds sur une topologie de taille 1000 en utilisant trois densités différentes[3] et en faisant varier la taille des zones. Comme nous pouvons l'observer, le temps de calcul dépend fortement de la taille des zones. Diviser la topologie en un nombre important de petits carrés n'est pas efficace. Dans ce cas, chaque nœud passe son temps à émettre sa position sur un nombre trop important de signaux. De même, diviser la topologie en de grands carrés est également peu efficace. Ici, chaque nœud reçoit un nombre important de position et doit calculer sa distance avec des nœuds qui sont très éloignés de lui[4]. Les résultats de simulations montrent que des carrés de coté égal à 2 fois le rayon de couverture semblent être un bon compromis.

[3]On rappelle que la densité représente le nombre moyen de nœuds par zone de couverture.

[4]Le cas où la taille de la zone est égale à la taille de la topologie correspond à la première méthode.



(a) Temps de simulation. (b) Occupation mémoire.

FIG. 9.5 – Simulations dépendant du nombre de nœuds avec une densité D=20.

Routage

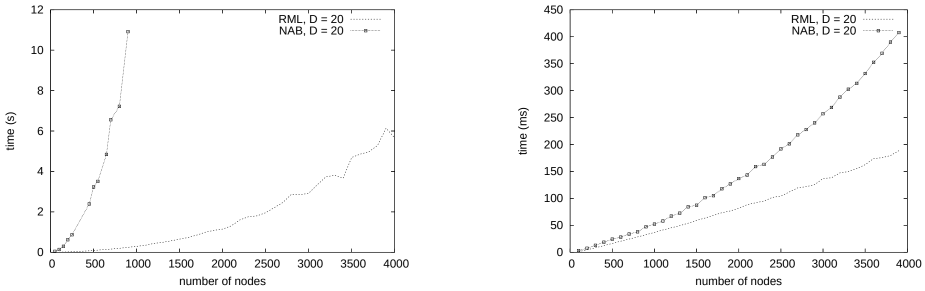
La dernière étape est le routage des paquets. L'algorithme de routage a été décrit dans la section 9.1. Le point important est que nous supposons que le routage est instantané. Ceci signifie que les connexions dans le réseau sont figées pendant le routage. Cette hypothèse est réaliste car nous supposons que les nœuds se déplacent à vitesse humaine alors que les paquets sont transmis à la vitesse des ondes radio. La topologie du réseau change donc à une échelle de temps correspondant à la seconde alors que les paquets mettent quelques dizaines de millisecondes pour aller de la source à la destination. La conséquence de cette hypothèse est que l'on peut implanter la fonction de routage avec une fonction OCAML qui est instantanée.

Dans le simulateur, les deux méthodes de dissémination de coordonnées que nous comparons utilisent le même algorithme de transfert de paquets. Un paquet est toujours transmis au voisin le plus proche (en distance Euclidienne) de la destination. Du point de vue de l'implantation, le point intéressant de la fonction `forward` est qu'un nœud peut accéder à l'état interne des autres nœuds qui s'exécutent en parallèle. REACTIVEML garantit que les actions combinatoires sont atomiques. Les accès aux données partagées n'ont donc pas à être protégés comme dans le modèle des threads préemptifs standards.

9.5 Analyse des performances du simulateur

Le temps de simulation dépend de nombreux paramètres : nombre de nœuds, rayon de couverture, nombre de paquets émis, taille de la topologie, etc. Pour faire des simulations réalistes, nous lions ces paramètres par la densité du réseau qui représente le nombre de nœuds par zone de couverture.

Nous commençons par mesurer les limites de notre simulateur. La figure 9.5(a) représente le temps de simulation en fonction du nombre de nœuds dans le réseau en gardant une densité constante. On observe qu'à partir de 3300 nœuds environ, le temps de calcul devient subitement beaucoup plus important. Ce saut est dû à l'occupation mémoire. Une fois la limite de 3300 nœuds dépassée, il n'y a plus suffisamment d'espace dans la mémoire vive et le programme doit faire des échanges avec le disque. Les échanges disques font écrouler les performances. On peut voir dans la figure 9.5(b) l'occupation mémoire pour les mêmes simulations. La consommation mémoire est au moins quadratique car chaque nœud a une table de positions contenant des informations sur tous les autres nœuds. Pour dépasser cette limitation, nous pouvons utiliser la



(a) avec émission de paquets. (b) sans émission de paquets.

Fig. 9.6 – Comparaison des temps de simulation entre NAB et le simulateur ReactiveML. Les simulations dépendent du nombre de nœuds avec une densité D=20.

même technique que dans le simulateur NAB. Cette technique consiste à limiter le nombre de nœuds destinations pour que seul ces nœuds aient à apparaître dans les tables de positions.

Nous comparons maintenant notre simulateur avec NAB [78], le simulateur développé par les auteurs de EASE. La figure 9.6(a) montre le temps d'exécution pour une simulation où chaque nœud émet un paquet à chaque instant. Ce type de simulation avec beaucoup de mobilité et de communications est intéressant pour évaluer les protocoles de dissémination. Les courbes montrent que l'implantation en NAB est moins efficace que celle réalisée en ReactiveML. Mais cette comparaison n'est pas équitable ! En effet, NAB simule la couche liaison alors que le simulateur ReactiveML ne le fait pas. Même si on choisit une couche liaison parfaite, la simulation de la couche liaison induit un sûrcout pour le routage de tous les paquets.

Afin de comparer équitablement les implantations NAB et ReactiveML, figure 9.6(b), nous les comparons en faisant des simulations sans émission de paquets. Ainsi, nous comparons seulement le temps de découverte du voisinage. Dans ces simulations, la couche liaison n'intervient pas. Les deux implantations font donc exactement la même chose. Ces résultats restent intéressants car l'étape de découverte du voisinage représente environ 25% du temps global de simulation lorsque l'on utilise la version optimisée du simulateur. Les courbes de la figure 9.6(b) nous montrent que l'implantation ReactiveML est environ deux fois plus rapide que NAB sur ces simulations.

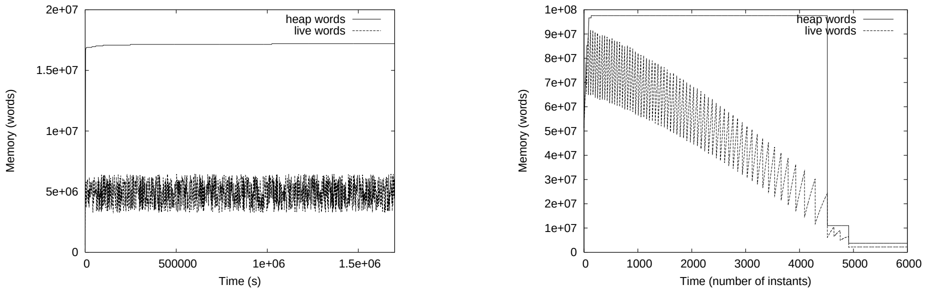
Le dernier point que nous avons testé est la robustesse de l'implantation. Nous avons laissé une simulation s'exécuter pendant 20 jours. Nous pouvons voir dans la figure 9.7(a) que l'occupation mémoire est restée constante pendant cette longue simulation.

9.6 Extensions dynamiques

Dans les réseaux mobiles ad hoc, les protocoles doivent être robustes aux changements de topologies. En particulier, ils doivent prendre en compte l'arrivée et le départ de nœuds. Donc, notre simulateur doit pouvoir gérer l'ajout et la suppression de nœuds en cours de simulation.

Un nœud préemptible est défini par l'exécution d'un nœud dans une construction `do/until` :

```
let process preemptible_node pos_init move make_msg kill =
  do
    run (node pos_init move make_msg)
```



(a) Occupation mémoire pendant 20 jours.

(b) Occupation mémoire pour une simulation où un nœud est supprimé à chaque instant.

Fig. 9.7 – Occupation mémoire en fonction du temps

```
  until kill done
```

Dans la figure 9.7(b), nous observons l'occupation mémoire pour une simulation où un nœud est supprimé à chaque instant. Cela montre que le glaneur de cellules fonctionne aussi sur les processus.

Nous voyons maintenant la création dynamique de nœuds. Elle est réalisée grâce à la composition parallèle et la récursion. Nous définissons le processus `add` qui crée un nouveau processus à chaque fois que le signal `new_node` est émis :

```
let rec process add new_node start =
  await new_node (pos) in
  run (add new_node start)
  ||
  await immediate start;
  run (node pos
          (random_waypoint (random_pos()))
         make_msg)
```

Ce processus est paramétré par deux signaux : `new_node` et `start`. Le signal `new_node` est émis (avec une position initiale) lorsqu'un nouveau nœud doit être créé. `start` est utilisé pour synchroniser les nouveaux nœuds avec les autres. Ce signal est émis à chaque nouvelle étape de déplacement.

9.7 Conclusion

L'implantation complète du simulateur (avec l'interface graphique et la sortie des statistiques) représente environ 1200 lignes de ReactiveML. L'expressivité du langage nous a permis d'écrire assez simplement un simulateur suffisamment efficace pour réaliser les simulations qui nous intéressent. Le simulateur a également été adapté pour tester les performances des différentes améliorations de ELIP ([7, 6]).

Cette expérience de réalisation d'un simulateur de protocoles pour réseaux mobiles ad hoc ne cherche pas à concurrencer des simulateurs comme NS-2 ou OPNET. L'idée est de proposer une alternative aux langages de programmation généralistes afin de faciliter l'implantation de simulateurs dédiés à des problèmes non standards.

Les foncteurs ont manqué lors du développement. En particulier, nous avons réalisé plusieurs implantations des tables de positions et l'utilisation de foncteurs permettrait d'en changer simplement.

Résumé

Nous avons décrit dans ce chapitre la réalisation d'un simulateur de réseaux mobiles ad hoc.

La première partie du chapitre présente le principe des protocoles de routage géographique basé sur l'âge dans les réseaux mobiles ad hoc. Ces protocoles utilisent une estimation de la position de la destination pour le routage des paquets. Cette estimation est affinée pendant le routage. Pour utiliser cette famille d'algorithmes de routages, les nœuds possèdent une table de positions associant à chaque nœud une position géographique. Nous avons présenté deux méthodes de dissémination qui servent à mettre à jour ces tables de positions. Le simulateur a été réalisé pour comparer ces deux méthodes.

Ensuite, nous avons montré que les simulateurs de réseaux classiques comme NS-2 ou OPNET ne sont pas adaptés pour notre problème. C'est pourquoi nous avons décidé d'implanter un nouveau simulateur dédié à la simulation des protocoles de dissémination de coordonnées. Le choix de ReactiveML pour réaliser cette implantation se justifie par l'expressivité du langage algorithmique, la facilité de définir des structures de données complexes et le fait que le modèle réactif de Boussinot est adapté pour la programmation de ce type de systèmes. En effet, nous retrouvons du parallélisme, de nombreuses synchronisations et communications et également de la création dynamique de processus.

Enfin, nous avons décrit le simulateur, son implantation et analysé ses performances. Ce qui ressort de cette expérience est qu'il a été assez simple d'implanter un simulateur suffisamment efficace pour réaliser nos simulations. Nous avons illustré ce point en présentant une optimisation simple permettant de calculer le voisinage d'un nœud.

Cinquième partie

Discussions et conclusion

Chapitre 10

Discussions

Dans cette partie, nous nous intéressons aux choix faits dans le design du langage et aux extensions que nous pourrions apporter. Dans le premier chapitre, nous comparons les constructions de ReactiveML par rapport à celles des autres implantations du modèle réactif et nous présentons d'autres choix sémantiques que nous avons envisagés. Le second chapitre présente les nouvelles extensions du langage incluses dans la distribution et que nous n'avons pas décrites dans les chapitres précédents. Nous présentons également quelques extensions possibles.

Le langage est encore jeune. Les choix que nous présentons ici ne sont pas définitifs. Ils s'appuient sur nos expériences de programmation en ReactiveML, sur la formalisation du langage et sur son implantation. Ces aspects nous ont guidés pour trouver un compromis pour que les constructions du langage soient à la fois utiles, simples et que leur implantation reste efficace.

10.1 Les compositions parallèles

10.1.1 Parallèle imbriqué

Des langages comme Loft et ULM proposent un seul niveau de composition parallèle comme dans le modèle des threads. Au contraire, dans SL, Junior ou ReactiveML, nous avons le droit d'utiliser la composition parallèle à l'intérieur d'un bloc. C'est un des héritages de Esterel.

La seconde forme de parallèle est caractérisée par la synchronisation de la terminaison de ses branches. Ainsi, il est possible d'écrire simplement le programme qui émet le signal `finished` lorsque deux processus `p` et `q` exécutés en parallèle ont terminé leur exécution.

```
(run p || run q); emit finished
```

À l'usage, nous trouvons plus simple de pouvoir composer directement deux expressions plutôt que d'utiliser les primitives `create` de Loft ou `thread` de ULM. Ces primitives implantent à la fois l'exécution d'un processus et sa mise en parallèle. En ReactiveML, le `run` se comporte comme l'application de fonction, il n'exécute pas le processus en parallèle avec l'appelant. Il est toujours possible de remplacer les `run` par le corps des processus à exécuter : le principe de substitution est ainsi préservé.

10.1.2 Parallèle associatif et commutatif

Une des différences entre les calculs basés sur le modèle réactif vient de la commutativité du parallèle. En ReactiveML comme en Junior, la composition parallèle est commutative. Par contre, dans les SugarCubes, l'opérateur `merge` de mise en parallèle garantit que la branche gauche est toujours activée avant la branche droite. Pour Loft, les deux modes d'exécution sont possibles.

Nous avons souhaité conserver l'opérateur de composition parallèle commutatif pour deux raisons : le compositionalité et l'efficacité. Ce n'est pas parce que le parallèle est de gauche à droite que l'ordonnancement est toujours le même. Illustrons cela sur la cascade inverse :

```
let process p s1 s2 s3 =
  await immediate s3; print_int 3
  ||
  await immediate s2; print_int 2;
  emit s3
  ||
  await immediate s1; print_int 1;
  emit s2
  ||
  emit s1
val p :
(unit, 'a) event -> (unit, 'b) event -> (unit, 'c) event -> unit process
```

Nous exécutons ce processus dans trois conditions différentes. Au premier instant le processus `p` est exécuté seul, au deuxième instant il est exécuté en parallèle avec l'émission de `s2` à sa droite et au troisième instant avec l'émission de `s2` à sa gauche :

```
let process main =
  signal s1, s2, s3 in
  print_string "Instant␣1␣:␣"; run (p s1 s2 s3); pause;
  print_string "\nInstant␣2␣:␣"; (run (p s1 s2 s3) || emit s2); pause;
  print_string "\nInstant␣3␣:␣"; (emit s2 || run (p s1 s2 s3))
val main : unit process
```

Avec un ordonnancement de gauche à droite, la sortie affichée par ce processus est la suivante :

```
Instant 1 : 123
Instant 2 : 213
Instant 3 : 231
```

Nous observons trois ordonnancements différents pour l'exécution de `p` alors que l'environnement des signaux est le même à chaque instant. La composition de `p` avec d'autres processus change l'ordonnancement à l'intérieur de `p`. Il est donc dangereux de se baser sur la priorité de la branche gauche pour construire les programmes.

Nous proposons dans le chapitre 11.5 une construction `|>` de composition parallèle de gauche à droite. Cette construction est une version moins expressive de `merge` où la branche gauche ne peut pas dépendre de la branche droite. Ainsi l'ordonnancement ne peut pas changer en fonction du contexte.

Il faut remarquer que l'ordonnancement a de l'impact uniquement sur les programmes à effets de bord. Nous avons vu chapitre 3.4 que la sémantique de ReactiveML sans références est déterministe. Par conséquent, si les opérations faites sur la mémoire dans les différentes branches parallèles sont associatives et commutatives, le programme est déterministe quelque soit l'ordonnancement. Par exemple le programme suivant affiche toujours 2 :

```
let x = ref 0 in
(x := !x + 1 || x := !x + 1); print_int !x
```

Les propriétés importantes du modèle réactif sont d'une part de garantir que l'exécution des expressions instantanées est atomique. Ainsi nous évitons les problèmes des threads où il faut définir des sections critiques pour pouvoir modifier de la mémoire partagée. D'autre part, d'avoir une implantation déterministe afin de pouvoir reproduire une exécution.

Remarquons enfin que la composition parallèle associative et commutative est plus facile à implanter efficacement. L'ordre des parallèles ne doit pas nécessairement être conservé. De plus, la sémantique reste ouverte à l'implantation de nouvelles techniques d'ordonnancement plus efficaces.

10.2 Les préemptions

10.2.1 Le choix de la sémantique du `do/until`

Le modèle réactif propose deux sémantiques pour la préemption. Celle de la construction `kill` de SL [24] et celle du `do/until` de Junior [56].

La préemption de ReactiveML a repris la sémantique du `kill`. C'est une préemption faible : le corps est exécuté à l'instant où le signal de préemption est émis. La particularité de cette construction est d'effectuer le traitement d'échappement à l'instant suivant la préemption.

La construction `do/until` de Junior a une sémantique plus fine. Elle définit également une préemption faible, mais en cas de préemption, le traitement d'échappement peut être effectué instantanément ou reporté à l'instant d'après. Son comportement dépend de l'expression qui est préemptée. Si le corps est arrêté uniquement sur des instructions `pause`, le traitement d'échappement est instantané. Au contraire, si le corps test la présence d'un signal, le traitement d'échappement est retardé d'un instant pour éviter les problèmes de causalité. Par exemple, dans l'expression de gauche, le traitement de l'échappement est instantané et dans celle de droite, il est effectué à l'instant suivant la préemption pour ne pas contredire l'hypothèse `s'` est absent :

```
do
  loop pause end
until s(x) -> emit s' done
```

```
do
  loop await s' end
until s(x) -> emit s' done
```

La sémantique du `do/until` de Junior est plus expressive. Elle permet par exemple de définir le processus `await_immediate_or` qui attend que `s1` ou `s2` soit émis :

```
let process await_immediate_or s1 s2 =
  do do
    loop pause end
  until s1 done until s2 done
val await_immediate_or : ('a, 'b) event -> ('c, 'd) event -> unit process
```

Avec la sémantique de Junior, dès que `s1` ou `s2` est émis, le processus termine. Alors qu'en ReactiveML (avec la sémantique de SL), ce processus a le comportement d'un `await` non immédiat.

Remarque :

> Nous verrons chapitre 11.3 les configurations événementielles en REACTIVEML. Elles permettent d'écrire le programme suivant :
>
> ```
> let process await_immediate_or s1 s2 =
>   await immediate (s1 \/ s2)
> val await_immediate_or : ('a, 'b) event -> ('c, 'd) event -> unit process
> ```

L'inconvénient majeur du `do/until` de JUNIOR vient de la difficulté de sa finesse. Il faut examiner le corps de la construction pour comprendre comment se passe la préemption. De plus, il se marie mal avec l'ordre supérieur. Par exemple, le programme suivant termine après `n` ou `n+1` instant en fonction de `p` :

```
let process timeout n p =
  signal finished in
  do
    run p
    ||
    for i = 1 to n do pause done;
    emit finished
  until finished done
val timeout : int -> 'a process -> unit process
```

Remarquons enfin que la préemption à la SL est plus facile à implanter efficacement que celle à la JUNIOR. Les signaux qui contrôlent les préemptions ont seulement besoin d'être testées à la fin de l'instant lorsque l'environnement des signaux est figé. Il n'est pas nécessaire de tester les `do/until` pendant l'instant.

10.2.2 Les exceptions

OCAML dispose d'un mécanisme d'exception. Nous montrons ici comment elles sont actuellement traitées en REACTIVEML et une autre piste que nous avons envisagée.

Actuellement, les exceptions implantent des opérations de préemption forte : lever une exception interrompt instantanément l'exécution. Pour éviter les problèmes de causalité liés à ce type de préemption, les exceptions levées dans les expressions réactives ne peuvent pas être rattrapées. Ainsi une exception qui n'est pas rattrapée au niveau instantané interrompt nécessairement l'exécution complète du programme.

En JUNIOR et SUGARCUBES, le choix est différent. Les exceptions levées par du code JAVA n'interrompent jamais un programme réactif. Elles sont automatiquement rattrapées au niveaux des expressions instantanées.

Une alternative est de voir les exceptions comme des opérations de préemption faible. Dans ce cas, il est possible de traduire le mécanisme d'exception avec des `do/until` et donc rattraper les exceptions dans des expressions réactives.

Nous proposons une fonction de traduction $T_f[e]$ qui remplace les `try/with` présents dans l'expression e par des `do/until`. L'idée de la traduction est de remplacer les constructions `try/with` par des `do/until` et de remplacer les levées d'exception par des émissions de signaux.

Définissons la fonction T_f. Cette fonction est paramétrée par f une fonction qui filtre les exceptions pour les transformer en des émissions de signaux qui déclenchent les récupérateurs d'exceptions.

Toutes les expressions ML sont encapsulées dans un `try/with` qui récupère toutes les exceptions et applique la fonction f pour déclencher les récupérateurs d'exceptions.

$$T_f[e] = \texttt{try}\ T[e]\ \texttt{with}\ x\ \texttt{->}\ f\,x\,\texttt{;}\,\texttt{halt} \qquad \text{si } 0 \vdash e$$

Lorsqu'une exception est levée par l'exécution de e, un signal est émis par l'application $f\,x$ puis l'exécution est suspendue par l'expression `halt` en attendant la préemption.

Remarque :

> Tout cet ajout de code coûte cher. Il peut être ajouté uniquement si l'expression traduite peut lever une exception [84].

Étudions maintenant la traduction des récupérateurs d'exceptions.

$$
\begin{array}{l}
T_f[\texttt{try}\ e\ \texttt{with}\ E\ \texttt{->}\ e'] = \\
\quad \texttt{signal}\ exn_E\ \texttt{in} \\
\quad \texttt{do} \\
\quad\quad \texttt{let}\ f'\ exn\ \texttt{=} \\
\quad\quad\quad \texttt{match}\ exn\ \texttt{with} \\
\quad\quad\quad \texttt{|}\ E\ \texttt{->}\ \texttt{emit}\ exn_E \\
\quad\quad\quad \texttt{|}\ \_\ \texttt{->}\ f\ exn \\
\quad\quad \texttt{in}\ T_{f'}[e] \\
\quad \texttt{until}\ exn_E\ \texttt{->}\ T_f[e']\ \texttt{done}
\end{array}
$$

Un nouveau signal exn_E est introduit. Il est émit par la traduction de l'expression e si l'exception E est levée. Il est donc utilisé comme condition d'échappement. Pour émettre le signal exn_E quand l'exception E est levée, le corps du `try/with` est traduit avec la fonction de filtrage f'. Cette fonction f' filtre l'exception qui lui est donnée en entrée. Si c'est l'exception E, alors exn_E est émis. Sinon, la fonction f est appelée. Remarquons que si la fonction f filtre l'exception E, la définition de f' masque ce traitement. Ainsi, la sémantique des `try/with` imbriqués qui rattrapent les exceptions avec le `try/with` le plus interne est bien respectée.

Regardons enfin la traduction de `process` e et `run` e. Pour toutes les autres expressions du langage, la fonction T_f traduit seulement les sous-expressions.

$$
\begin{array}{l}
T[\texttt{process}\ e] = \texttt{process}\ (\lambda f.T_f[e]) \\
T_f[\texttt{run}\ e] = \texttt{run}\ ((T_f[e])\ f)
\end{array}
$$

Les définitions de processus sont paramétrées par une fonction de filtrage des exceptions. L'application de processus donne la fonction de filtrage correspondant au contexte où le processus est exécuté.

Si nous appliquons cette transformation, le processus principal doit être appelé par le processus `main` suivant. Il interrompt l'exécution du programme si une exception n'est pas rattrapée :

```
exception Exn of exn list
let process main p =
  signal exn in
  let f x = emit exn x in
  do
    run (p f)
  until exn(x) -> raise (Exn x) done
val main : ((exn -> unit) -> 'a process) -> 'a process
```

Il collecte toutes les exceptions qui n'ont pas été rattrapées et préempte le processus principal en levant l'exception Exn.

Avec cette sémantique de préemption faible pour le try/with, plusieurs exceptions peuvent être levées en parallèle. Par exemple :

```
try
  try
    raise E1 || raise E2
  with E1 -> print_string "E1"
with E2 -> print_string "E2"
```

Dans ce cas, l'exception qui remonte le plus haut préempte le reste du programme. Si la même exception est levée deux fois, les deux exceptions sont rattrapées mais le traitement de l'exception est effectué une seule fois.

```
try
  raise E || raise E
with E -> print_string "E"
```

Dans la traduction, nous n'avons pas traité le cas des exceptions valuées. Ces exceptions posent un problème si une même exception est levée plusieurs fois au cours d'un instant. C'est un cas similaire au problème de multi-émission. Nous pouvons envisager au moins deux solutions. La première est de fournir une fonction de combinaison à la déclaration des exceptions. Cette solution a pour inconvénient de ne pas être compatible avec OCAML. La seconde solution est d'appliquer en parallèle le traitement d'échappement pour chaque exception qui est levée.

Le défaut majeur de la traduction est que la construction try/with a deux comportements différents en fonction de son corps. Nous retombons sur le même problème que le do/until de JUNIOR. Si le corps du try/with est une expression ML, le traitement de l'échappement est fait instantanément pour ne pas changer la sémantique de OCAML. En revanche, si le corps est une expression réactive, le traitement de l'échappement est décalé d'un instant. Une solution à ce problème est de proposer une nouvelle construction : try/catch par exemple, qui permet de récupérer les exceptions dans les expressions réactives.

10.3 Les signaux

La statut des signaux a beaucoup évolué au cours des différentes versions de REACTIVEML. Nous présentons ici leurs caractéristiques actuelles et les différents choix que nous avons envisagés.

10.3.1 Liaison des noms de signaux

La première caractéristique que nous avons choisie et sur laquelle nous ne sommes pas revenus est la liaison statique des noms de signaux. Illustrons cela sur l'exemple suivant :

```
signal to_exec in
begin signal s in
  await s; print_string "Static"; print_newline()
  ||
  begin signal s in
    await s; print_string "Dynamic"; print_newline()
```

```
    ||
    await to_exec(fun_list) in List.iter (fun f -> f()) fun_list
  end
  ||
  emit to_exec (fun () -> emit s)
end
```

Dans l'exemple, deux signaux `s` sont déclarés l'un dans l'autre. Si le signal le plus extérieur est émis, le message `Static` est affiché. Si c'est le signal `s` intérieur qui est émis, le message `Dynamic` est alors affiché. Sous la portée du signal `s` extérieur, la fonction qui émet `s` est émise sur le signal `to_exec`. Cette fonction est récupérée et exécutée sous la portée du second `s`. Comme la liaison des noms est statique, l'exécution de cette fonction émet le `s` extérieur même si elle est exécutée sous la portée du `s` intérieur. Ainsi, le message `Static` est affiché.

Le choix de la liaison statique des noms de signaux est cohérent avec la liaison des noms de variables en OCAML. De plus, cela permet d'avoir une exécution plus efficace des programmes. Avec de la liaison dynamique des noms, à chaque accès à un signal, il faut aller rechercher dans le contexte la définition à laquelle le signal se rapporte, alors que dans notre cas, l'accès est direct (en temps constant).

Nous avons également déjà vu que la liaison statique permettait d'utiliser le glaneur de cellules de OCAML pour gérer l'allocation des signaux. Avec de la liaison dynamique, nous aurions été obligés de manipuler explicitement l'ensemble des signaux et donc de mettre en place un mécanisme d'élimination des signaux inutiles.

En revanche, un des avantages de la liaison dynamique est qu'il ne peut pas y avoir de phénomène d'échappement de portée. En effet, si un signal sort de sa portée, il entre alors sous la portée d'un autre signal qui le redéfinit. Cela permet donc d'utiliser toutes les optimisations que nous avons vues pour les signaux qui n'échappent pas de leur portée.

10.3.2 Les signaux comme des valeurs de première classe

En REACTIVEML, les signaux sont des valeurs de première classe. Cela veux dire que les signaux peuvent être manipulés directement. Par exemple, des listes de signaux ou une référence vers un signal peuvent être créées. De plus, comme dans le π-calcul, les signaux peuvent transmettre des signaux. Dans l'exemple suivant, le signal `x` reçoit le signal `y` sur lequel il émet la valeur `42` :

```
await x(y) in emit y 42
```

Dans les premières versions de REACTIVEML, les signaux n'était pas des valeurs de première classe. Cette approche se mariait mal avec la nature du langage et n'évitait pas les problèmes d'échappement de portée. Par exemple, à la place d'émettre le signal, les fonctions d'émission et d'attente du signal peuvent être émises :

```
signal x in
signal y in
emit x ((fun v -> emit y v), process (await y))
```

Remarque :

> En REACTIVEML, nous avons gardé une syntaxe spéciale pour introduire les noms de signaux. En revanche, en ULM, les signaux sont nommés avec la construction `let` et

> l'expression `sig` crée un nouveau signal sans le nommer. Cette expression `sig` peut se programmer en REACTIVEML avec la fonction `new_sig` :
>
> ```
> let new_sig () = signal s in s
> val new_sig : unit -> ('a, 'a list) event
> ```

10.3.3 La multi-émission

Une des spécificités de REACTIVEML est la gestion de la multi-émission des signaux valués. Dans les autres implantations du modèle réactif, il y a un mécanisme qui permet de récupérer toutes les valeurs émises pendant un instant. En ESTEREL, une fonction de combinaison peut être définie, mais le type des valeurs émises sur le signal doit être le même que celui des valeurs qui sont reçues. En REACTIVEML, avec la construction `signal/default/gather/in`, ces deux choix peuvent être implantés et d'autres façons de combiner les signaux peuvent également être proposées.

La récupération de tous les valeurs d'un signal est aussi expressive que la solution proposée. En effet, une fois les valeurs reçues, elles peuvent ensuite être combinées. Mais à l'usage, la définition de la fonction de combinaison à la déclaration du signal s'est révélée très pratique.

Avec l'utilisation de la fonction de combinaison, l'absence de multi-émission peut être vérifiée à l'exécution. Dans l'exemple suivant, la fonction `single` crée un signal qui s'il est émis plus d'une fois par instant lève l'exception `Multi_emission` :

```
exception Multi_emission

let single () =
  signal s default None
  gather
    fun x y ->
      match y with
      | None -> Some x
      | Some _ -> raise Multi_emission
  in s
val single : unit -> ('a, 'a option) event
```

La puissance de l'analyse de causalité de ESTEREL permet de rejeter les programmes avec multi-émission. Faire cette analyse de causalité pour un langage comme OCAML nous semble difficile.

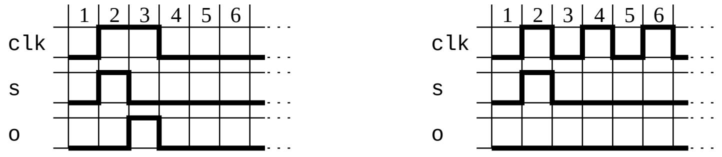
10.3.4 Sémantique du `pre`

L'expression `pre s` est une expression booléenne qui s'évalue en vrai si le signal était présent à l'instant précédent et en faux sinon. Cette expression se compose mal avec la suspension. Illustrons cela avec le processus `danger` :

```
let process danger o =
  signal s in
  emit s;
  pause;
  if pre s then emit o
val danger : (unit, 'a) event -> unit process
```

Étudions la réaction de ce processus exécuté dans un `do/when` contrôlé par le signal `clk`. Nous présentons deux exécutions de ce programme en fonction des valeurs de `clk` :

```
do run (danger o) when clk
```



Dans le chronogramme de gauche, le signal `clk` est présent pendant deux instants successifs. Dans ce cas, le signal `o` est émis (`pre s` s'est évalué en `true`). Dans le chronogramme de droite, `clk` est présent un instant sur deux. Cette fois ci, le signal `o` n'est pas émis car à l'instant 4, lorsque l'expression `pre s` est évaluée, le statut de `s` à l'instant précédent est absent. Nous constatons que même si le signal `s` est défini localement au processus `danger`, la valeur de `pre s` dépend de son contexte d'exécution.

En Esterel le comportement du processus `danger` est différent, quelque soit le contexte dans lequel il est exécuté, il émet toujours le signal `o`. Cette différence vient de l'*horloge* des signaux. En Esterel, un signal a l'horloge de son contexte de définition. Cela signifie que le statut précédent d'un signal est le dernier statut qu'avait le signal lorsque son contexte de définition était actif. Un signal est défini uniquement si son contexte de définition est actif. En ReactiveML, tous les signaux sont définis à tous les instants. On dit qu'il sont sur l'horloge de base.

Cette différence entre Esterel et ReactiveML vient du phénomène d'échappement de portée qui existe en ReactiveML mais pas en Esterel. Avec l'échappement de portée un signal peut sortir de son contexte de définition et donc être émis sur une horloge plus rapide que celle sur laquelle le signal est défini. C'est le cas par exemple du programme suivant :

```
let process scope_extrusion =
  signal x, clk in
  emit clk
  ||
  do
    signal s in
    emit x s;
    await s
  when clk
  ||
  await one x (y) in loop emit y; pause end
```

Étudions maintenant le cas des signaux valués :

```
let process no_danger o =
  signal s default 0 gather (+) in
  emit s 42;
  pause;
  emit o (pre ?s)
val no_danger : (int, 'a) event -> unit process
```

L'expression `pre ?s` correspond à la dernière valeur associée au signal vers `s`. Donc ici, quelque soit le contexte d'exécution du processus `no_danger`, le signal `o` est émis avec la valeur `42`. Même si plusieurs instants sont exécutés entre l'émission de `s` et le calcul de `pre ?s`, la valeur de `pre ?s` est la dernière valeur du signal lorsqu'il a été émis.

Dans la première version de `pre ?s` que nous avions implantée, si le signal était absent à l'instant précédent l'évaluation de `pre ?s` alors la valeur rendue était la valeur par défaut du signal. Mais avec cette sémantique de `pre ?s`, nous retombons dans les mêmes problèmes de composition avec la suspension que `pre s`.

10.3.5 Les `await`

Nous terminons la discussion sur les signaux en présentant les choix sur la sémantique de `await`. En SL, Junior, Loft ou ULM, l'instruction `await` correspond à notre `await/immediate`. En ReactiveML la distinction entre `await` et `await/immediate` a été conservée pour les mêmes raisons que Esterel. En particulier attendre deux occurrences de `s` s'écrit :

```
await s; await s
```

Mais contrairement à Esterel, l'expression `await s` est équivalente à :

```
await immediate s; pause
```

alors que pour Esterel, l'instruction `pause` est avant le test de présence :

```
pause; await immediate s
```

Nous avons fait ce choix pour être cohérent avec l'attente de signaux valués. Ainsi, `await s` est équivalent à :

```
await s(_) in ()
```

10.4 Séparation instantané/réactif

10.4.1 Le statut de `signal/in` et `emit`

Le critère choisi pour la séparation entre expressions instantanées et réactives est basé sur le nombre d'instants nécessaires pour exécuter les expressions. Une expression qui est toujours exécutée en un instant est instantanée, sinon c'est une expression réactive. En particulier, les expressions `signal/in` et `emit` sont des expressions instantanées.

Nous n'avons pas toujours fait ce choix. Dans [68], les expressions instantanées sont les expressions purement ML. Cette différence a des répercutions sur la présentation de la sémantique, sur l'expressivité du langage et sur l'implantation.

D'un point de vue sémantique, interdire les constructions `emit` et `signal` dans les expressions instantanées permet de présenter ReactiveML comme un langage à deux niveaux. Le langage hôte garde sa sémantique inchangée, et nous proposons par dessus de nouvelles constructions réactives qui peuvent utiliser des constructions du langage hôte. Par exemple dans la sémantique comportementale, la règle pour `emit`, s'écrit alors :

$$\frac{e_1 \Downarrow n \qquad e_2 \Downarrow v}{\emptyset \vdash \texttt{emit}\ e_1\ e_2 \xrightarrow[S]{[\{v\}/n],\, true} \texttt{()}}$$

où la réduction $e \Downarrow v$ est la réaction d'une expression ML. Un des avantages de cette approche est de conserver toutes les propriétés du langage hôte. Mais en contrepartie, les constructions qui sont présentes dans les deux niveaux comme `let/in` sont dupliquées.

Dans notre approche, il n'y a plus de séparation entre le langage hôte et la partie réactive. Ceci oblige à refaire les preuves des propriétés des expressions purement ML. En contrepartie, il n'y a qu'un seul ensemble de règles, ce qui donne une présentation unifiée de la sémantique.

D'un point de vue conception de langage, nous avons trouvé à l'usage plus naturel de considérer la déclaration et l'émission de signal comme des expressions instantanées. Par exemple, il est pratique de pouvoir utiliser les fonctions de la librairie standard de OCAML pour manipuler les listes de signaux. Ainsi, émettre tous les signaux d'une liste s'écrit :

```
let emit_list l = List.iter (fun x -> emit x) l
val emit_list : (unit, 'a) event list -> unit
```

Remarquons qu'avec la séparation actuelle des expressions instantanées, l'application de fonction peut émettre des signaux alors que si l'émission est limitée aux processus, nous savons que les expressions instantanées n'émettent pas de signaux. Nous verrons chapitre 11.5 que cela empêche la définition de l'opérateur de composition `<|` qui doit garantir que la partie droite n'émet pas de signaux.

Enfin, il faut noter que la séparation a aussi des conséquences sur l'efficacité de l'implantation. À cause de la déclaration de signaux dans les parties instantanées, l'implantation de la bibliothèque REACTIVEML n'est pas *thread-safe*. Cela signifie que certains modules ont un état et donc deux machines réactives ne peuvent pas être exécutées en parallèle avec la bibliothèque de processus légers de OCAML.

En effet, si plusieurs machines réactives sont exécutées simultanément, lors de la création d'un signal il faut déterminer la machine courante. Les parties de code instantané ne pouvant pas être modifiées, cette information devrait être récupérée par un mécanisme d'introspection coûteux.

10.4.2 Processus avec une valeur de retour

Dans les versions précédentes de REACTIVEML, il était interdit d'utiliser une expression réactive dans la partie gauche d'un `let/in`. Ceci correspond à la règle de bonne formation :

$$\frac{0 \vdash e_1 \quad k \vdash e_2}{k \vdash \texttt{let}\ x \texttt{ = } e_1\ \texttt{in}\ e_2}$$

Ainsi, il était impossible de récupérer la valeur d'un processus. Donc les processus n'avaient pas le type τ `process` mais simplement `process`. Pour transmettre une valeur calculée par un processus, nous utilisions les signaux. Cette approche est celle de JUNIOR.

Pour des exemples comme les automates cellulaires ou le simulateur de réseaux, ce choix n'est pas pénalisant car en général, chaque processus a un comportement qu'il doit exécuter pour toujours et donc ne calcule pas de valeurs. En revanche, pour définir une fonction comme la fonction factorielle, cela alourdit beaucoup l'écriture du programme :

```
let rec process fact n res =
  pause;
  if n <= 1 then emit res 1
```

```
  else
    signal res' in
    run (fact (n-1) res')
    ||
    await immediate one res'(x) in
    emit res (x * n)
val fact : int -> (int, int list) event -> unit process
```

En utilisant la récupération des valeurs de retour des processus, le processus `fact` se définit par :

```
let rec process fact n =
  pause;
  if n <= 1 then 1
  else
    let x = run (fact (n-1)) in x * n
val fact : int -> int process
```

Contrairement à ULM, nous avons gardé une distinction entre les processus et les fonctions. La première raison est l'efficacité de l'exécution. Les fonctions combinatoires et les processus sont compilées différemment. En présence d'ordre supérieur, si nous ne pouvons pas distinguer l'application de processus et l'application de fonction, cela conduit à considérer (et donc compiler) toutes les fonctions comme des processus et donc produire du code moins efficace.

De plus, si les expressions réactives ne peuvent pas être identifiées, des problèmes d'ordre d'évaluation se posent. Dans l'exemple suivant, si `g` et `h` sont des expressions réactives, l'ordre d'évaluation de `(g 1)` et `(h 2)` peut changer la sémantique du programme.

```
let f g h = (g 1) + (h 2)
```

Comme en OCaml, l'ordre d'évaluation des paramètres d'une fonction n'est pas spécifié, cela changerait la nature du langage si cet ordre est fixé en ReactiveML. Donc nous ne pouvons pas utiliser le solution de ULM pour résoudre ce problème.

10.4.3 Exécution du programme et code réentrant

Dans des langages comme Junior ou ReactiveC, la fonction d'exécution des instants `react` est disponible dans le langage. C'est au programmeur de définir la boucle d'exécution des instants. Cette approche laisse une grande liberté dans la définition du mode d'exécution du programme. Par exemple, le programme peut être échantillonné ou réagir sur l'arrivée d'un événement externe. En revanche, cette liberté peut conduire à des problèmes de réentrance. La fonction `react` étant disponible dans le langage, le programme réactif peut pendant l'exécution d'un instant appeler récursivement la fonction `react`. Afin d'éviter ce problème, un langage comme SugarCubes teste à l'exécution que l'appel à la fonction `react` n'est pas réentrant.

En ReactiveML, pour éviter ce problème, nous avons choisi une solution différente. La fonction `react` n'est pas disponible dans le langage. C'est au moment de la compilation que le processus principal est défini. Le compilateur génère alors le code faisant progresser les instants.

Résumé

Dans ce chapitre, nous avons discuté du choix des constructions du langage et de leur variantes. Nous avons commencé par justifier le choix d'un opérateur de composition parallèle commutatif et qui peut être utilisé à l'intérieur d'un bloc. Puis nous avons regardé les préemptions et proposé une nouvelle solution pour le traitement des exceptions. Ce chapitre se termine avec une discussion sur les signaux et la séparation des expressions instantanées et réactives.

Chapitre 11

Extensions

Dans ce chapitre, nous présentons des extensions du langage. Certaines telles les configurations événementielles existent déjà. D'autres, comme les automates, sont encore à implanter.

11.1 Les scripts réactifs

Nous proposons un mode interactif pour ReactiveML [1]. Ce mode correspond à la boucle d'interaction (ou *toplevel*) de Ocaml. Dans ce mode, les programmes ReactiveML peuvent être définis et exécutés de façon interactive. Le toplevel (`rmltop`) prend des phrases ReactiveML en entrée et les interprète. Ce mode d'exécution des programmes permet l'observation et la mise au point de systèmes réactifs.

Dans le monde du réactif, ce mode d'interprétation de programmes reprend l'idée des Reactive Scripts [25] de Frédéric Boussinot et Laurent Hazard. La version originale de ce langage de script a été réalisée au-dessus de ReactiveC et de Tcl-Tk. Par la suite, Jean-Fredy Sunini en a proposé une version au-dessus de SugarCubes et Java [100].

Les Reactive Scripts ont été introduits pour prototyper de manière interactive des comportements réactifs sans avoir à arrêter l'exécution du programme. Ainsi, il est possible de modifier dynamiquement le comportement d'un programme.

Un exemple de session `rmltop` est donné figure 11.1. Cette session commence par la ligne :

```
signal s;;
```

qui déclare un signal global `s`. Cette déclaration est suivie de trois informations données par l'interprète : (1) le type inféré par le compilateur ReactiveML, (2) le type de la valeur Ocaml générée et (3) sa représentation. Nous pourrions masquer l'implantation concrète des signaux et fournir des fonctions d'inspection mais lors de la mise au point des programmes il est utile d'avoir un accès direct à ces informations.

Cette déclaration est suivie par la définition d'un processus `p` qui affiche `Present` lorsque le signal `s` est émis :

```
let process p =
  await s;
  print_string "Present";
  print_newline()
```

[1] Je tiens à remercier Jean-Ferdy Susini de m'avoir conseillé de profiter de la boucle d'interaction de Ocaml pour implanter les Reactive Scripts.

```
louis@vesuve:/tmp# rmltop
        ReactiveML version 1.04.04-dev
        Objective Caml version 3.08.3

# signal s;;
val s : ('_a , '_a list) event
val s : ('_a, '_a list) Sig_env.Record.t * '_b list ref * '_c list ref =
  ({Sig_env.Record.status = -2; Sig_env.Record.value = [];
    Sig_env.Record.pre_status = -2; Sig_env.Record.pre_value = [];
    Sig_env.Record.default = []; Sig_env.Record.combine = <fun>},
   {contents = []}, {contents = []})
# let process p =
    await s;
    print_string "Present";
    print_newline()
  ;;
val p : unit process
val p :
  unit ->
  (unit -> unit) ->
  Implantation.Lco_ctrl_tree_record.contol_tree ->
  unit Implantation.Lco_ctrl_tree_record.step = <fun>
# #run p;;
- : unit = ()
# #emit s ();;
- : unit = ()
Present
```

FIG. 11.1 – Session `rmltop`

Une instance du processus `p` est ensuite exécutée en utilisant la directive `#run p`. Les directives sont des instructions ajoutées au langage pour pouvoir interagir directement avec la machine réactive qui exécute le programme.

En exécutant la directive `#emit s` le signal `s` est émis dans la machine réactive. Donc l'instance de `p` qui attendait ce signal affiche le message `Present`.

La directive principale est `#run p;;` qui exécute le processus `p`. À partir de cette directive, deux nouvelles directives ont été construites. La directive `#emit` émet un signal et `#exec` exécute une expression réactive :

```
#emit s v;;  ≡  #run (process (emit s v));;
#exec e;;    ≡  #run (process e);;
```

Il existe également des directives qui contrôlent la machine d'exécution de programmes réactifs. Les directives `#suspend;;` et `#resume;;` permettent d'arrêter l'exécution du programme et de la reprendre. `#sampling n;;` change la vitesse d'échantillonnage. `#step_by_step;;` passe dans un mode où le programme est exécuté instant par instant. Dans ce mode, la direc-

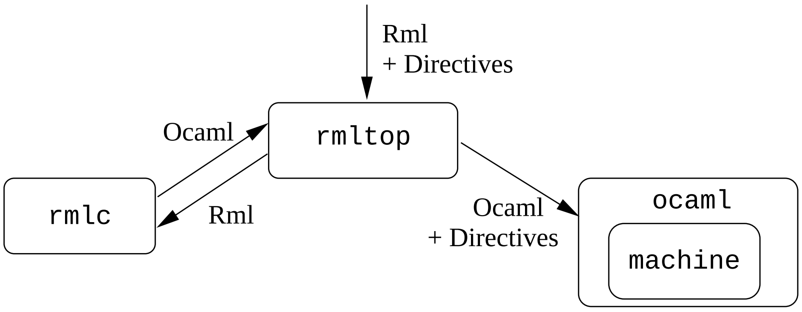


FIG. 11.2 – Structure de l'implantation du mode interactif de REACTIVEML (`rmltop`).

tive `#step;;` exécute un instant. Pour revenir dans le mode échantillonné, il y a la directive `#sampled;;`.

Enfin, que cela soit pour REACTIVEC ou les SUGARCUBES, les REACTIVE SCRIPTS proposent un nouveau langage avec une syntaxe et une sémantique propres. Au contraire, pour REACTIVEML tous les programmes écrits dans le mode interactif peuvent être compilés et tous les programmes REACTIVEML peuvent être exécutés dans le mode interactif.

Implantation

L'implantation du mode interactif de REACTIVEML a la structure présentée figure 11.2. Un processus `rmltop` coordonne l'exécution parallèle d'un compilateur `rmlc` et d'une boucle d'interaction `ocaml`. Les phrases REACTIVEML données en entrée sont envoyées au compilateur `rmlc` qui retourne du code OCAML. Ce code OCAML est envoyé à la boucle d'interaction pour être exécuté. À l'intérieur de la boucle d'interaction, il y a un *toplevel* qui évalue les définitions et enrichit l'environnement et il y a également un processus léger qui exécute une machine réactive dans le même environnement. Cette machine exécute le programme réactif et interprète les directives données en entrée.

Comme il est montré dans [52], le contrôle de l'exécution d'un programme réactif est un système réactif. Ainsi, le contrôle de la machine qui exécute le programme réactif se fait par un processus écrit en REACTIVEML. Le cœur du contrôleur est le processus `machine` qui fait passer les instants. Il est composé de deux modes : (1) le mode échantillonné et (2) le mode pas à pas. Il doit passer du premier au second lorsque le signal `step_by_step` est émis et du second au premier quand `sampled` est émis. Lorsque la machine est dans le premier mode, l'exécution est suspendue et reprise par l'émission du signal `suspend_resume`.

```
let process machine =
  loop
    do
      control
        loop Rml_machine.react(); pause end
      with suspend_resume
    until step_by_step done;
    do
      loop await step; Rml_machine.react() end
    until sampled done
  end
val machine : unit process
```

La fonction `Rml_machine.react()` exécute un instant de l'interprète de programmes réactifs.

11.2 Interface avec d'autres langages

Nous avons vu qu'il était possible d'appeler des fonctions OCAML si nous connaissions leur interface. Nous allons voir ici comment REACTIVEML peut être interfacé avec d'autres langages et plus particulièrement avec d'autres langages réactifs.

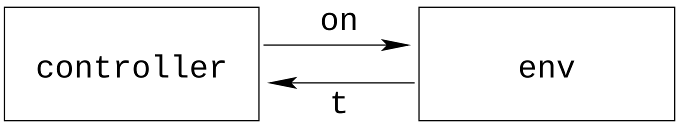
L'expérience de Lucky

LUCKY [58] est un langage de description de systèmes réactifs non déterministes. Ce langage a été développé à l'origine pour le test de programmes réactifs. Il est utilisé pour décrire l'environnement du système testé et le simuler.

L'intérêt de pouvoir appeler des programmes LUCKY dans un programme REACTIVEML est de construire des simulations comprenant le système réactif et son environnement ou de décrire toute partie non déterministe d'une application REACTIVEML.

Un exemple simple est celui du test d'un contrôleur de chaudière. Supposons que le processus REACTIVEML `controller` détermine en fonction de la température extérieure s'il faut allumer la chaudière. Nous voulons simuler ce programme dans un environnement où, lorsque la chaudière est allumée, la température de l'environnement augmente, et diminue lorsque la chaudière est éteinte.

Pour simuler le contrôleur en REACTIVEML, il faut l'exécuter en parallèle avec son environnement :



Le comportement de l'environnement n'est pas déterministe et peut simplement être modélisé avec le programme LUCKY suivant :

```
inputs { on: bool }
outputs { t: float }
locals { delta: float ~alias 0.5 }
nodes { init: stable; s : stable }
start_node { init }

transitions {
   init -> s ~cond t = 17.0;
   s -> s ~cond
     if on
     then 0.0 <= (t - pre t) and (t - pre t) <= delta
     else -delta <= (t - pre t) and (t - pre t) <= 0.0
}
```

Ce programme LUCKY prend en entrée `on`, la commande de la chaudière, et il calcule la température `t`. Le comportement de l'environnement est décrit par un automate à deux états. L'état `init` fixe la température initiale et passe dans l'état `s`. Dans l'état `s`, la température est donnée par un ensemble de contraintes. Si la chaudière est allumée, la température augmente d'une valeur comprise entre `0` et `delta`. Sinon, la température diminue de façon similaire. L'exécution de ce programme LUCKY tire aléatoirement à chaque instant une valeur pour `t` qui satisfait les contraintes en fonction de l'entrée.

Nous proposons une façon simple de créer des processus REACTIVEML à partir de programmes LUCKY. Par exemple, nous créons un processus `env` à partir de l'exemple précédent en écrivant :

```
external.luc env { on: bool } { t: float } = ["heater.luc"]
```

`env` est le nom du processus créé. `on` et `t` reprennent l'interface du programme LUCKY. À droite du symbole égal se trouve la liste des fichiers LUCKY qui définissent le processus.

Le processus `env` créé a la signature suivante :

```
val env : ('a, bool) event -> (float, 'b) event -> unit process
```

Le premier paramètre représente les entrées du processus. Ici, c'est un signal sur lequel la valeur lue doit être de type `bool`. Le second paramètre est un signal sur lequel sont émises les valeurs calculées par LUCKY.

Ce processus `env` peut être utilisé comme tous les autres processus REACTIVEML. Son comportement à chacune de ses activations est de lire la valeur précédente du signal d'entrée et d'émettre la valeur calculée par LUCKY. Intuitivement, la traduction en REACTIVEML de la déclaration du processus `env` est la suivante :

```
let process env on t =
  let state = ref (Luc4ocaml.make ["heater.luc"]) in
  loop
    pause;
    let v = Luc4ocaml.step_se state ["on", pre ?on] in
    emit t v
  end
val env : ('a, bool) event -> (float, 'b) event -> unit process
```

Le module `Luc4ocaml` est l'interface OCAML de LUCKY. Au premier instant, l'état du processus LUCKY est alloué. Puis à chaque instant, la fonction de transition est appelée avec en entrée la valeur précédente de `on`. La valeur ainsi calculée est émise sur `t`.

Dans cet exemple, l'entrée booléenne `on` est représentée par un signal qui produit des valeurs booléennes. La présence et l'absence des signaux peuvent également être utilisées pour représenter les flots booléens. Dans ce cas, il faut déclarer l'entrée ou la sortie avec le type `event`.

```
external.luc env { on: event } { t: float } = ["heater.luc"]
val env : ('a, 'b) event -> (float, 'c) event -> unit process
```

L'interface avec LUCKY est utilisée par Ludovic Samper (France Telecom/Verimag) dans un simulateur de réseaux de capteurs [96]. Le but du réseau est de détecter la présence de nuages toxiques. Dans ce simulateur, le comportement de chaque nœud et les communications entre les nœuds sont programmés en REACTIVEML, mais la position des nuages est programmée en LUCKY. Une fois importés depuis LUCKY, les nuages sont considérés comme des processus REACTIVEML, ainsi des nuages peuvent être créés ou détruits dynamiquement pendant la simulation.

Nous envisageons d'interfacer ReactiveML avec d'autres langages réactifs comme Lustre et Lucid Synchrone en utilisant une approche similaire. Comme pour Lucky, cela permettra d'utiliser les spécificités de ces langages pour programmer des parties de programmes ReactiveML. Ainsi des comportements s'exprimant mieux avec une approche flots de données pourrons être décrits directement dans ces langages et composés au niveau ReactiveML pour profiter de ses aspects dynamiques.

Cette intégration d'autres langages dans ReactiveML peut être vue comme une première étape pour son utilisation comme un langage de composition de programmes réactifs hétérogènes. Une application peut être la description de scénarios de simulation. Prenons par exemple le simulation d'un réseau de capteurs dont le code embarqué sur les capteurs est écrit en Lustre. Les capteurs sont alors combinés entre eux et avec un environnement modélisé en Lucky au niveau ReactiveML. L'évolution de la topologie du réseaux est décrite en utilisant les aspects dynamiques du modèle réactif.

11.3 Configurations événementielles

Les configurations événementielles (ou configurations) sont des expressions booléennes sur la présence des signaux. Par exemple, l'expression :

```
await ((s1 /\ s2) \/ s3)
```

attend que `s1` et `s2` soient présents simultanément ou que `s3` soit émis.

En ReactiveML, les configurations sont formées de conjonctions ($\wedge$) et de disjonctions ($\vee$). Elles peuvent être utilisées dans toutes les constructions qui testent la présence d'un signal : `await`, `await/immediate`, `present`, `do/when`, `control/with` et `do/until`. Pour le `do/until`, le cas où la valeur des signaux est récupérée n'est pas considéré. Nous discuterons dans la prochaine section de ce cas.

Les configurations de ReactiveML ne sont pas des expressions de première classe. En particulier, nous ne pouvons pas écrire :

```
let c = s1 /\ s2 in await c
```

Ce choix a été guidé par des problèmes d'inférence de type. Par exemple, si l'on définit le processus `my_await`, alors `s` est de type `event` et ne peut pas être une configuration.

```
let process my_await s = await s
val my_await : ('a, 'b) event -> unit process
```

Nous avons envisagé de différencier l'attente d'un configuration de l'attente d'un signal en ajoutant le symbole `?` devant les signaux. Ainsi le processus `await_sig` qui attend un signal et le processus `await_config` qui attend une configuration pourrait être défini par :

```
let process await_sig s = await ?s
val await_sig : ('a, 'b) event -> unit process

let process await_config c = await c
val await_config : 'a configuration -> unit process
```

Les configurations événementielles existent en SugarCubes ou en Junior, mais ne sont pas présentes dans des langages comme Loft et ULM. Pour le moment, les configurations de ReactiveML sont moins expressives que celles de Junior : nous n'autorisons pas les négations.

Le problème de la négation est qu'elle se marie mal avec les tests instantanés. Cela pose des problèmes de causalité. Par exemple, le programme suivant n'est pas causal :

```
await immediate (not s); emit s
```

En effet, si `s` est supposé absent, alors `s` est émit. Si `s` est supposé présent, la condition du `await` n'est pas satisfaite donc `s` n'est pas émis. Dans tous les cas, l'hypothèse est contredite.

De même, nous ne savons pas donner de sémantique à

```
do emit s when (not s)
```

Pour l'expression `present`, la situation est un peu différente. La question qui se pose est de savoir quelle est la branche qui ne peut pas être exécutée instantanément. Par exemple, l'expression :

```
present not s then e1 else e2
```

peut exécuter instantanément sa branche `else` et doit ajouter un délai avant l'exécution de la branche `then`. De plus, si des tests de présence et d'absence sont mélangés, l'ajout d'un délai peut dépendre de l'exécution. Dans l'expression suivante, la branche `then` peut être exécutée instantanément si `s1` est émis ou avec un délai si `s1` et `s2` sont absents.

```
present s1 \/ not s2 then e1 else e2
```

La solution que nous proposons actuellement est d'interdire les négations. Nous pourrions envisager des solutions différentes comme interdire les négations uniquement dans les constructions `await/immediate`, `do/when` et `present` et les autorisées dans les constructions comme le `do/until`. La négation ne pose pas de problèmes sémantiques dans les constructions qui peuvent attendre la fin d'instant pour tester leur configuration événementielle.

Une autre solution pour accepter les négation dans la construction `present` est de changer sa sémantique en ajoutant un délai avant l'exécution des deux branches dans tous les cas et proposer une construction `present/immediate` qui aurait la sémantique classique de `present`. De plus la construction `present` deviendrait alors symétrique.

Implantation

L'extension de la sémantique avec les configurations est directe. Cela se fait comme dans le chapitre 6.4.1. En revanche, proposer une implantation efficace n'est pas facile.

La compilation d'une configuration fournit une formule booléenne et la liste des signaux dont elle dépend. Ainsi une expression qui teste une configuration a un accès direct à l'ensemble des files d'attente dans lesquelles elle doit s'enregistrer.

Lorsqu'une expression est bloquée sur une configuration, elle n'enregistre pas directement sa fonction de transition dans les files d'attente mais une référence vers cette fonction. Lorsqu'un signal de la configuration est émis, la référence est sortie de la file d'attente et la fonction de transition est exécutée. Si la configuration n'est pas satisfaite, la référence est enregistrée de nouveau dans la file d'attente. En revanche, si la configuration est satisfaite, la fonction de transition fait pointer la référence enregistrée dans les autres files d'attente vers une fonction qui ne fait rien. Ainsi, il n'y a pas besoin de désabonner la fonction de transition des autres files d'attente.

11.4 Filtrage de la valeur des signaux

Jusqu'à maintenant, nous avons supposé que les motifs utilisés dans les constructions `await/in` et `do/until` sont exhaustifs. Nous traitons ici le comportement des motifs non exhaustifs comme `await s(1) in e`.

La première solution que nous avons proposée a été d'afficher un avertissement de motif non exhaustif à la compilation et produire une erreur à l'exécution si la valeur émise sur le signal n'était pas filtrée par le motif. Dans ce cas, l'expression `await s(1) in e` est équivalente à

```
await s(x) in let 1 = x in e
```

Le comportement actuellement implanté est différent. Le filtrage sert de garde. Par exemple, l'expression `await s(1) in e` peut commencer l'exécution de `e` uniquement si `s` est émis et que sa valeur est égale à `1`.

Ce comportement permet de programmer la construction `await/one/in`. Une des valeurs émise sur le signal peut être récupérée en filtrant la liste valeurs émises par la liste qui a au moins un élément :

`await one s(x) in ...` ≡ `await s(x::_) in ...`

Si le signal `s` utilise la fonction de combinaison par défaut, la construction `await/one/in` peut être généralisée avec un nombre arbitraire de valeur. Par exemple, nous pouvons attendre qu'un signal soit émis au moins trois fois ou exactement trois fois pendant un instant :

```
await s (x1 :: x2 :: x3 :: _) in ...
await s ([x1; x2; x3]) in ...
```

Il faut remarquer l'ordre de `x1`, `x2` et `x3` ne correspond pas nécessairement à l'ordre des émissions et peut changer en fonction de l'interprète utilisé. Pour rester déterministe, il faut que les opérations faites sur ces trois valeurs soient commutatives. En particulier, nous pouvons remarquer que si les motifs qui filtrent les éléments de la liste ne sont pas exhaustifs, le programme n'est pas déterministe. Par exemple l'expression suivante n'attend pas que au moins un `42` soit émis, mais que la valeur `42` soit émise et que la fonction de combinaison l'ait placée en tête de la liste :

```
await s(42::_) in ...
```

Nous proposerons peut-être des mécanismes de filtrage plus évolué comme ceux utilisés dans [33]. Ils permettent par exemple d'utiliser les propriétés des multi-ensembles dans le filtrage et donc de résoudre le problème précédent. Le danger de ce type de méthode est d'encourager l'utilisation de constructions du langage que nous ne savons pas exécuter efficacement.

Une autre perspective moins ambitieuse est d'ajouter des gardes `when` dans les motifs comme dans la construction `match/with`. Attendre que la valeur `42` soit émise s'écrirait alors :

```
await s(x when List.mem 42 x) in ...
```

Cet ajout ne pose pas de problème sémantique et permet de tester des gardes arbitrairement compliquées.

La construction `await/immediate/in`

Nous venons de voir que l'on peut généraliser la construction `await/one/in` en utilisant le filtrage. Nous allons étudier le cas de `await/immediate/one/in` pour voir si nous pouvons proposer une méthode similaire dans le cas instantané.

Remarque :

> Contrairement à la version non immédiate de `await/one/in`, le motif utilisé dans la construction `await/immediate/one/in` doit être exhaustif pour ne pas causer d'erreurs à l'exécution. Ce choix est fait pour éviter les problèmes de non-déterminisme évoqués précédemment.

Nous proposons d'ajouter au langage la construction `await/immediate/in` dans laquelle des contraintes sont imposées sur la forme des motifs. Avec cette nouvelle construction, le codage de `await/immediate/one/in` est le même que celui de `await/one/in` :

```
await immediate one s(x) in ...  ≡  await immediate s(x::_) in ...
```

Nous autorisons également d'attendre un nombre minimum de valeurs :

```
await immediate s (x1 :: x2 :: x3 :: _) in ...
```

En revanche, attendre un nombre fixe de valeurs est interdit pour des raisons de causalité. Dans l'exemple suivant, nous pouvons réagir s'il y a exactement trois valeurs émises et dans ce cas une quatrième valeur est émise ce qui contredit l'hypothèse précédente :

```
await immediate s ([x1; x2; x3]) in emit s 4
```

Nous interdisons de récupérer la valeur de la fin d'une liste :

```
await immediate s (x1 :: x2 :: x3 :: x) in ...
```

Cette construction est un cas particulier de l'expression `await immediate s (x) in ...` qui récupère instantanément toutes les valeurs émises pendant un instant. Par définition du modèle réactif, nous savons cette construction impossible.

Finalement, les seuls motifs acceptés dans la construction `await/immediate/in` sont les listes *d'au moins n éléments* définies par :

$$\begin{array}{rcl} \textit{pattern} & ::= & \_ \mid \textit{exhaustive-pattern}\,\texttt{::}\,\textit{pattern} \\ \textit{exhaustive-pattern} & ::= & \_ \mid x \mid (\textit{exhaustive-pattern}, \textit{exhaustive-pattern}) \end{array}$$

Configurations événementielles

Nous nous intéressons maintenant à la combinaison des configurations événementielles et de la récupération de valeurs. Nous voulons par exemple attendre que les signaux `s1` et `s2` soient émis simultanément et calculer leur somme :

```
await s1(x) /\ s2(y) in x + y
```

La conjonction de signaux valués ne cause pas de problèmes sémantique. Il faut que tous les signaux soient présents et que toutes les valeurs puissent être filtrées pour que la configuration soit satisfaite. Les noms introduits dans le corps du `await/in` sont l'union des noms introduits par chaque motif et il faut que l'intersection des noms introduits par chaque motif soit vide.

La sémantique de la disjonction est plus problématique. Illustrons cela sur un exemple :

```
await s1(x) \/ s2(x) in x
```

Dans cet exemple, si `s1` est présent et `s2` est absent, alors `x` prend la valeur de `s1`. Si `s2` est présent et `s1` est absent, alors `x` prend la valeur de `s2`. Le problème est quelle est la valeur de `x` lorsque `s1` et `s2` sont présents ? Sémantiquement, la disjonction est commutative, il n'y a pas de raison de privilégier la valeur de `s1` ou `s2`. Pour cette raison, nous pensons autoriser uniquement les conjonctions dans les constructions `await/in` et `do/until`.

Une autre alternative serait de proposer un opérateur ou exclusif garantissant qu'un seul des deux signaux est présent. Nous pouvons remarquer que dans l'opérateur ou exclusif, il y a un test d'absence de signal. Mais cet opérateur peut être autorisé dans les `await/in` et les `do/until` car le test de la condition peut être fait à la fin d'instant.

11.5 La composition parallèle séquentielle

Nous avons choisi en ReactiveML que la composition parallèle soit associative et commutative. Par conséquent, l'ordre d'activation des branches parallèles n'est pas connu. Néanmoins, dans certain cas, il peut être utile de pouvoir spécifier cet ordre. C'est souvent le cas en présence d'effets de bord.

Nous proposons l'opérateur `|>` de composition parallèle séquentielle. Il exécute ses deux branches en parallèle en garantissant que la branche gauche est toujours exécutée avant la branche droite. Il permet par exemple la définition du processus `print_instants` qui affiche le numéro de l'instant avant l'exécution d'un processus `p`.

```
let process print_instants p =
  let cpt = ref 0 in
  loop
    incr cpt;
    print_string ("******␣Instant␣"^(string_of_int !cpt)^"␣******␣\n");
    pause
  end
  |>
  run p
val print_instants : 'a process -> unit process
```

Contrairement à l'opérateur `merge` des SugarCubes, l'exécution de la branche droite d'un `|>` ne peut pas activer une expression de la partie gauche. Ainsi, les problèmes de compositionnalité présentés chapitre 10.1 sont évités. En particulier, il est impossible d'écrire l'exemple de la cascade inverse avec l'opérateur `|>`.

Afin de garantir l'absence de dépendance de droite à gauche, nous n'avons pas fait d'analyse de causalité. Nous utilisons un critère syntaxique. Tous les tests de signaux dans les parties gauches des parallèles séquentiels sont simplement interdits. Ainsi, les expressions qui se trouvent dans les parties gauches sont : les expressions instantanées ($0 \vdash e$), `loop`, `pause` et les compositions `||`, `;` et `|>`. C'est une analyse grossière, mais elle est à fois simple et modulaire. De plus, ReactiveML autorisant des constructions comme les références et les signaux qui peuvent émettre des signaux, il est difficile de faire une analyse de causalité dans ce langage.

Nous avons envisagé d'introduire l'opérateur `<|` qui garantit que la branche droite est exécuté après la branche gauche. Dans ce cas, toutes les constructions du langage dans la partie droite sauf l'émission de signal peuvent être utilisées. Ainsi l'exécution de la branche droite ne peut pas débloquer l'exécution d'une expression de la partie gauche. Le problème pour l'implantation de cette construction est qu'il est difficile d'interdire les émissions de signaux car tous les appels de fonctions peuvent émettre potentiellement un signal.

11.6 Automates

Nous souhaitons ajouter à ReactiveML une construction d'automate similaire à celle de Lucid Synchrone [34]. Voici par exemple comment nous pourrions redéfinir le processus `machine` (cf. chapitre 11.1) qui contrôle l'exécution du programme réactif dans le mode interactif.

```
let process machine =
  automaton
  | Sampled ->
     do
       control
         loop Rml_machine.react(); pause end
       with suspend_resume
     until step_by_step then Step_by_step done
  | Step_by_step ->
     do
       loop await step; Rml_machine.react() end
     until sampled then Sampled done
  end
val machine : unit process
```

C'est un automate à deux états (`Sampled` et `Step_by_step`). Le premier état est l'état initial. Il exécute à chaque instant la fonction `Rml_machine.react` lorsqu'il n'est pas suspendu par la construction `control/with`. Dans l'état `Step_by_step`, la fonction `Rml_machine.react` est appelée à chaque fois que le signal `step` est émis. Le passage de l'état `Sampled` à l'état `Step_by_step` se fait lorsque le signal `step_by_step` est émis. Le retour dans l'état `Sampled` se fait lorsque `sampled` est émis.

L'entrée dans les états de cet automate se fait par réinitialisation. Cela signifie qu'à chaque fois que l'on entre dans un état son code est exécuté du début. Ici, à chaque fois que l'on entre dans l'état `Sampled`, le `control/with` est actif. Si nous voulons entrer dans un état en reprenant l'exécution au point où elle en était, nous pouvons utiliser l'entrée par histoire. Dans cet exemple, si la ligne :

```
until sampled then Sampled done
```

est remplacée par :

```
until sampled continue Sampled done
```

alors l'entrée dans l'état `Sampled` conserve le statut qu'avait la construction `control/with` en quittant l'état.

La syntaxe des automates est la suivante :

$$
\begin{array}{rcl}
e & ::= & \dots \\
 & & \mid \texttt{automaton}\ \mathit{State}\ \texttt{->}\ \texttt{do}\ e\ u\ \texttt{...|}\ \mathit{State}\ \texttt{->}\ \texttt{do}\ e\ u\ \texttt{end} \\
u & ::= & \ \ \texttt{until}\ e(x)\ \texttt{then}\ \mathit{State}\ u \\
 & & \mid \texttt{until}\ e(x)\ \texttt{continue}\ \mathit{State}\ u \\
 & & \mid \texttt{until}\ e(x)\ \texttt{->}\ e\ u \\
 & & \mid \texttt{done}
\end{array}
$$

Le langage des expressions est étendu avec la construction `automaton/end`. Un automate est composée d'une liste d'états. Chaque état a la forme *State* `->` `do` e u. *State* est le nom de l'état. L'expression e définit le corps de l'état et u représente la liste des conditions d'échappement. Il y a trois types d'échappements : `until/then`, `until/continue` et `until/->`. Les trois définissent des préemptions faibles. La forme `until/then` change d'état en réinitialisant l'état cible. La forme `until/continue` entre dans un nouvel état par histoire. La forme `until/->` termine l'exécution de l'automate. Contrairement à Lucid Synchrone, il n'y a pas de préemption forte (`unless`) afin de rester cohérent avec le modèle réactif.

Les automates de ReactiveML par rapport à ceux de Lucid Synchrone peuvent terminer leur exécution. Si le corps d'un état se réduit en une valeur v alors l'automate termine instantanément et renvoie la valeur v. De même, si une condition d'échappement `until/->` est satisfaite, l'automate termine à l'instant suivant en exécutant l'expression de traitement d'échappement.

Résumé

Ce chapitre commence avec la présentation de la boucle d'interaction de ReactiveML : `rmltop`. Cette méthode d'exécution de programmes réactifs reprend l'idée des Reactive Scripts où le comportement d'un processus peut être programmé dynamiquement. Cette boucle d'interaction peut servir d'outil de mise au point de programmes ReactiveML.

L'implantation de `rmltop` repose sur l'utilisation de la boucle d'interaction de Ocaml dans laquelle une machine réactive est exécutée dans un thread. `rmltop` récupère en entrée le fichier source ReactiveML, le compile avec `rmlc` et exécute le code généré dans le toplevel Ocaml.

Nous pouvons remarquer que le programme qui contrôle l'exécution de la machine réactive est un programme ReactiveML.

La seconde section est consacrée à l'interface de ReactiveML avec Lucky, un langage de description de comportements non déterministes. Mise à part l'utilité de cette interface, cette expérience nous permet d'envisager d'utiliser ReactiveML comme un langage de composition de processus réactifs écrits dans plusieurs langages.

Enfin, nous avons présenté de nouvelles constructions pour ReactiveML : les configurations événementielles, le filtrage de la valeur des signaux, la composition parallèle séquentielle et les automates.

Chapitre 12

Conclusion

12.1 Résumé

Ce document a présenté ReactiveML un langage pour la programmation de systèmes réactifs. Ce langage étend Ocaml avec une opération de composition de programmes qui ne terminent pas : la composition parallèle. Le modèle de la concurrence utilisé par ReactiveML est le modèle réactif de Frédéric Boussinot.

ReactiveML est particulièrement adapté à la programmation de systèmes qui évoluent au cours du temps : les processus peuvent être créés et détruits dynamiquement, et les canaux de communication entre les processus peuvent évoluer dynamiquement. Ainsi le langage est utilisé pour la simulation de réseaux ad hoc où la taille et la topologie ne peuvent pas être déterminées statiquement.

Conception

ReactiveML étend un langage de programmation généraliste afin de bénéficier des structures de données (tableaux, enregistrements, ...) et des structures de contrôle (boucles, conditionnelle, récursion, ...), qui sont essentielles pour programmer de gros systèmes.

Par ailleurs le modèle réactif correspond à nos besoins. Ce modèle étant basé sur le modèle synchrone, il est à la fois simple et déterministe. Les restrictions apportées par le modèle réactif par rapport au modèle synchrone permettent de supprimer les problèmes de causalité qui sont difficiles à analyser dans le cadre d'un langage généraliste.

Les choix de conception du langage ont d'abord été présentés au chapitre 2, puis ils ont été discutés au chapitre 10. Ces choix se sont à la fois appuyés sur (1) notre expérience de programmation en ReactiveML, (2) la sémantique et (3) la possibilité d'implanter efficacement les nouvelles constructions proposées.

Un exemple de choix est d'avoir distingué les fonctions dont l'exécution prend du temps des fonctions du langage hôte qui sont supposées instantanées. Les fonctions réactives sont définies par des processus. Dans les définitions de processus, toutes les nouvelles constructions ajoutées à ReactiveML peuvent être utilisées. Cette séparation permet à l'utilisateur d'identifier simplement les machines à état des fonction instantanées. Du point de vue de l'implantation, la séparation garantit que l'exécution des fonctions instantanées est aussi efficace que du code Ocaml.

Sémantique

La sémantique formelle de ReactiveML est définie dans partie II. La sémantique comportementale (chapitre 3) présente l'exécution d'un instant en une réaction. Cette approche abstrait l'ordonnancement à l'intérieur de l'instant. Ainsi la preuve de propriétés comme le déterminisme est plus simple à effectuer.

La sémantique opérationnelle (chapitre 4) quant à elle présente la réaction d'un instant comme une succession de petites réactions. Elle permet la description des différentes stratégies d'ordonnancement possibles d'un programme. Elle est ainsi un bon formalisme pour prouver la correction d'une implantation. Cette sémantique est également utilisée au chapitre 5 pour prouver la sûreté du système de type.

La preuve d'équivalence entre la sémantique comportementale et la sémantique opérationnelle est présentée à la fin du chapitre 4. Les deux sémantiques étant de nature différente, leur équivalence offre de ce fait un cadre sémantique large. L'utilisateur peut choisir le formalisme en fonction de la propriété à prouver.

De plus, ces sémantiques prennent en compte l'ensemble du langage. Elles décrivent à la fois les parties réactives et combinatoire. Cela permet de décrire formellement les signaux valués et ainsi de mettre en évidence le phénomène d'échappement de portée.

L'étude de la sémantique du langage a eu un impact sur la conception du langage. L'exemple le plus marquant est le choix de ML comme langage hôte. Ce choix s'est imposé de lui même pour avoir un langage généralise expressif et dont la sémantique formelle est simple. Nous avons essayé autant que possible de garder une sémantique simple lors de l'ajout des nouvelles constructions.

Par ailleurs, l'étude de la sémantique de ReactiveML permet de lever les ambiguïtés sur les constructions du langage et les interactions entre les parties combinatoires et réactives. Ceci permet d'augmenter la confiance dans le code produit par le compilateur.

Enfin, dans les chapitres 6 et 7, les techniques d'implantation que nous avons utilisées sont décrites formellement. Ce travail permet à la fois de mieux comprendre l'implantation mais est également une étape importante pour le lien entre les sémantiques précédentes et le code qui est exécuté.

Implantation

La partie III du document présente des techniques d'ordonnancement efficace de programmes réactifs et leur implantation. Le chapitre 6 présente Glouton, la première implantation du modèle réactif que nous avons réalisée. Cette implantation tire son nom de l'idée qu'il faut stocker les expressions en attente d'un signal pour pouvoir exécuter un programme réactif efficacement.

Dans le chapitre 7, l'implantation actuelle de ReactiveML est décrite. Elle est basée sur l'utilisation d'un langage intermédiaire avec continuations : L_k. Comme pour Glouton, la sémantique est basée sur l'utilisation de files d'attentes et le point délicat est la prise en compte des constructions de contrôle `do/when` et `do/until`.

La partie III se termine au chapitre 8 avec l'implantation de L_k et la présentation d'une bibliothèque pour la programmation réactive en Ocaml. L'implantation de L_k dans un langage purement fonctionnel se déduit directement des règles de sémantique. Puis l'implantation efficace en Ocaml est présentée. Enfin, la présentation de la bibliothèque montre comment nous avons

% de cellules actives	0 %	4 %	42 %	60 %	83 %
Ocaml	0.74 s	0.75 s	0.76 s	0.77 s	0.77 s
Loft	0.02 s	0.11 s	0.93 s	1.57 s	2.09 s
ReactiveML	0.05 s	0.08 s	0.89 s	1.46 s	1.94 s

Fig. 12.1 – Temps d'exécution moyen d'un instant pour un automate cellulaire de Fredkin de taille 500x500.

pu expérimenter simplement différentes techniques d'ordonnancement sans avoir à modifier le compilateur.

Du point de vue de l'ordonnancement, la propriété principale pour exécuter efficacement un programme est l'absence d'attente active. Cela évite de tester plusieurs fois la présence d'un même signal dont le statut ne peut pas être déterminé.

Cette propriété s'illustre bien sur l'exemple de Frédéric Boussinot des automates cellulaires [22]. Dans l'implantation de l'automate, les cellules inactives attendent un signal d'activation de sorte que seules les cellules actives sont exécutées. La figure 12.1 compare le temps d'exécution pour Loft, ReactiveML et une version impérative écrite en Ocaml. La version impérative parcourt à chaque instant toutes les cellules de l'automate avec des boucles `for`. Les chiffres de la figure 12.1 montrent que l'implantation de ReactiveML est aussi efficace que celle de Loft qui est écrite en C et lorsqu'il y a peu de cellules actives, l'approche paresseuse implantée en ReactiveML et en Loft est plus efficace que la version impérative.

Il est important de remarquer que l'exécution efficace de programmes ReactiveML ne repose pas uniquement sur les techniques d'ordonnancement. La représentation de la structure de contrôle du programme pour pouvoir accéder rapidement aux branches parallèles à exécuter ou l'accès aux signaux sont aussi des facteurs déterminants. En ReactiveML, une attention particulière a été portée sur ces points.

Le simulateur de réseau mobile ad hoc présenté chapitre 9 a permis de tester le langage sur un exemple complet. Il est sorti de cette expérience que les performances de l'interprète ont été suffisantes pour réaliser les simulations qui intéressaient l'équipe réseaux.

Réalisations logicielles

Au cours de cette thèse, un travail important d'implantation a été réalisé. Il a commencé par une implantation de Junior qui s'appelle Glouton. Cette implantation a une efficacité comparable avec les autres implantations de cette bibliothèque Java. Des tests de performance ont été réalisés dans [1, 28].

Un compilateur ReactiveML est disponible. Il prend en compte l'ensemble du langage présenté dans ce document et une partie des extensions du chapitre 11.

Autour du langage, il existe d'autres outils comme : le mode interactif présenté au chapitre 11.1, la bibliothèque Ocaml pour la programmation réactive, le mode Emacs réalisé par Sarah Maarek [65] et des exemples de programmes (une partie des exemples ont été réalisés par Matthieu Carlier [29]).

Enfin, le langage est utilisé pour la simulation de réseaux mobiles ad hoc par Farid Benbadis de l'équipe Réseaux et Performances du Laboratoire d'Informatique de Paris 6 (LIP6) [6, 7, 8, 67] et par Ludovic Samper dans le carde d'une thèse CIFRE entre France Telecom et le laboratoire Verimag (Grenoble) pour la simulation de réseaux de capteurs [96].

12.2 Perspectives

Nous avons vu chapitre 11 des modifications que nous envisageons d'apporter à ReactiveML à court terme. Nous présentons ici des axes de recherche à développer.

12.2.1 La composition asynchrone

Le mélange des aspects synchrones et asynchrones est une question importante. Le premier besoin qui apparaît lors du développement d'applications réactives est la gestion asynchrone des entrées/sorties. Actuellement, une fonction bloquante appelée dans un processus réactif bloque l'exécution de la machine réactive car ReactiveML est basé sur un ordonnancement coopératif. La bibliothèque de threads de Ocaml peut être utilisée, mais nous souhaitons proposer des constructions spécifiques mieux intégrées dans le langage.

Loft et les Fair Threads proposent une solution à ce problème. Dans ces langages, un thread peut être exécuté dans un mode synchrone ou asynchrone et il peut changer de mode en cours d'exécution. Ainsi pour faire une entrée/sortie, on peut créer un thread qui permet la communication entre les processus synchrones et leur environnement.

Un travail de conception de langage est à faire pour trouver des constructions qui soient simples à utiliser et suffisamment expressives. Les threads de service des Fair Threads en Scheme sont une proposition intéressante [97]. Mais nous pouvons également imaginer d'autres solutions comme un programme principal paramétré par des signaux externes.

Une autre approche du mélange réactif/asynchrone est celle suivie par ULM. Dans ce cas, le modèle réactif est utilisé pour la réalisation d'applications distribuées. Sur chaque site une machine réactive synchrone est exécutée, et les communications entre les sites sont asynchrones. En ULM, ces communications se font par migration de processus.

L'intérêt du modèle réactif pour programmer ce type de système est de pouvoir donner une sémantique simple au comportement de chaque site tout en étant expressif. Il est possible par exemple de faire de la reconfiguration dynamique. C'est-à-dire modifier le comportement d'une application sans arrêter le service qu'elle fournit.

Pour ces deux types de mélange réactif/asynchrone, en cas de programmation de systèmes ouverts où des processus peuvent être ajoutés dynamiquement, la question de contrôle de ressource se pose. En particulier comment garantir qu'un programme n'a pas des instants "trop longs". Un processus dont le temps d'exécution d'un instant est beaucoup plus long que celui des autres processus parallèles pénalise tous les autres.

Le premier point à étudier est l'évaluation de la durée d'exécution d'un instant. Puis une piste intéressante à explorer est le *retiming* : comment découper un instant en ajoutant des instructions `pause` dans un programme sans changer sa sémantique ? Cette transformation permet de faire des instants dont la durée est plus courte. Le retiming est une transformation classique dans les circuits. Un exemple simple en ReactiveML est qu'il est toujours possible d'ajouter des `pause` dans un processus composé d'une branche parallèle qui ne communique pas avec les autres processus du système.

12.2.2 Programmation de systèmes embarqués

Une autre direction de recherche est l'utilisation de ReactiveML pour la programmation de systèmes temps-réel. Il est possible d'identifier un sous-ensemble de ReactiveML pour lequel nous pouvons avoir des garanties temps-réel. Ce sous-ensemble du langage doit en particulier être sans création dynamique de processus. Dans ce cas, comme pour Esterel, le temps de réaction du programme et sa consommation mémoire sont bornés. Ainsi, ReactiveML pourrait être utilisé pour la programmation de systèmes embarqués.

Le premier avantage de cette approche est de permettre la description du système et de son environnement de simulation dans le même langage. Par exemple, il peut être intéressant de faire une simulation de réseaux de capteurs en ReactiveML et de raffiner le code des capteurs jusqu'à l'obtention du code qui sera effectivement embarqué dans les capteurs.

Le second avantage est l'absence de problèmes de causalité. Les problèmes de causalité étant souvent difficile à comprendre, leur absence aide au prototypage rapide des applications. Nos expériences nous ont montré que le retard pour la réaction à l'absence ne nous a pas gêné pour la programmation d'exemples de simulations. Il nous faudrait voir dans le cadre de la programmation de systèmes temps-réel si cette contrainte n'est pas trop pénalisante.

Enfin pour générer du code à exécuter sur des systèmes embarqués, il faut travailler sur la compilation du sous-ensemble de ReactiveML d'une part pour des questions de GC et d'autre part pour l'ordonnancement statique des programmes réactifs : c'est-à-dire déterminer à la compilation l'ordonnancement des différents processus parallèles comme cela est fait dans les langages synchrones classiques. De plus, l'ordonnancement statique de sous-ensembles de programmes réactifs pourrait être utilisé pour augmenter l'efficacité des interprètes ReactiveML.

12.2.3 Vérification formelle

La définition de la sémantique formelle de ReactiveML a permis de montrer des propriétés du langage (déterminisme, sûreté du typage, ...). Une autre étape est la vérification de propriétés des programmes écrits en ReactiveML.

Une première approche est l'utilisation d'outils de vérification formelle. Le langage est certainement trop riche pour pouvoir utiliser directement des techniques de model checking pour vérifier des propriétés sur les programmes, les bases sémantiques de ReactiveML permettent de définir une abstraction du modèle à vérifier. De plus, la composition parallèle synchrone étant simple, cela limite les espaces d'états à vérifier.

Une seconde approche est de se placer dans un langage permettant déjà de faire de la vérification et de l'étendre avec le modèle réactif. Nous pouvons par exemple étendre l'atelier Focal [41] avec de la programmation réactive comme nous l'avons fait pour Ocaml. Focal est un environnement de développement de programmes certifiés permettant de mélanger programmes et preuves. Le développement en Focal étant basé sur des techniques de raffinement, cela ouvre également la question du raffinement de programmes réactifs.

Sixième partie

Annexes

Annexe A

Manuel de référence

A.1 Le compilateur ReactiveML

`rmlc` est le compilateur pour le langage ReactiveML. Son comportement dépend du fichier d'entrée.

Un fichier terminant par `.rml` est un fichier source ReactiveML. Il définit des types, des valeurs, des fonctions et des processus. À partir du fichier `file.rml` le compilateur produit le fichier d'interface compilée `file.rzi` et un fichier Ocaml `file.ml`.

Un fichier terminant par `.rmli` est un fichier source ReactiveML contenant l'interface d'un module. À partir du fichier `file.rmli` le compilateur produit le fichier d'interface compilée `file.rzi` et un fichier Ocaml `file.mli`.

Un fichier terminant par `.mli` est un fichier source Ocaml contenant l'interface de fonctions exportées par Ocaml. À partir du fichier `file.mli` le compilateur produit le fichier d'interface compilée `file.rzi`

L'utilisation classique du compilateur est

```
rmlc [options] -s <process> <file>.rml
```

où `<process>` est le nom du processus à exécuter. Les principales options reconnues par le compilateur sont :

- `-stdlib <dir>` Le répertoire de la bibliothèque standard.
- `-v` Affiche la version du compilateur et le chemin de la bibliothèque standard et termine.
- `-version` Affiche la version du compilateur et termine.
- `-where` Affiche le chemin de la bibliothèque standard et termine.
- `-c` Produit uniquement le fichier `.rzi`.
- `-I <dir>` Ajoute `<dir>` à la liste des répertoires à inclure.
- `-s <proc>` Exécute le processus principal `<proc>`.
- `-n <n>` Exécute le processus principal au plus pendant `<n>` instants.
- `-sampling <rate>` Fixe le temps d'échantillonnage à `<rate>` seconds.
- `-i` Affiche les types.

`-dtypes` Sauvegarde l'information de type dans le fichier `<filename>.rannot`.

`-help` Affiche la liste des options.

Le fichier OCAML produit peut être compilé par :

```
ocamlc -I `rmlc -where` unix.cma rml_interpreter.cma <obj_files> <file>.ml
```

A.2 Syntaxe concrète de ReactiveML

Remarque préliminaire

Cette partie du manuel est directement reprise du manuel de référence de OCAML [63].

Notation

La syntaxe du langage est donnée avec une syntaxe à la BNF. Les symboles terminaux ont une police machine à écrire (`comme cela`). Les symboles non terminaux sont en italique (*comme cela*). Les crochets [...] représentent des composants optionnels. Les accolades {...} représentent zéro, un ou une répétition de composants. Les accolades avec un plus représentent un ou une répétition de composants. Les parenthèses (...) représentent un regroupement.

A.2.1 Conventions lexicales

Blancs et commentaires

Les conventions pour les blancs et les commentaires sont les mêmes que pour OCAML.

Identificateurs

$$\begin{array}{rcl}
\textit{ident} & ::= & \textit{lowercase-ident} \mid \textit{capitalized-ident} \\
\textit{lowercase-ident} & ::= & (\textit{lower} \mid \texttt{_})\ \{\textit{letter} \mid \texttt{0}\ldots\texttt{9} \mid \texttt{_}\} \\
\textit{capitalized-ident} & ::= & \textit{upper}\ \{\textit{letter} \mid \texttt{0}\ldots\texttt{9} \mid \texttt{_}\} \\
\textit{letter} & ::= & \textit{lower} \mid \textit{upper} \\
\textit{lower} & ::= & \texttt{a}\ldots\texttt{z} \\
\textit{upper} & ::= & \texttt{A}\ldots\texttt{Z}
\end{array}$$

Entiers

$$\begin{array}{rcl}
\textit{integer-literal} & ::= & [\texttt{-}]\ (\texttt{0}\ldots\texttt{9})\ \{\texttt{0}\ldots\texttt{9} \mid \texttt{_}\} \\
& \mid & [\texttt{-}]\ (\texttt{0x} \mid \texttt{0X})\ (\texttt{0}\ldots\texttt{9} \mid \texttt{A}\ldots\texttt{F} \mid \texttt{a}\ldots\texttt{f})\ \{\texttt{0}\ldots\texttt{9} \mid \texttt{A}\ldots\texttt{F} \mid \texttt{a}\ldots\texttt{f} \mid \texttt{_}\} \\
& \mid & [\texttt{-}]\ (\texttt{0o} \mid \texttt{0O})\ (\texttt{0}\ldots\texttt{7})\ \{\texttt{0}\ldots\texttt{7} \mid \texttt{_}\} \\
& \mid & [\texttt{-}]\ (\texttt{0b} \mid \texttt{0B})\ (\texttt{0}\ldots\texttt{1})\ \{\texttt{0}\ldots\texttt{1} \mid \texttt{_}\}
\end{array}$$

Nombres à virgule flottante

float-literal ::= [-] (`0`...`9`) {`0`...`9` | `_`} [`.` {`0`...`9` | `_`}] [(`e` | `E`) [`+` | `-`] (`0`...`9`) {`0`...`9` | `_`}]

Caractères

char-literal ::= `'` *regular-char* `'`
    | `'` *escape-sequence* `'`
escape-sequence ::= `\` (`\` | `"` | `'` | `n` | `t` | `b` | `r`)
    | `\` (`0`...`9`) (`0`...`9`) (`0`...`9`)
    | `\x` (`0`...`9` | `A`...`F` | `a`...`f`) (`0`...`9` | `A`...`F` | `a`...`f`)

Chaînes de caractères

string-literal ::= `"` {*string-character*} `"`
string-character ::= *regular-char-str*
    | *escape-sequence*

Symboles préfixes et infixes

infix-symbol ::= (`=` | `<` | `>` | `@` | `^` | `|` | `&` | `+` | `-` | `*` | `/` | `$` | `%`) {*operator-char*}
prefix-symbol ::= (`!` | `?` | `~`) {*operator-char*}
operator-char ::= `!` | `$` | `%` | `&` | `*` | `+` | `-` | `.` | `/` | `:` | `<` | `=` | `>` | `?` | `@` | `^` | `|` | `~`

Mots clés

Les identificateurs suivant sont les mots clés de OCAML :

```
and         as          assert      asr         begin       class
constraint  do          done        downto      else        end
exception   external    false       for         fun         function
functor     if          in          include     inherit     initializer
land        lazy        let         lor         lsl         lsr
lxor        match       method      mod         module      mutable
new         object      of          open        or          private
rec         sig         struct      then        to          true
try         type        val         virtual     when        while
with
```

La liste suivante contient les mots clés ajoutés par ReactiveML :

`await`	`control`	`default`	`dopar`	`emit`	`gather`
`immediate`	`loop`	`nothing`	`one`	`pause`	`pre`
`present`	`process`	`run`	`signal`	`until`	

Les caractères suivant sont aussi des mots clés :

`!=`	`#`	`&`	`&&`	`'`	`(`	`)`	`*`	`+`	`,`	`-`
`-.`	`->`	`.`	`..`	`:`	`::`	`:=`	`:>`	`;`	`;;`	`<`
`<-`	`=`	`>`	`>]`	`>}`	`?`	`??`	`[`	`[<`	`[>`	`[|`
`]`	`_`	`` ` ``	`{`	`{<`	`|`	`|]`	`}`	`~`	`||`	

A.2.2 Valeurs

Les valeurs Ocaml sont aussi des valeurs ReactiveML. Les processus et les signaux sont aussi des valeurs.

A.2.3 Noms

Les identificateurs sont utilisés pour nommer plusieurs classes d'objets :
- les noms de valeurs (*value-name*),
- les constructeurs de valeurs et d'exceptions (*constr-name*),
- les constructeurs de type (*typeconstr-name*),
- les champs d'enregistrement (*field-name*),
- les noms de modules (*module-name*),

Ces espaces de noms sont distingués par le contexte et la casse de la première lettre des identificateurs.

Nommage des objets

value-name	::=	*lowercase-ident*
	|	`(` *operator-name* `)`
operator-name	::=	*prefix-symbol* | *infix-op*
infix-op	::=	*infix-symbol*
	|	`*` | `=` | `or` | `&` | `:=`
	|	`mod` | `land` | `lor` | `lxor` | `lsl` | `lsr` | `asr`
constr-name	::=	*capitalized-ident*
typeconstr-name	::=	*lowercase-ident*
field-name	::=	*lowercase-ident*
module-name	::=	*capitalized-ident*

Référence à des objets nommés

$$
\begin{array}{rcl}
\textit{value-path} & ::= & \textit{value-name} \\
& | & \textit{module-name}\ \texttt{.}\ \textit{value-name} \\[1ex]
\textit{constr} & ::= & \textit{constr-name} \\
& | & \textit{module-name}\ \texttt{.}\ \textit{constr-name} \\[1ex]
\textit{typeconstr} & ::= & \textit{typeconstr-name} \\
& | & \textit{module-name}\ \texttt{.}\ \textit{typeconstr-name} \\[1ex]
\textit{field} & ::= & \textit{field-name} \\
& | & \textit{module-name}\ \texttt{.}\ \textit{field-name}
\end{array}
$$

A.2.4 Expressions de type

$$
\begin{array}{rcl}
\textit{typexpr} & ::= & \texttt{'}\ \textit{ident} \\
& | & \texttt{\_} \\
& | & \texttt{(}\ \textit{typexpr}\ \texttt{)} \\
& | & \textit{typexpr}\ \texttt{->}\ \textit{typexpr} \\
& | & \textit{typexpr}\ \{\texttt{*}\ \textit{typexpr}\}^{+} \\
& | & \textit{typeconstr} \\
& | & \textit{typexpr}\ \textit{typeconstr} \\
& | & \texttt{(}\ \textit{typexpr}\ \{,\textit{typexpr}\}\ \texttt{)}\ \textit{typeconstr} \\
& | & \textit{typexpr}\ \texttt{as}\ \texttt{'}\ \textit{ident}
\end{array}
$$

Le tableau suivant présente la précédence et l'associativité des opérateurs. Les constructions avec la plus forte précédence sont donnés en premier.

Operator	Associativity
Type constructor application	–
`*`	–
`->`	right
`as`	–

A.2.5 Constantes

$$
\begin{array}{rcl}
\textit{constant} & ::= & \textit{integer-literal} \\
& | & \textit{float-literal} \\
& | & \textit{char-literal} \\
& | & \textit{string-literal} \\
& | & \textit{constr} \\
& | & \texttt{false} \\
& | & \texttt{true} \\
& | & \texttt{[]} \\
& | & \texttt{()}
\end{array}
$$

A.2.6 Motifs

```
pattern ::= value-name
          | _
          | constant
          | pattern as value-name
          | ( pattern )
          | ( pattern : typexpr )
          | pattern | pattern
          | constr pattern
          | pattern {, pattern}
          | { field = pattern {; field = pattern} }
          | [ pattern {; pattern} ]
          | pattern :: pattern
          | [| pattern {; pattern} |]
```

A.2.7 Expressions

```
expr ::= value-path
       | constant
       | ( expr )
       | begin expr end
       | ( expr : typexpr )
       | expr , expr {, expr}
       | constr expr
       | expr :: expr
       | [ expr {; expr} ]
       | [| expr {; expr} |]
       | { field = expr {; field = expr} }
       | expr {expr}+
       | run expr
       | prefix-symbol expr
       | expr infix-op expr
       | expr . field          | expr . field <- expr
       | expr .( expr )        | expr .( expr ) <- expr
       | expr .[ expr ]        | expr .[ expr ] <- expr
       | if expr then expr [else expr]
       | present expr then expr [else expr]     | present expr else expr
       | while expr do expr done
       | for ident = expr (to | downto) expr (do | dopar) expr done
       | expr ; expr
       | expr || expr
       | match expr with pattern-matching
       | function pattern-matching
       | fun multiple-matching
       | process expr
       | try expr with pattern-matching
       | do expr until expr [( pattern ) -> expr] done
       | do expr when expr
       | control expr with expr
       | await [[immediate] one] signal ( pattern ) in expr
       | let lowercase-ident < pattern > in expr
       | let [rec] let-binding {and let-binding} in expr
       | signal lowercase-ident {, lowercase-ident} [default expr gather expr] in expr
       | assert expr
       | lazy expr
       | emit expr [expr]
       | await [immediate] expr
       | pre [?] expr
       | pause
       | nothing
```

pattern-matching	::=	[`|`] *pattern* `->` *expr* {`|` *pattern* `->` *expr*}
multiple-matching	::=	{*parameter*}+ - > *expr*
let-binding	::=	[`process`] *pattern* `=` *expr*
	|	[`process`] *value-name* {*parameter*} [`:` *typexpr*] `=` *expr*
parameter	::=	*pattern*

A.2.8 Définitions de types et d'exceptions

Définitions de types

type-definition	::=	`type` *typedef* {`and` *typedef*}
typedef	::=	[*type-params*] *typeconstr-name* [*type-information*]
type-information	::=	[*type-equation*]
	|	[*type-representation*]
type-equation	::=	`=` *typexpr*
type-representation	::=	`=` *constr-decl* {`|` *constr-decl*}
	|	`= {` *field-decl* {`;` *field-decl*} `}`
type-params	::=	*type-param*
	|	`(` *type-param* {`,` *type-param*} `)`
type-param	::=	`'` *ident*
constr-decl	::=	*constr-name*
	|	*constr-name* `of` *typexpr*
field-decl	::=	*field-name* `:` *typexpr*
	|	`mutable` *field-name* `:` *typexpr*

Définitions d'exceptions

exception-definition	::=	`exception` *constr-name* [`of` *typexpr*]
	|	`exception` *constr-name* `=` *constr*

A.2.9 Spécifications de modules

specification	::=	`val` *value-name* `:` *typexpr*
	|	*type-definition*
	|	*exception constr-decl*
	|	`open` *module-name*

A.2.10 Implantations de modules

definition	::=	*let* [`rec`] *let-binding* {*and let-binding*}
	|	`signal` *lowercase-ident* {`,` *lowercase-ident*} [`default` *expr* `gather` *expr*]
	|	*type-definition*
	|	*exception-definition*
	|	*open module-path*

A.2.11 Unités de compilation

unit-interface	::=	{*specification* [`;;`]}
unit-implementation	::=	{*definition* [`;;`]}

Annexe B

Sémantique Rewrite

La sémantique de ReactiveML donnée à la façon Rewrite est définie figures B.1 et B.2. Cette définition reprend celle donnée par F. Boussinot et J-F. Susini dans [27].

Il faut remarquer que nous annotons chaque branche de la composition parallèle avec son statut d'activation. Ainsi l'opérateur `let/and/in` a la syntaxe suivante :

$$\texttt{let}\ x_1 \texttt{=}_\alpha\ e_1\ \texttt{and}\ x_2 \texttt{=}_\beta\ e_2\ \texttt{in}\ e$$

La réaction de cette expression utilise les fonctions auxiliaires γ, δ_1 et δ_2 définies par :

- $$\begin{array}{lll} \gamma(\alpha,\beta) = \textsc{susp} & \text{si} & \alpha = \textsc{susp} \text{ ou } \beta = \textsc{susp} \\ \gamma(\alpha,\beta) = \textsc{term}\,((v_1,v_2)) & \text{si} & \alpha = \textsc{term}\,(v_1) \text{ et } \beta = \textsc{term}\,(v_2) \\ \gamma(\alpha,\beta) = \textsc{stop} & \text{sinon} & \end{array}$$

- $$\begin{array}{lll} \delta_1(\alpha,\beta) = \textsc{susp} & \text{si} & \alpha = \textsc{stop} \text{ et } \beta = \textsc{stop} \text{ ou } \beta = \textsc{term}\,(v_2) \\ \delta_1(\alpha,\beta) = \alpha & \text{sinon} & \end{array}$$

- $$\begin{array}{lll} \delta_2(\alpha,\beta) = \textsc{susp} & \text{si} & \beta = \textsc{stop} \text{ et } \alpha = \textsc{stop} \text{ ou } \alpha = \textsc{term}\,(v_1) \\ \delta_2(\alpha,\beta) = \beta & \text{sinon} & \end{array}$$

La construction `do/until` a une seconde forme : `do/until`*. Cette forme est utilisée pour attendre la fin d'instant.

$$\frac{0 \vdash e \qquad e/S \Downarrow v/S'}{e,\, S \xrightarrow{\textsc{term}\,(v)} v,\, S'} \qquad \frac{e/S \Downarrow n/S' \qquad n \in S' \qquad e_1,\, S' \xrightarrow{\alpha} e_1',\, S''}{\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, S \xrightarrow{\alpha} e_1',\, S''}$$

$$\frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad eoi \notin S'}{\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, S \xrightarrow{\textsc{susp}} \texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, S'}$$

$$\frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad eoi \in S'}{\texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2,\, S \xrightarrow{\textsc{stop}} e_2,\, S'}$$

$$\frac{e_1,\, S \xrightarrow{\alpha} e_1',\, S' \qquad \alpha \neq \textsc{term}\,(v)}{e_1;e_2,\, S \xrightarrow{\alpha} e_1';e_2,\, S'} \qquad \frac{e_1,\, S \xrightarrow{\textsc{term}\,(v)} e_1',\, S' \qquad e_2,\, S' \xrightarrow{\alpha} e_2',\, S''}{e_1;e_2,\, S \xrightarrow{\alpha} e_2',\, S''}$$

$$\frac{e/S \Downarrow \texttt{process}\ e_1/S_1 \qquad e_1,\, S_1 \xrightarrow{\alpha} e',\, S'}{\texttt{run}\ e,\, S \xrightarrow{\alpha} e',\, S'} \qquad \frac{}{\texttt{pause},\, S \xrightarrow{\textsc{stop}} \texttt{()},\, S}$$

$$\frac{\begin{array}{c} e_1/S \Downarrow v_1/S_1 \qquad e_2/S_1 \Downarrow v_2/S_2 \qquad n \notin Dom(S_2) \\ e[x \leftarrow n],\, S_2[(v_1, v_2, (\mathit{false}, v_1), \emptyset)/n] \xrightarrow{\alpha} e',\, S' \end{array}}{\texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e,\, S \xrightarrow{\alpha} e',\, S'}$$

$$\frac{e_1,\, S \xrightarrow{\alpha} e_1',\, S_1 \qquad e_2,\, S_1 \xrightarrow{\beta} e_2',\, S_2 \qquad \alpha' = \delta_1(\alpha,\beta) \qquad \beta' = \delta_2(\alpha,\beta)}{\texttt{let}\ x_1 \texttt{=}_{\textsc{susp}}\ e_1\ \texttt{and}\ x_2 \texttt{=}_{\textsc{susp}}\ e_2\ \texttt{in}\ e,\, S \xrightarrow{\gamma(\alpha,\beta)} \texttt{let}\ x_1 \texttt{=}_{\alpha'}\ e_1'\ \texttt{and}\ x_2 \texttt{=}_{\beta'}\ e_2'\ \texttt{in}\ e,\, S_2}$$

$$\frac{\beta \neq \textsc{susp} \qquad e_1,\, S \xrightarrow{\alpha} e_1',\, S' \qquad \alpha' = \delta_1(\alpha,\beta) \qquad \beta' = \delta_2(\alpha,\beta)}{\texttt{let}\ x_1 \texttt{=}_{\textsc{susp}}\ e_1\ \texttt{and}\ x_2 \texttt{=}_{\textsc{susp}}\ e_2\ \texttt{in}\ e,\, S \xrightarrow{\gamma(\alpha,\beta)} \texttt{let}\ x_1 \texttt{=}_{\alpha'}\ e_1'\ \texttt{and}\ x_2 \texttt{=}_{\beta'}\ e_2\ \texttt{in}\ e,\, S'}$$

$$\frac{\alpha \neq \textsc{susp} \qquad e_2,\, S \xrightarrow{\alpha} e_2',\, S' \qquad \alpha' = \delta_1(\alpha,\beta) \qquad \beta' = \delta_2(\alpha,\beta)}{\texttt{let}\ x_1 \texttt{=}_{\textsc{susp}}\ e_1\ \texttt{and}\ x_2 \texttt{=}_{\textsc{susp}}\ e_2\ \texttt{in}\ e,\, S \xrightarrow{\gamma(\alpha,\beta)} \texttt{let}\ x_1 \texttt{=}_{\alpha'}\ e_1\ \texttt{and}\ x_2 \texttt{=}_{\beta'}\ e_2'\ \texttt{in}\ e,\, S'}$$

$$\frac{e[x_1 \leftarrow v_1, x_2 \leftarrow v_2],\, S \xrightarrow{\alpha} e',\, S'}{\texttt{let}\ x_1 \texttt{=}_{\textsc{term}\,(v_1)}\ v_1\ \texttt{and}\ x_2 \texttt{=}_{\textsc{term}\,(v_2)}\ v_2\ \texttt{in}\ e,\, S \xrightarrow{\alpha} e',\, S'}$$

FIG. B.1 – Sémantique REWRITE (1).

$$\frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad eoi \notin S'}{\texttt{do } e_1 \texttt{ when } e,\, S \xrightarrow{\text{SUSP}} \texttt{do } e_1 \texttt{ when } e,\, S'} \qquad \frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad eoi \in S'}{\texttt{do } e_1 \texttt{ when } e,\, S \xrightarrow{\text{STOP}} \texttt{do } e_1 \texttt{ when } e,\, S'}$$

$$\frac{e/S \Downarrow n/S' \qquad n \in S' \qquad e_1,\, S' \xrightarrow{\text{TERM}\,(v)} v,\, S''}{\texttt{do } e_1 \texttt{ when } e,\, S \xrightarrow{\text{TERM}\,(v)} v,\, S''}$$

$$\frac{e/S \Downarrow n/S' \qquad e_1,\, S' \xrightarrow{\text{TERM}\,(v)} v,\, S''}{\texttt{do } e_1 \texttt{ until } e(x) \texttt{ -> } e_2 \texttt{ done},\, S \xrightarrow{\text{TERM}\,(v)} v,\, S''}$$

$$\frac{e/S \Downarrow n/S' \qquad e_1,\, S' \xrightarrow{\text{SUSP}} e_1',\, S''}{\texttt{do } e_1 \texttt{ until } e(x) \texttt{ -> } e_2 \texttt{ done},\, S \xrightarrow{\text{SUSP}} \texttt{do } e_1' \texttt{ until } n(x) \texttt{ -> } e_2 \texttt{ done},\, S''}$$

$$\frac{e/S \Downarrow n/S' \qquad e_1,\, S' \xrightarrow{\text{STOP}} e_1',\, S''}{\texttt{do } e_1 \texttt{ until } e(x) \texttt{ -> } e_2 \texttt{ done},\, S \xrightarrow{\text{SUSP}} \texttt{do } e_1' \texttt{ until}^* \; n(x) \texttt{ -> } e_2 \texttt{ done},\, S''}$$

$$\frac{eoi \notin S}{\texttt{do } e_1 \texttt{ until}^* \; n(x) \texttt{ -> } e_2 \texttt{ done},\, S \xrightarrow{\text{SUSP}} \texttt{do } e_1 \texttt{ until}^* \; n(x) \texttt{ -> } e_2 \texttt{ done},\, S}$$

$$\frac{eoi \in S \quad n \notin S}{\texttt{do } e_1 \texttt{ until}^* \; n(x) \texttt{ -> } e_2 \texttt{ done},\, S \xrightarrow{\text{STOP}} \texttt{do } e_1 \texttt{ until } n(x) \texttt{ -> } e_2 \texttt{ done},\, S}$$

$$\frac{eoi \in S \quad n \in S \quad S(n) = (d, g, p, m) \quad v = \mathit{fold}\ g\ m\ d}{\texttt{do } e_1 \texttt{ until}^* \; n(x) \texttt{ -> } e_2 \texttt{ done},\, S \xrightarrow{\text{STOP}} e_2[x \leftarrow v],\, S}$$

Fig. B.2 – Sémantique Rewrite (2).

Index

Bibliographie

[1] Raúl Acosta-Bermejo. *Rejo - Langage d'Objets Réactifs et d'Agents.* Thèse de doctorat, Ecole des Mines de Paris, 2003.

[2] Roberto M. Amadio. The SL synchronous langage, revisited. Technical report, Université Paris 7, November 2005.

[3] Roberto M. Amadio, Gérard Boudol, Fédéric Boussinot, and Ilaria Castellani. Reactive concurrent programming revisited. In *Extended abstract presented at the workshop Algebraic Process Calculi : the first twenty five years and beyond*, Bertinoro, August 2005.

[4] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1991.

[5] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang.* Prentice-Hall, second edition, 1996.

[6] Farid Benbadis, Marcelo Dias de Amorim, and Serge Fdida. 3P : Packets for positions prediction. In *Proceedings of IEEE INFOCOM STUDENTS WORKSHOP'05*, Miama, FL, USA, March 2005.

[7] Farid Benbadis, Marcelo Dias de Amorim, and Serge Fdida. Dissémination prédictive des coordonnées pour le routage géographique basé sur l'âge. In *Proceedings of CFIP 2005 Conference*, Bordeaux, France, March 2005.

[8] Farid Benbadis, Marcelo Dias de Amorim, and Serge Fdida. ELIP : Embedded location information protocol. In *Proceedings of IFIP Networking 2005 Conference*, Waterloo, Canada, May 2005.

[9] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE, Special issue on embedded systems*, 91(1) :64–83, January 2003.

[10] Gérard Berry. Preemption in concurrent systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'93)*, pages 72–93, London, UK, 1993. Springer-Verlag.

[11] Gérard Berry. The constructive semantics of esterel, 1998.

[12] Gérard Berry. *The Esterel-V5 Language Primer.* CMA and Inria, Sophia-Antipolis, France, June 2000.

[13] Gérard Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction : Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.

[14] Gérard Boudol. Ulm : A core programming model for global computing. In *Proceedings of the 13th European Symposium on Programming (ESOP'2004)*, pages 234–248, 2004.

[15] F. Boussinot, J-F. Susini, F. Dang Tran, and L. Hazard. A reactive behavior framework for dynamic virtual worlds. In *Proceedings of the sixth international conference on 3D Web technology*, pages 69–75. ACM Press, 2001.

[16] Frédéric Boussinot. Reactive C : An extension of C to program reactive systems. *Software Practice and Experience*, 21(4) :401–428, April 1991.

[17] Frédéric Boussinot. Réseaux de processus réactifs. Research report 1588, INRIA, 1992.

[18] Frédéric Boussinot. *La programmation réactive - Application aux systèmes communicants.* Collection technique et scientifique des télécommunications, masson edition, 1996.

[19] Frédéric Boussinot. Java Fair Threads. Research report 4139, INRIA, 2001.

[20] Frédéric Boussinot. Concurrent programming with Fair Threads : The LOFT language, 2003.

[21] Frédéric Boussinot. FairThreads : mixing cooperative and preemptive threads in C. Research report 5039, INRIA, 2003.

[22] Frédéric Boussinot. Reactive programming of cellular automata. Technical Report 5183, INRIA, 2004.

[23] Frédéric Boussinot. Simulations of self replicating loops. 2004.

[24] Frédéric Boussinot and Robert de Simone. The SL synchronous language. *Software Engineering*, 22(4) :256–266, 1996.

[25] Frédéric Boussinot and Laurent Hazard. Reactive scripts. In *Proceedings of the Third International Workshop on Real-Time Computing Systems Application (RTCSA'96)*, page 270, Washington, DC, USA, 1996. IEEE Computer Society.

[26] Frédéric Boussinot and Jean-Ferdy Susini. The SugarCubes tool box : A reactive java framework. *Software Practice and Experience*, 28(4) :1531–1550, 1998.

[27] Frédéric Boussinot and Jean-Ferdy Susini. Junior rewrite semantics. 2000.

[28] Christian Brunette. *Construction et simulation graphiques de comportements : le modèle des Icobjs.* Thèse de doctorat, Université de Nice-Sophia Antipolis, 2004.

[29] Matthieu Carlier. Développement d'applications en Reactive ML. Rapport de Magistère, Université Paris 6, September 2004.

[30] Paul Caspi and Marc Pouzet. A functional extension to Lustre. In M. A. Orgun and E. A. Ashcroft, editors, *International Symposium on Languages for Intentional Programming*, Sydney, Australia, May 3-5 1995. World Scientific.

[31] Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Proceedings of Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998.

[32] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil. SAXO-RT : interpreting Esterel semantic on a sequential execution structure. In *Electronic Notes in Theoretical Computer Science*, volume 65, pages 864–878. Elsevier Science Publishers, 2002.

[33] Julien Cohen. *Intégration des collections topologiques et des transformations dans un langage fonctionnel.* Thèse de doctorat, Université d'Evry-val d'Essonne, 2004.

[34] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proceedings of ACM International Conference on Embedded Software (EMSOFT'05)*, September 2005.

[35] Julien Demaria. Programmation réactive fonctionnelle avec Senior. Rapport de DEA, ESSI, 2001.

[36] Damien Doligez. *Conception, réalisation et certification d'un glaneur de cellules concurrent.* Thèse de doctorat, Université Paris 7, May 1995.

[37] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux. 2005.

[38] Stephen A. Edwards. Compiling esterel into sequential code. In *Proceedings of the 37th Design Automation Conference (DAC'2000)*, pages 322–327, June 5-9 2000.

[39] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming (ICFP'97)*, pages 263–273, New York, NY, USA, 1997. ACM Press.

[40] Stéphane Epardaud. Mobile reactive programming in ulm. In *Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming*, 2004.

[41] Focal.
`http://focal.inria.fr`.

[42] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'96)*, pages 372–385, New York, NY, USA, 1996. ACM Press.

[43] Cédric Fournet and Luc Maranget. *The Join-Calculus language, release 1.05.* INRIA.
`http://join.inria.fr`.

[44] The gnu portable threads.
`http://www.gnu.org/software/pth/`.

[45] Georges Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones ; application à Esterel.* Thèse de doctorat, Ecole des Mines de Paris, 1988.

[46] James Gosling, Bill Joy, and Guy Steeleand Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley, 2005.

[47] Matthias Grossglauser and Martin Vetterli. Locating nodes with EASE : Last encounter routing in ad hoc networks through mobility diffusion. In *Proceedings of IEEE INFOCOM'03*, March 2003.

[48] Gtk+, the GIMP toolkit.
`http://www.gtk.org`.

[49] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Lemaire. Programming real-time applications with signal. *Proceedings of IEEE*, 79(9) :1321–1336, September 1991.

[50] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic, 1993.

[51] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language lustre. *Proceedings of IEEE*, 79(9) :1305–1320, September 1991.

[52] Grégoire Hamon and Marc Pouzet. Un simulateur synchrone pour Lucid Synchrone. In *Journées Francophones des Langages Applicatifs (JFLA'99)*, Morzine-Avoriaz, February 1999. INRIA.

[53] David Harel and Amir Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pages 477–498, 1985.

[54] Fabrice Harrouet. *oRis : s'immerger par le langage pour le prototypage d'univers virtuels à base d'entités autonomes.* Thèse de doctorat, Université de Bretagne Occidentale, 2000.

[55] Laurent Hazard. Simple. An efficient implementation of Junior.

[56] Laurent Hazard, Jean-Ferdy Susini, and Frédéric Boussinot. The Junior reactive kernel. Research report 3732, INRIA, 1999.

[57] Laurent Hazard, Jean-Ferdy Susini, and Frédéric Boussinot. Programming with Junior. Research report 4027, INRIA, 2000.

[58] Erwan Jahier and Pascal Raymond. The lucky language reference manual. Technical report, Unité Mixte de Recherche 5104 CNRS - INPG - UJF, 2004.

[59] Gilles Kahn. The semantics of simple language for parallel programming. In *Proceedings of IFIP 74 Conference*, pages 471–475, 1974.

[60] Brian W. Kernighan and Denis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[61] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with threads.* SunSoft Press, Mountain View, CA, USA, 1996.

[62] Xavier Leroy. Typage et programmation. Notes de cours de DEA, INRIA, 2001.

[63] Xavier Leroy. The Objective Caml system release 3.09 Documentation and user's manual. Technical report, INRIA, 2005.

[64] LURC : Light ULM/reactive library for C.
`http://www-sop.inria.fr/mimosa/Stephane.Epardaud/lurc`.

[65] Sarah Maarek. Mode Emacs pour Reactive ML. Rapport de Licence, Université Paris 7, September 2004.

[66] Louis Mandel. Aspects dynamiques dans les langages synchrones : la cas des SugarCubes. Rapport de DEA, Université Paris 6, September 2002.

[67] Louis Mandel and Farid Benbadis. Simulation of mobile ad hoc network protocols in ReactiveML. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'05)*, Edinburgh, Scotland, April 2005. Electronic Notes in Theoretical Computer Science.

[68] Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. In *Proceedings of 7th ACM SIGPLAN International conference on Principles and Practice of Declarative Programming (PPDP'05)*, July 2005.

[69] Louis Mandel and Marc Pouzet. ReactiveML, un langage pour la programmation réactive en ML. In *Journées Francophones des Langages Applicatifs (JFLA'05)*, Obernai, France, March 2005. INRIA.

[70] Florence Maraninchi. The Argos Language : Graphical Representation of Automata and Description of Reactive Systems. In *Proceedings of the IEEE, Workshop on Visual Languages*, October 1991.

[71] MetaOCaml.
`http://www.metaocaml.org/`.

[72] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[73] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, 1978.

[74] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3) :267–310, 1983.

[75] Robin Milner. *Communicating and Mobile Systems : The π-Calculus*. Cambridge University Press, 1999.

[76] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[77] Edward F. Moore. Gedanken-experiments on sequential machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 129–153. Princeton University Press, 1956.

[78] Network in A Box.
`http://nab.epfl.ch/`.

[79] The Network Simulator.
`http://www.isi.edu/nsnam/ns`.

[80] OPNET Modeler.
`http://www.opnet.com`.

[81] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[82] John K. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk, USENIX Technical Conference, January 1996.

[83] Olivier Parra. Programmation réactive sur systèmes embarqués. Rapport de DEA, UNSA, 2003.

[84] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2) :340–377, 2000.

[85] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries : The Revised Report*. Cambridge University Press, April 2003.

[86] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'96)*, pages 295–308, New York, NY, USA, 1996. ACM Press.

[87] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[88] Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1 :125–159, 1975.

[89] Dumitru Potop-Butucaru. *Generation of Fast C Code from Esterel Programs*. Thèse de doctorat, ENSMP/CMA, 2002.

[90] Marc Pouzet. *Lucid Synchrone, version 3.0*. LRI, 2005.

[91] Riccardo Pucella. Reactive programming in Standard ML. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 48–57. IEEE Computer Society Press, 1998.

[92] Reactive programming.
`http://www-sop.inria.fr/mimosa/rp`.

[93] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[94] Nadine Richard. *Description de comportements d'agents autonomes évoluant dans des mondes virtuels habités*. Thèse de doctorat, ENST, 2001.

[95] Alexander Samarin. Application de la programmation réactive à la modélisation en physique. Rapport de DEA, Université de Nice Sophia Antipolis, 2002.

[96] Ludovic Samper, Florence Maraninchi, Laurent Mounier, and Louis Mandel. GLONEMO : Global and accurate formal models for the analysis of ad-hoc sensor networks. In *Proceedings of the First International Conference on Integrated Internet Ad hoc and Sensor Networks (InterSense'06)*, Nice, France, May 2006.

[97] Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme Fair Threads. In *Proceedings of 6th ACM SIGPLAN International conference on Principles and Practice of Declarative Programming (PPDP'04)*, pages 203–214, 2004.

[98] Programming with Java Fair Threads.
`http://www-sop.inria.fr/mimosa/rp/FairThreads/FTJava/DemosFairThreads`.

[99] Standard ML of New Jersey user's guide.
`http://www.smlnj.org/doc`.

[100] Jean-Ferdy Susini. *L'approche réactive au dessus de Java : sémantique et implémentation des SugarCubes et de Junior*. Thèse de doctorat, Ecole des Mines de Paris, 2001.

[101] Walid Taha. *Multi-Stage Programming : Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[102] Olivier Tardieu. *Loops in Esterel : From Operational Semantics to Formally Specified Compilers*. Thèse de doctorat, Ecole des Mines de Paris, 2004.

[103] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1) :1–34, 1990.

[104] Julien Verlaguet and Emmanuel Chailloux. HirondML : Fair Threads migrations for Objective Caml. In *Proceedings of the Third International Workshop on High-level Parallel Programming and Applications (HLPP 2005)*, July 2005.

[105] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI'00)*, pages 242–252, New York, NY, USA, 2000. ACM Press.

[106] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4) :343–355, 1995.

[107] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1) :38–94, 1994.